Programming Concepts

Correctness of algorithms

Topics

1. Reasoning about programs

Topics

1. Reasoning about programs in terms of **correctness**

Topics

1. Reasoning about programs in terms of **correctness**
   ⇝ will make use of **assertions**
2. Lots of examples

The problem

Terrible things go wrong with software projects

- American spaceship Mariner sent to Venus in 1960s lost forever - due to software error.
- Mars Climate Orbiter. Wrong units of measurement...
- Software bug in radiation machine in Texas caused numerous deaths due to radiation overdoses
- See The Risks Digest (`http://catless.ncl.ac.uk/Risks/`) for endless examples

The problem

Terrible things go wrong with software projects

- American spaceship Mariner sent to Venus in 1960s lost forever - due to software error.
- Mars Climate Orbiter. Wrong units of measurement...
- Software bug in radiation machine in Texas caused numerous deaths due to radiation overdoses
- See The Risks Digest (`http://catless.ncl.ac.uk/Risks/`) for endless examples

More and more agencies are demanding certified software

Programming errors

1. Language errors:

    e.g. incorrect use of syntax

    Normally spotted by complier or interpreter

Programming errors

1. Language errors:

   e.g. incorrect use of syntax

   Normally spotted by complier or interpreter

2. Logical errors, e.g.:

   $\mathrm{J} \leftarrow 1$

   $\textsc{total} \leftarrow 0$

   while $\mathrm{J} \neq 100$ **do**

   $\quad \textsc{total} \leftarrow \textsc{total} + \mathrm{J}$

   $\quad \mathrm{J} \leftarrow \mathrm{J} + 2$

   Normally due to flaws in underlying algorithm

Debugging programs

Logical errors can lie undetected for ages

Test suites can be developed to exercise different execution paths through the program

Debugging programs

Logical errors can lie undetected for ages

Test suites can be developed to exercise different execution paths through the program

Dijkstra:

Debugging

- cannot be used to demonstrate the *absence* of bugs
- only their presence.

Debugging programs

Logical errors can lie undetected for ages

Test suites can be developed to exercise different execution paths through the program

Dijkstra:

Debugging

- cannot be used to demonstrate the *absence* of bugs
- only their presence.

We need methods to develop methods for formally proving that programs are correct:

Debugging programs

Logical errors can lie undetected for ages

Test suites can be developed to exercise different execution paths through the program

Dijkstra:

Debugging

- cannot be used to demonstrate the *absence* of bugs
- only their presence.

We need methods to develop methods for formally proving that programs are correct: all possible runs of program will produce the correct results

What does correctness mean ?

Recall: Specification of an algorithmic problem:

- a characterisation of all legal inputs
- a description of the required outputs as a function of the inputs

What does correctness mean ?

Recall: Specification of an algorithmic problem:

- a characterisation of all legal inputs
- a description of the required outputs as a function of the inputs

Partial correctness:

An algorithm is *partially correct* with respect to a specification whenever, for every legal input

- if the algorithm halts, then the output satisfies the required relationship with the original input

What does correctness mean ?

Recall: Specification of an algorithmic problem:

- a characterisation of all legal inputs
- a description of the required outputs as a function of the inputs

Partial correctness:

An algorithm is *partially correct* with respect to a specification whenever, for every legal input

- if the algorithm halts, then the output satisfies the required relationship with the original input

Total correctness:

An algorithm is *totally correct* with respect to a specification whenever, for every legal input

- the algorithm halts
- when the algorithm halts, then the output satisfies the required relationship with the original input

What you need to understand

How to show rigorously that an algorithm is correct
for a given problem specification

What you need to understand

How to show rigorously that an algorithm is correct for a given problem specification

How do we do that? **Assertions**

Assertions

What is an assertion?

A mathematical statement about program variables at some specific checkpoint in a program

Assertions

What is an assertion?

A mathematical statement about program variables at some specific checkpoint in a program

Examples:

What is an assertion?

A mathematical statement about program variables at some specific checkpoint in a program

Examples:

Input: Array $A[1 \ldots n]$

Output: largest element of A

$\textsc{ptr} \leftarrow 1$

$\textsc{cMax} \leftarrow A[\textsc{ptr}]$

$\Longleftarrow \boxed{\textsc{cMax} = A[1]}$

while $\textsc{ptr} \neq n$ **do**

  $\textsc{ptr} \leftarrow \textsc{ptr} + 1$

  if $A[\textsc{ptr}] > \textsc{cMax}$ **then**

    $\textsc{cMax} \leftarrow A[\textsc{ptr}]$

return $\textsc{cMax}$

Assertions

What is an assertion?

A mathematical statement about program variables at some specific checkpoint in a program

Examples:

Input: Array $A[1 \dots n]$

Output: largest **...**

$\textsc{ptr} \leftarrow 1$

$\textsc{cMax} \leftarrow A[\textsc{ptr}]$

while $\textsc{ptr} \neq n$ **do**

  $\Longleftarrow \boxed{\textsc{cMax} = A[\textsc{ptr}]}$

  $\textsc{ptr} \leftarrow \textsc{ptr} + 1$

  if $A[\textsc{ptr}] > \textsc{cMax}$ **then**

    $\textsc{cMax} \leftarrow A[\textsc{ptr}]$

return $\textsc{cMax}$

What makes an assertion valid?

What is an assertion?

A mathematical statement about program variables at some specific checkpoint in a program

What makes an assertion valid?

What is an assertion?

A mathematical statement about program variables at some specific checkpoint in a program

Valid assertions

An assertion is valid if it is true of the program variables every time control passes the checkpoint

Valid assertions

An assertion is valid if it is true of the program variables *every time* control passes the checkpoint

Which are valid?

Valid assertions

An assertion is valid if it is true of the program variables

every time control passes the checkpoint

Which are valid?

Input: Array $A[1 \ldots n]$

Output: largest element of A

$\textsc{ptr} \leftarrow 1$

$\textsc{cMax} \leftarrow A[\textsc{ptr}]$

$\Leftarrow \boxed{\text{Prop}}$

while $\textsc{ptr} \neq n$ **do**

  $\textsc{ptr} \leftarrow \textsc{ptr} + 1$

  if $A[\textsc{ptr}] > \textsc{cMax}$ **then**

    $\textsc{cMax} \leftarrow A[\textsc{ptr}]$

return $\textsc{cMax}$

When $\boxed{\text{Prop}}$ is

- $\textsc{cMax} = A[1]$?
- $A[\textsc{ptr}] = A[1] + A[n]$?
- $\textsc{cMax} \geq A[\textsc{ptr}]$?

Valid assertions

An assertion is valid if it is true of the program variables every time control passes the checkpoint

Which are valid?

Input: Array $A[1 \dots n]$
Output: largest ...

```
PTR ← 1
CMAX ← A[PTR]
while PTR ≠ n do
    ⟸ Prop
    PTR ← PTR + 1
    if A[PTR] > CMAX then
        CMAX ← A[PTR]
return CMAX
```

When $\boxed{\text{Prop}}$ is

- $\textsc{cMax} = A[\textsc{ptr}]$?
- $\textsc{cMax} \geq A[1]$?
- $A[1] \leq A[\textsc{ptr}]$?

Important assertions

- **Initial assertion:** Captures requirements on legal inputs
- **Final assertion:** Determines properties of data output
- **Loop assertions:** Difficult to assign.
  Must be true every time control goes through the loop.

Important assertions

- **Initial assertion:** Captures requirements on legal inputs
- **Final assertion:** Determines properties of data output
- **Loop assertions:** Difficult to assign.
  Must be true every time control goes through the loop.

```
Input: Array A[1 . . . n]
Output: largest . . .
⟸ [InitialProp]
PTR ← 1
CMAX ← A[PTR]
while PTR ≠ n do
    ⟸ [LoopProp]
    PTR ← PTR + 1
    if A[PTR] > CMAX then
        CMAX ← A[PTR]
⟸ [FinalProp]
return CMAX
```

Assigning valid assertions

- **Initial assertion:** Captures requirements on legal inputs
- **Final assertion:** Determines properties of data output
- **Loop assertions:** Difficult to assign
  Must be true every time control goes through the loop

Input: Array $A[1 \dots n]$
Output: largest ...

$\Longleftarrow$ $\boxed{n \geq 1}$

PTR $\leftarrow$ 1
cMAX $\leftarrow$ A[PTR]
while PTR $\neq n$ **do**
  $\Longleftarrow$ $\boxed{\text{LoopProp ???}}$
  PTR $\leftarrow$ PTR + 1
  if A[PTR] $>$ cMAX **then**
    cMAX $\leftarrow$ A[PTR]

$\Longleftarrow$ $\boxed{\text{cMAX is largest element in } A}$
return cMAX

Assigning valid assertions

- **Initial assertion:** Captures requirements on legal inputs
- **Final assertion:** Determines properties of data output
- **Loop assertions:** Difficult to assign

  Must be true every time control goes through the loop

Input: Array $A[1 \dots n]$
Output: largest ...

$\Longleftarrow$ $\boxed{n \geq 1}$

$\textsc{ptr} \leftarrow 1$
$\textsc{cMax} \leftarrow A[\textsc{ptr}]$
while $\textsc{ptr} \neq n$ **do**

  $\Longleftarrow$ $\boxed{\text{LoopProp ???}}$

  $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
  if $A[\textsc{ptr}] > \textsc{cMax}$ **then**
    $\textsc{cMax} \leftarrow A[\textsc{ptr}]$

$\Longleftarrow$ $\boxed{\textsc{cMax} \text{ is largest element in } A}$
return $\textsc{cMax}$

that is:

- $\textsc{cMax} \geq A[j]$, for j between 1 and n
- $\textsc{cMax}$ occurs in A

How do we assign valid assertions?

We use

- informal mathematical reasoning
- Loop Invariance theorems

$\rightsquigarrow$ best explained via an example

There are semi-automatic software systems

(it is a mathematical fact that you can't do
this completely automatically: see Rice's theorem)

Assigning valid assertions

Input: Array $A[1 \dots n]$
Output: largest element in A

$\textsc{ptr} \leftarrow 1$
$\textsc{cMax} \leftarrow A[\textsc{ptr}]$

while $\textsc{ptr} \neq n$ **do**
  $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
  if $A[\textsc{ptr}] > \textsc{cMax}$ **then**
    $\textsc{cMax} \leftarrow A[\textsc{ptr}]$

return $\textsc{cMax}$

Assigning valid assertions

Input: Array $A[1 \dots n]$

Output: largest element in A

$\Longleftarrow \boxed{n \geq 1}$

$\textsc{ptr} \leftarrow 1$

$\textsc{cMax} \leftarrow A[\textsc{ptr}]$

while $\textsc{ptr} \neq n$ **do**

  $\textsc{ptr} \leftarrow \textsc{ptr} + 1$

  if $A[\textsc{ptr}] > \textsc{cMax}$ **then**

    $\textsc{cMax} \leftarrow A[\textsc{ptr}]$

return $\textsc{cMax}$

Assigning valid assertions

Input: Array $A[1 \ldots n]$

Output: largest element in A

$\Longleftarrow \boxed{n \geq 1}$

$\textsc{ptr} \leftarrow 1$

$\textsc{cMax} \leftarrow A[\textsc{ptr}]$

$\Longleftarrow \boxed{\textsc{cMax} = A[1]}$

while $\textsc{ptr} \neq n$ **do**

  $\textsc{ptr} \leftarrow \textsc{ptr} + 1$

  if $A[\textsc{ptr}] > \textsc{cMax}$ **then**

    $\textsc{cMax} \leftarrow A[\textsc{ptr}]$

return $\textsc{cMax}$

Assigning valid assertions

Input: Array $A[1 \ldots n]$

Output: largest element in A

$\Longleftarrow \boxed{n \geq 1}$

$\textsc{ptr} \leftarrow 1$

$\textsc{cMax} \leftarrow A[\textsc{ptr}]$

$\Longleftarrow \boxed{\textsc{cMax} = A[1]}$

while $\textsc{ptr} \neq n$ **do**

  $\Longleftarrow$ $\boxed{\textsc{cMax} \text{ is largest element in } A[1 \ldots \textsc{ptr}]}$

  $\textsc{ptr} \leftarrow \textsc{ptr} + 1$

  if $A[\textsc{ptr}] > \textsc{cMax}$ **then**

    $\textsc{cMax} \leftarrow A[\textsc{ptr}]$

return $\textsc{cMax}$

Assigning valid assertions

Input: Array $A[1 \ldots n]$

Output: largest element in A

$\Longleftarrow \boxed{n \geq 1}$

$\text{PTR} \leftarrow 1$

$\text{CMAX} \leftarrow A[\text{PTR}]$

$\Longleftarrow \boxed{\text{CMAX} = A[1]}$

while $\text{PTR} \neq n$ **do**

  $\Longleftarrow \boxed{\text{CMAX is largest element in } A[1 \ldots \text{PTR}]}$

  $\text{PTR} \leftarrow \text{PTR} + 1$

  if $A[\text{PTR}] > \text{CMAX}$ **then**

    $\text{CMAX} \leftarrow A[\text{PTR}]$

$\Longleftarrow \boxed{\text{PTR} = n \text{ and CMAX is largest element in } A[1 \ldots \text{PTR}]}$

return CMAX

Assigning valid assertions

Input: Array $A[1 \dots n]$

Output: largest element in A

$\Longleftarrow$ $n \geq 1$

$\textsc{ptr} \leftarrow 1$

$\textsc{cMax} \leftarrow A[\textsc{ptr}]$

$\Longleftarrow$ $\textsc{cMax} = A[1]$

while $\textsc{ptr} \neq n$ **do**

  $\Longleftarrow$ $\textsc{cMax}$ is largest element in $A[1 \dots \textsc{ptr}]$

  $\textsc{ptr} \leftarrow \textsc{ptr} + 1$

  if $A[\textsc{ptr}] > \textsc{cMax}$ **then**

    $\textsc{cMax} \leftarrow A[\textsc{ptr}]$

$\Longleftarrow$ $\textsc{ptr} = n$ and $\textsc{cMax}$ is largest element in $A[1 \dots \textsc{ptr}]$

return $\textsc{cMax}$

$\Longleftarrow$ largest element in A returned

Assigning valid assertions

Input: Array $A[1 \ldots n]$

Output: largest element in A

$\Longleftarrow$ $\boxed{n \geq 1}$

$\text{PTR} \leftarrow 1$

$\text{CMAX} \leftarrow A[\text{PTR}]$

$\Longleftarrow$ $\boxed{\text{CMAX} = A[1]}$

while $\text{PTR} \neq n$ **do**

$\Longleftarrow$ $\boxed{\text{CMAX is largest element in } A[1 \ldots \text{PTR}]}$ Where did this come from ?

$\text{PTR} \leftarrow \text{PTR} + 1$

if $A[\text{PTR}] > \text{CMAX}$ **then**

$\text{CMAX} \leftarrow A[\text{PTR}]$

$\Longleftarrow$ $\boxed{\text{PTR} = n \text{ and CMAX is largest element in } A[1 \ldots \text{PTR}]}$

return CMAX

$\Longleftarrow$ $\boxed{\text{largest element in A returned}}$

The annotated algorithm as comments

Input: Array $A[1 \dots n]$
Output: largest …

$\textsc{ptr} \leftarrow 1$
$\textsc{cMax} \leftarrow A[\textsc{ptr}]$
`// assert:` $\textsc{cMax} = A[1]$
while $\textsc{ptr} \neq n$ **do**
  `// assert:` $\textsc{cMax}$ `is largest element in` $A[1 \dots \textsc{ptr}]$
  if $A[\textsc{ptr}] > \textsc{cMax}$ **then**
    $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
    $\textsc{cMax} \leftarrow A[\textsc{ptr}]$
`// assert:`$\textsc{ptr} = n$ `and` $\textsc{cMax}$ `is largest element in`
`//` $A[1 \dots \textsc{ptr}]$
return $\textsc{Max}$
`// assert: largest element in A returned`

Warning

Termination

- Assertions say nothing about termination
- Final assertion describes properties which are true if the program terminates
- Separate reasoning needs to be done to assure termination
  - Usually involving some quantity which decrease each time round the loop
  - Sometimes involves the manner in which this quantity decreases

Termination

- Assertions say nothing about termination
- Final assertion describes properties which are true if the program terminates
- Separate reasoning needs to be done to assure termination
  - Usually involving some quantity which decrease each time round the loop
  - Sometimes involves the manner in which this quantity decreases
- Never write a loop, without knowing why it will terminate, for every possible input

An algorithm

What does this do?

Input: number $k \geq 0$
Output: ?????

$\mathrm{X} \leftarrow 0$

$\mathrm{Y} \leftarrow 0$

while $\mathrm{X} \neq k$ **do**

  $\mathrm{Y} \leftarrow \mathrm{Y} + k$

  $\mathrm{X} \leftarrow \mathrm{X} + 1$

return Y

An algorithm

Input: number $k \geq 0$
Output: $k * k$

$\text{X} \leftarrow 0$

$\text{Y} \leftarrow 0$

while $\text{X} \neq k$ **do**

  $\text{Y} \leftarrow \text{Y} + k$

  $\text{X} \leftarrow \text{X} + 1$

return Y

Assigning valid assertions

Input: number $k \geq 0$

Output: $k * k$

$\textsc{x} \leftarrow 0$

$\textsc{y} \leftarrow 0$

while $\textsc{x} \neq k$ **do**

  $\textsc{y} \leftarrow \textsc{y} + k$

  $\textsc{x} \leftarrow \textsc{x} + 1$

return $\textsc{y}$

Assigning valid assertions

Input: number $k \geq 0$

Output: $k * k$

$\Longleftarrow \boxed{k \geq 0}$

$\text{X} \leftarrow 0$

$\text{Y} \leftarrow 0$

while $\text{X} \neq k$ **do**

$\quad \text{Y} \leftarrow \text{Y} + k$

$\quad \text{X} \leftarrow \text{X} + 1$

return Y

Assigning valid assertions

Input: number $k \geq 0$

Output: $k * k$

$\Longleftarrow \boxed{k \geq 0}$

$\mathrm{X} \leftarrow 0$

$\mathrm{Y} \leftarrow 0$

while $\mathrm{X} \neq k$ **do**

$\Longleftarrow$ $y = x * k$ a loop invariant

$\mathrm{Y} \leftarrow \mathrm{Y} + k$

$\mathrm{X} \leftarrow \mathrm{X} + 1$

return Y

Assigning valid assertions

Input: number $k \geq 0$

Output: $k * k$

$\Longleftarrow \boxed{k \geq 0}$

$\mathrm{X} \leftarrow 0$

$\mathrm{Y} \leftarrow 0$

$\Longleftarrow \boxed{y = x * k}$

while $\mathrm{X} \neq k$ **do**

  $\Longleftarrow \boxed{y = x * k \quad \text{a loop invariant}}$

  $\mathrm{Y} \leftarrow \mathrm{Y} + k$

  $\mathrm{X} \leftarrow \mathrm{X} + 1$

return Y

Assigning valid assertions

Input: number $k \geq 0$

Output: $k * k$

$\Longleftarrow$ $\boxed{k \geq 0}$

$\mathrm{X} \leftarrow 0$

$\mathrm{Y} \leftarrow 0$

$\Longleftarrow$ $\boxed{y = x * k}$

while $\mathrm{X} \neq k$ **do**

  $\Longleftarrow$ $\boxed{y = x * k \quad \text{a loop invariant}}$

  $\mathrm{Y} \leftarrow \mathrm{Y} + k$

  $\mathrm{X} \leftarrow \mathrm{X} + 1$

$\Longleftarrow$ $\boxed{y = x * k \text{ and } x = k}$

return Y

Assigning valid assertions

Input: number $k \geq 0$

Output: $k * k$

$\Longleftarrow$ $\boxed{k \geq 0}$

$\mathrm{X} \leftarrow 0$

$\mathrm{Y} \leftarrow 0$

$\Longleftarrow$ $\boxed{y = x * k}$

while $\mathrm{X} \neq k$ **do**

    $\Longleftarrow$ $\boxed{y = x * k}$ a loop invariant

    $\mathrm{Y} \leftarrow \mathrm{Y} + k$

    $\mathrm{X} \leftarrow \mathrm{X} + 1$

$\Longleftarrow$ $\boxed{y = x * k \text{ and } x = k}$

return Y

$\Longleftarrow$ $\boxed{k * k \text{ returned}}$

Assigning valid assertions

Input: number $k \geq 0$

Output: $k * k$

$\Longleftarrow \boxed{k \geq 0}$

$\mathrm{X} \leftarrow 0$

$\mathrm{Y} \leftarrow 0$

$\Longleftarrow \boxed{y = x * k}$

while $\mathrm{X} \neq k$ **do**

$\quad \Longleftarrow \boxed{y = x * k \qquad \text{a loop invariant}}$

$\quad \mathrm{Y} \leftarrow \mathrm{Y} + k$

$\quad \mathrm{X} \leftarrow \mathrm{X} + 1$

$\Longleftarrow \boxed{y = x * k \text{ and } x = k}$

return Y

$\Longleftarrow \boxed{k * k \text{ returned}}$

Finding the most appropriate loop invariant is the creative step

The annotated algorithm

Input: integer k

Output: $k * k$

```
// assert: k ≥ 0
X ← 0
Y ← 0
// assert:  Y = X * k
while X ≠ k do
  // assert:  Y = X * k
  Y ← Y + k
  X ← X + 1
// assert:  Y = X * k and
    X = k
return Y
// assert: k * k returned
```

The annotated algorithm

Input: integer k

Output: $k * k$

```
// assert: k ≥ 0
X ← 0
Y ← 0
// assert: Y = X * k
while X ≠ k do
    // assert: Y = X * k
    Y ← Y + k
    X ← X + 1
// assert: Y = X * k and
    X = k
return Y
// assert: k * k returned
```

k is a logical variable

- is unchanged by execution

Reasoning about programs

Proving correctness of an algorithm can be hard,
so we break it into smaller problems.

Proving correctness of an algorithm can be hard, so we break it into smaller problems.

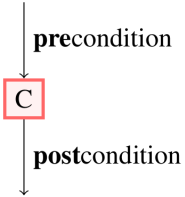
How can we propogate valid assertions through an algorithm?

For each block of pseudocode, we think about if something is true beforehand, then what is true after?

Proving correctness of an algorithm can be hard, so we break it into smaller problems.

How can we propogate valid assertions through an algorithm?

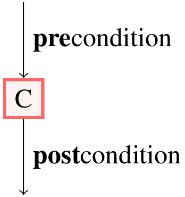
For each block of pseudocode, we think about if something is true beforehand, then what is true after?



Proving correctness of an algorithm can be hard,
so we break it into smaller problems.

How can we propogate valid assertions through an algorithm?

For each block of pseudocode, we think about if something is true beforehand, then what is true after?



We'll say this is valid if, whenever *precondition* is true before we run C, then *postcondition* is true after we run C.

Reasoning about programs: Floyd-Hoare logic

We write:

$$\{Pre\}\ C\ \{Post\}$$

where

- Pre is a mathematical assertion - the precondition
- Post is a mathematical assertion - the postcondition
- C is some program code

Reasoning about programs: Floyd-Hoare logic

We write:

{Pre} C {Post}

where

- Pre is a mathematical assertion - the precondition
- Post is a mathematical assertion - the postcondition
- C is some program code

{Pre} C {Post} is valid if

- whenever the precondition is true, and the code is executed
- if the code terminates, then the postcondition is true

Which are valid?

{Pre} Code {Post}

Pre	Code	Post
$X = Y + 2$	$Y \leftarrow Y + 1$	$X > Y * 2$
$X = Y + 1$	$Y \leftarrow Y + X$	$X > Y * 2$
$X + Y > k$	$X \leftarrow X + k$ $Y \leftarrow Y - 1$	$Y > k$
$X > Y$	$X \leftarrow X + 1$ $Y \leftarrow Y - 1$	$X - Y > 0$

Rules for applying valid assertions

1. Elementary logic
   - $\rightsquigarrow$ for manipulating/rearranging the assertions
2. Sequential rule
   - $\rightsquigarrow$ for propagating valid assertions through simple actions
3. Loop Invariant theorems
   - $\rightsquigarrow$ for propagating valid assertions through while and for-loops

Rules for applying valid assertions

1. Elementary logic
   - $\rightsquigarrow$ for manipulating/rearranging the assertions
2. Sequential rule
   - $\rightsquigarrow$ for propagating valid assertions through simple actions
3. Loop Invariant theorems
   - $\rightsquigarrow$ for propagating valid assertions through while and for-loops

All justified using Floyd-Hoare logic

Using logic

Suppose Prop logically implies Newprop:

Then any occurrence of $\boxed{\text{Prop}}$ can be $\boxed{\text{NewProp}}$

Suppose Prop logically implies Newprop:

Then any occurrence of $\boxed{\text{Prop}}$ can be $\boxed{\text{NewProp}}$

Example

while $\mathrm{X} \neq k$ **do**

$\Longleftarrow \boxed{y = x * k \quad \text{a loop invariant}}$

$\mathrm{Y} \leftarrow \mathrm{Y} + k$

$\mathrm{X} \leftarrow \mathrm{X} + 1$

$\Longleftarrow \boxed{y = x * k \text{ and } x = k}$

return Y

Suppose Prop logically implies Newprop:

Then any occurrence of $\boxed{\text{Prop}}$ can be $\boxed{\text{NewProp}}$

Example

while $\mathrm{X} \neq k$ **do**

$\Longleftarrow \boxed{y = x * k \quad \text{a loop invariant}}$

$\mathrm{Y} \leftarrow \mathrm{Y} + k$

$\mathrm{X} \leftarrow \mathrm{X} + 1$

$\Longleftarrow \boxed{y = x * k \text{ and } x = k}$

return Y

$\Longleftarrow \boxed{k * k \text{ returned}}$

Suppose Prop logically implies Newprop:

Then any occurrence of $\boxed{\text{Prop}}$ can be $\boxed{\text{NewProp}}$

Example

while $\mathrm{X} \neq k$ **do**

$\Longleftarrow \boxed{y = x * k \qquad \text{a loop invariant}}$

$\mathrm{Y} \leftarrow \mathrm{Y} + k$

$\mathrm{X} \leftarrow \mathrm{X} + 1$

$\Longleftarrow \boxed{y = x * k \text{ and } x = k}$

return Y

$\Longleftarrow \boxed{k * k \text{ returned}}$

Because $y = x * k$ and $x = k$ logically implies $y = k * k$

Given a valid triple, how do we propogate assertions through our algorithm?

{Pre} C {Post} is valid if

- whenever the precondition is true, and the code is executed
- if the code terminates, then the postcondition is true

Sequential rule

Suppose {Pre} Code {Post} is valid

Then

…

$\Longleftarrow \boxed{\text{Pre}}$

Code

…

Sequential rule

Suppose {Pre} Code {Post} is valid

Then

. . .

$\Longleftarrow$ $\boxed{\text{Pre}}$

Code

. . .

can be extended to:

. . .

$\Longleftarrow$ $\boxed{\text{Pre}}$

Code

$\Longleftarrow$ $\boxed{\text{Post}}$

. . .

Application of sequential rule

$\ldots$

$\Longleftarrow \boxed{\textsc{sum} = 0}$

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow \text{SUM} + \text{PTR}$

Application of sequential rule

$\ldots$

$\Longleftarrow \boxed{\text{SUM} = 0}$

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow \text{SUM} + \text{PTR}$

can be extended to:

$\Longleftarrow \boxed{\text{SUM} = 0}$

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow \text{SUM} + \text{PTR}$

$\Longleftarrow \boxed{\text{SUM} = \text{PTR}}$

Application of sequential rule

$\ldots$

$\Longleftarrow \boxed{\text{SUM} = 0}$

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow \text{SUM} + \text{PTR}$

can be extended to:

$\Longleftarrow \boxed{\text{SUM} = 0}$

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow \text{SUM} + \text{PTR}$

$\Longleftarrow \boxed{\text{SUM} = \text{PTR}}$

because

$$\{\text{SUM} = 0\} \begin{array}{l} \text{PTR} \leftarrow 1 \\ \text{SUM} \leftarrow \text{SUM} + \text{PTR} \end{array} \{\text{SUM} = \text{PTR}\}$$

is valid

What about **while**-loops?

Idea: use invariants.

{Pre} C {Post} is valid if

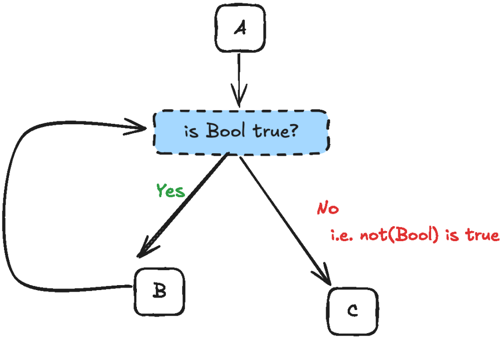
- whenever the precondition is true, and the code is executed
- if the code terminates, then the postcondition is true

Propogating assertions: while-loops

A

while *condition* **do**
| B

C

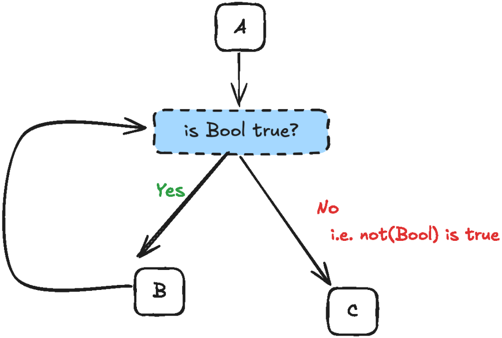


A

while *condition* **do**

| B

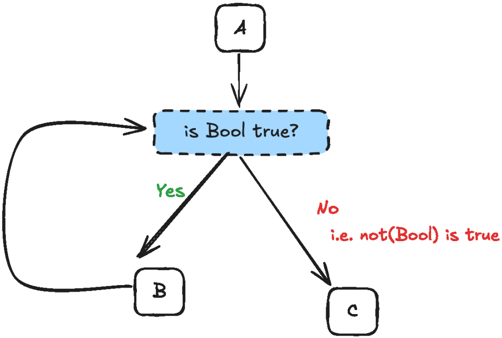
C



An invariant is a property which, if it's true before doing the loop, is also true after doing the loop.

Propogating assertions: while-loops

A
while *condition* **do**
| B
C



An invariant is a property which, if it's true before doing the loop, is also true after doing the loop.

⇝ To actually get into the loop, we need *condition* to be true!

while *Bool* **do** Body

Invariants for while-loops

The mathematical statement *Inv* is a While-invariant if

- {Bool and Inv} Body {Inv} is valid

while *Bool* **do** Body

Invariants for while-loops

The mathematical statement *Inv* is a While-invariant if

- {Bool and Inv} Body {Inv} is valid

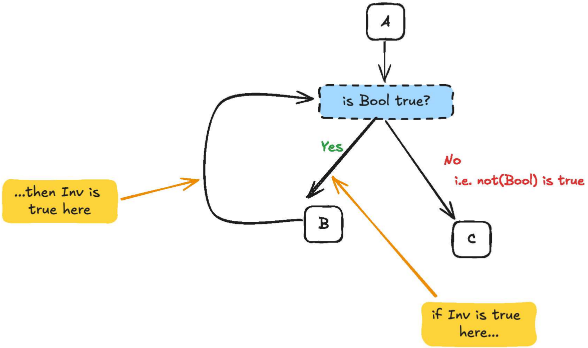
in English: *Inv* is preserved each time the body is executed

while *Bool* **do** Body

Invariants for while-loops

The mathematical statement *Inv* is a While-invariant if

- {Bool and Inv} Body {Inv} is valid



Invariants

while *Bool* **do** Body

Invariants for while-loops

The mathematical statement *Inv* is a While-invariant if

- {Bool and Inv} Body {Inv} is valid

in English: *Inv* is preserved each time the body is executed

Which are invariants ?

Bool	Body	Inv
$\mathrm{X} > 0$	$\mathrm{X} \leftarrow \mathrm{X} + 1$ $\mathrm{Y} \leftarrow \mathrm{Y} - 1$	$\mathrm{X} + \mathrm{Y} = k$
$\mathrm{Y} > 0$	$\mathrm{Y} \leftarrow \mathrm{Y} + k$ $\mathrm{X} \leftarrow \mathrm{X} + 1$	$\mathrm{Y} = k * \mathrm{X}$

While loop invariance theorem

while *Bool* **do** Body

Suppose Inv

1. is an invariant
2. is true *before* the While statement starts

Then, whenever the While statement terminates, if ever,
we know that

1. Inv remains true
2. Bool is false (i.e. *not(Bool)* is true)

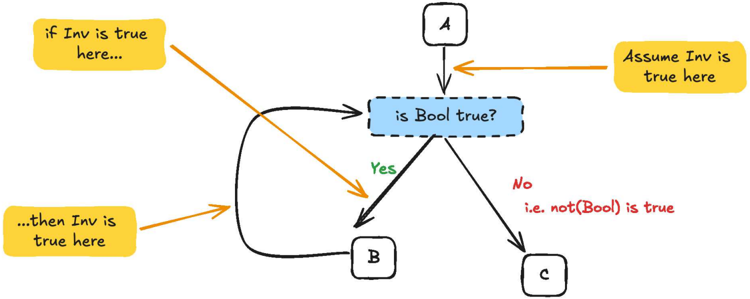
Why is this true? Look at the picture again

Suppose Inv

1. is an invariant
2. is true *before* the While statement starts

Then, whenever the While statement terminates, if ever, we know that

1. Inv remains true
2. Bool is false



Using loop invariance theorem

$\Longleftarrow$ $\boxed{\text{Inv}}$

while *Bool* **do**

  $\Longleftarrow$ $\boxed{\text{Inv}}$

  Body

...

can be extended to

$\Longleftarrow$ $\boxed{\text{Inv}}$

while *Bool* **do**

  $\Longleftarrow$ $\boxed{\text{Inv}}$

  Body

$\Longleftarrow$ $\boxed{\text{Inv and not Bool}}$

Example use

Input: number $k \geq 0$

Output: $k * k$

$\Longleftarrow$ $\boxed{k \geq 0}$

$\mathrm{X} \leftarrow 0$

$\mathrm{Y} \leftarrow 0$

$\Longleftarrow$ $\boxed{y = x * k}$

while $\mathrm{X} \neq k$ **do**

  $\Longleftarrow$ $\boxed{y = x * k \quad \text{a loop invariant}}$

  $\mathrm{Y} \leftarrow \mathrm{Y} + k$

  $\mathrm{X} \leftarrow \mathrm{X} + 1$

return Y

Example use

Input: number $k \geq 0$

Output: $k * k$

$\Longleftarrow$ $\boxed{k \geq 0}$

$\mathrm{X} \leftarrow 0$

$\mathrm{Y} \leftarrow 0$

$\Longleftarrow$ $\boxed{y = x * k}$

while $\mathrm{X} \neq k$ **do**

> $\Longleftarrow$ $\boxed{y = x * k \quad \text{a loop invariant}}$
>
> $\mathrm{Y} \leftarrow \mathrm{Y} + k$
>
> $\mathrm{X} \leftarrow \mathrm{X} + 1$

$\Longleftarrow$ $\boxed{y = x * k \text{ and } x = k}$

return Y

Example use

Input: number $k \geq 0$

Output: $k * k$

$\Longleftarrow \boxed{k \geq 0}$

$\mathrm{X} \leftarrow 0$

$\mathrm{Y} \leftarrow 0$

$\Longleftarrow \boxed{y = x * k}$

while $\mathrm{X} \neq k$ **do**

$\Longleftarrow \boxed{y = x * k \quad \text{a loop invariant}}$

$\mathrm{Y} \leftarrow \mathrm{Y} + k$

$\mathrm{X} \leftarrow \mathrm{X} + 1$

$\Longleftarrow \boxed{y = x * k \text{ and } x = k}$

return Y

because

- $\mathrm{Y} = \mathrm{X} * k$ before loop is entered
- $\{\mathrm{X} \neq k \text{ and } \mathrm{Y} = \mathrm{X} * k\} \begin{array}{l} \mathrm{Y} \leftarrow \mathrm{Y} + k \\ \mathrm{X} \leftarrow \mathrm{X} + 1 \end{array} \{\mathrm{Y} = \mathrm{X} * k\}$ is valid

Examples

Example: summing numbers

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\textsc{ptr} \leftarrow 1$

$\textsc{sum} \leftarrow 1$

while $\textsc{ptr} \neq n$ **do**

  $\textsc{ptr} \leftarrow \textsc{ptr} + 1$

  $\textsc{sum} \leftarrow \textsc{sum} + \textsc{ptr}$

return $\textsc{sum}$

Example: summing numbers

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow 1$

while $\text{PTR} \neq n$ **do**

    $\text{PTR} \leftarrow \text{PTR} + 1$

    $\text{SUM} \leftarrow \text{SUM} + \text{PTR}$

return SUM

Goal: establish that the sum of first n positive numbers is returned

Example: summing numbers

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow 1$

while $\text{PTR} \neq n$ **do**

  $\text{PTR} \leftarrow \text{PTR} + 1$

  $\text{SUM} \leftarrow \text{SUM} + \text{PTR}$

return SUM

$\Longleftarrow$ sum of first n positive numbers returned

Example: summing numbers

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow 1$

while $\text{PTR} \neq n$ **do**

$\Longleftarrow$ $\boxed{\text{loop invariant ???}}$

  $\text{PTR} \leftarrow \text{PTR} + 1$

  $\text{SUM} \leftarrow \text{SUM} + \text{PTR}$

return SUM

$\Longleftarrow$ $\boxed{\text{sum of first } n \text{ positive numbers returned}}$

What is the relevant loop assertion?

Input: number $n \geq 1$
Output: sum of first n positive numbers

$\textsc{ptr} \leftarrow 1$
$\textsc{sum} \leftarrow 1$

while $\textsc{ptr} \neq n$ **do**

  $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
  $\textsc{sum} \leftarrow \textsc{sum} + \textsc{ptr}$

return $\textsc{sum}$

Assigning assertions

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\Longleftarrow \boxed{n \geq 1}$

$\textsc{ptr} \leftarrow 1$

$\textsc{sum} \leftarrow 1$

while $\textsc{ptr} \neq n$ **do**

  $\textsc{ptr} \leftarrow \textsc{ptr} + 1$

  $\textsc{sum} \leftarrow \textsc{sum} + \textsc{ptr}$

return $\textsc{sum}$

Assigning assertions

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\Longleftarrow \boxed{n \geq 1}$

$\textsc{ptr} \leftarrow 1$

$\textsc{sum} \leftarrow 1$

$\Longleftarrow \boxed{\textsc{sum} = \textsc{ptr}}$

while $\textsc{ptr} \neq n$ **do**

$\quad\textsc{ptr} \leftarrow \textsc{ptr} + 1$

$\quad\textsc{sum} \leftarrow \textsc{sum} + \textsc{ptr}$

return $\textsc{sum}$

Assigning assertions

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\Longleftarrow \boxed{n \geq 1}$

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow 1$

$\Longleftarrow \boxed{\text{SUM} = \text{PTR}}$

while $\text{PTR} \neq n$ **do**

  $\text{PTR} \leftarrow \text{PTR} + 1$

  $\text{SUM} \leftarrow \text{SUM} + \text{PTR}$

return SUM

because

$$\{n \geq 1\} \begin{array}{l} \text{PTR} \leftarrow 1 \\ \text{SUM} \leftarrow 1 \end{array} \{\text{SUM} = \text{PTR}\}$$

is valid

Assigning assertions

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\Longleftarrow \boxed{n \geq 1}$

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow 1$

$\Longleftarrow \boxed{\text{SUM} = \text{PTR}}$

while $\text{PTR} \neq n$ **do**

$\quad \Longleftarrow \boxed{\text{SUM} = 1 + \cdots + \text{PTR}}$

$\quad \text{PTR} \leftarrow \text{PTR} + 1$

$\quad \text{SUM} \leftarrow \text{SUM} + \text{PTR}$

return SUM

Assigning assertions

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\Longleftarrow \boxed{n \geq 1}$

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow 1$

$\Longleftarrow \boxed{\text{SUM} = \text{PTR}}$

while $\text{PTR} \neq n$ **do**

$\quad \Longleftarrow \boxed{\text{SUM} = 1 + \cdots + \text{PTR}}$

$\quad \text{PTR} \leftarrow \text{PTR} + 1$

$\quad \text{SUM} \leftarrow \text{SUM} + \text{PTR}$

return SUM

because

$$\{\text{PTR} \neq n \textit{ and Inv}\} \begin{array}{ll} \text{PTR} & \leftarrow \\ \text{PTR} + 1 & \\ \text{SUM} & \leftarrow \\ \text{SUM} + \text{PTR} & \end{array} \{\textit{Inv}\}$$

is valid

Inv: $\text{SUM} = 1 + \cdots + \text{PTR}$

Assigning assertions

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\Longleftarrow \boxed{n \geq 1}$

$\textsc{ptr} \leftarrow 1$

$\textsc{sum} \leftarrow 1$

$\Longleftarrow \boxed{\textsc{sum} = \textsc{ptr}}$

while $\textsc{ptr} \neq n$ **do**

  $\Longleftarrow \boxed{\textsc{sum} = 1 + \cdots + \textsc{ptr}}$

  $\textsc{ptr} \leftarrow \textsc{ptr} + 1$

  $\textsc{sum} \leftarrow \textsc{sum} + \textsc{ptr}$

$\Longleftarrow \boxed{\textsc{sum} = 1 + \cdots + \textsc{ptr} \text{ and } \textsc{ptr} = n}$

return $\textsc{sum}$

Inv: $\textsc{sum} = 1 + \cdots + \textsc{ptr}$

Assigning assertions

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\Longleftarrow \boxed{n \geq 1}$

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow 1$

$\Longleftarrow \boxed{\text{SUM} = \text{PTR}}$

while $\text{PTR} \neq n$ **do**

  $\Longleftarrow \boxed{\text{SUM} = 1 + \cdots + \text{PTR}}$

  $\text{PTR} \leftarrow \text{PTR} + 1$

  $\text{SUM} \leftarrow \text{SUM} + \text{PTR}$

$\Longleftarrow \boxed{\text{SUM} = 1 + \cdots + \text{PTR} \text{ and } \text{PTR} = n}$

return SUM

because of Loop Invariance Theorem:

- Inv is before the loop is entered
- Inv is a loop invariant

Inv: $\text{SUM} = 1 + \cdots + \text{PTR}$

Assigning assertions

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\Longleftarrow \boxed{n \geq 1}$

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow 1$

$\Longleftarrow \boxed{\text{SUM} = \text{PTR}}$

while $\text{PTR} \neq n$ **do**

$\quad \Longleftarrow \boxed{\text{SUM} = 1 + \cdots + \text{PTR}}$

$\quad \text{PTR} \leftarrow \text{PTR} + 1$

$\quad \text{SUM} \leftarrow \text{SUM} + \text{PTR}$

$\Longleftarrow \boxed{\text{SUM} = 1 + \cdots + \text{PTR} \text{ and } \text{PTR} = n}$

return SUM

$\Longleftarrow \boxed{\text{sum of first } n \text{ positive numbers returned}}$

Inv: $\text{SUM} = 1 + \cdots + \text{PTR}$

Assigning assertions

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\Longleftarrow$ $\boxed{n \geq 1}$

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow 1$

$\Longleftarrow$ $\boxed{\text{SUM} = \text{PTR}}$

while $\text{PTR} \neq n$ **do**

$\Longleftarrow$ $\boxed{\text{SUM} = 1 + \cdots + \text{PTR}}$

$\text{PTR} \leftarrow \text{PTR} + 1$

$\text{SUM} \leftarrow \text{SUM} + \text{PTR}$

$\Longleftarrow$ $\boxed{\text{SUM} = 1 + \cdots + \text{PTR} \text{ and } \text{PTR} = n}$

return SUM

$\Longleftarrow$ $\boxed{\text{sum of first } n \text{ positive numbers returned}}$

because $\text{SUM} = 1 + \cdots + \text{PTR}$ and $\text{PTR} = n$
logically implies $\text{SUM} = 1 + \cdots + n$

Inv: $\text{SUM} = 1 + \cdots + \text{PTR}$

The annotated algorithm

Input: number $n \geq 1$

Output: sum of first n positive numbers

// **assert:** $n \geq 0$

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow 1$

// **assert:** $\text{SUM} = \text{PTR}$

while $\text{PTR} \neq n$ **do**

   // **assert:** $\text{SUM} = 1 + \cdots + \text{PTR}$

   $\text{PTR} \leftarrow \text{PTR} + 1$

   $\text{SUM} \leftarrow \text{SUM} + \text{PTR}$

// **assert:** $\text{SUM} = 1 + \cdots + \text{PTR}$ and $\text{PTR} = n$

return SUM

// **assert:** sum of first n positive numbers returned

Problem:

Use of dots in $\textsc{sum} = 1 + \cdots + \textsc{ptr}$ too imprecise

Mathematical interlude

Problem:

Use of dots in $\textsc{sum} = 1 + \cdots + \textsc{ptr}$ too imprecise

Solution:

Use mathematical notation:

> $\sum_{k=m}^{n} P(k)$ is precise mathematical notation
> for the sum $P(m) + \cdots + P(n)$

Here k is a *dummy variable*, part of the mechanics

Mathematical interlude

Problem:

Use of dots in $\textsc{sum} = 1 + \cdots + \textsc{ptr}$ too imprecise

Solution:

Use mathematical notation:

> $\sum_{k=m}^{n} P(k)$ is precise mathematical notation
> for the sum $P(m) + \cdots + P(n)$

Here k is a *dummy variable*, part of the mechanics

Variations:

$$\sum_{m \leq k \leq n} P(k) \text{ means } P(m) + \cdots + P(n)$$

Summing integers with maths notation

Input: number $n \geq 1$

Output: sum of first n positive numbers

```
// assert: n ≥ 0
PTR ← 1
SUM ← 1
// assert: SUM = PTR
while PTR ≠ n do
    // assert: SUM = ∑_{k=1}^{PTR} k
    PTR ← PTR + 1
    SUM ← SUM + PTR

// assert: SUM = ∑_{k=1}^{PTR} k and PTR = n
return SUM
// assert: sum of first n positive numbers
   returned
```

Summing integers: a variation

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\textsc{ptr} \leftarrow 1$

$\textsc{sum} \leftarrow 1$

while $\textsc{ptr} < n$ **do**

  $\textsc{ptr} \leftarrow \textsc{ptr} + 1$

  $\textsc{sum} \leftarrow \textsc{sum} + \textsc{ptr}$

return $\textsc{sum}$

Summing integers: a variation

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\Longleftarrow \boxed{n \geq 1}$

$\textsc{ptr} \leftarrow 1$

$\textsc{sum} \leftarrow 1$

$\Longleftarrow \boxed{\textsc{sum} = \textsc{ptr}}$

while $\textsc{ptr} < n$ **do**

- $\Longleftarrow \boxed{\textsc{sum} = 1 + \cdots + \textsc{ptr}}$
- $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
- $\textsc{sum} \leftarrow \textsc{sum} + \textsc{ptr}$

return $\textsc{sum}$

Summing integers: a variation

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\Longleftarrow \boxed{n \geq 1}$

$\textsc{ptr} \leftarrow 1$

$\textsc{sum} \leftarrow 1$

$\Longleftarrow \boxed{\textsc{sum} = \textsc{ptr}}$

while $\textsc{ptr} < n$ **do**

$\quad \Longleftarrow \boxed{\textsc{sum} = 1 + \cdots + \textsc{ptr}}$

$\quad \textsc{ptr} \leftarrow \textsc{ptr} + 1$

$\quad \textsc{sum} \leftarrow \textsc{sum} + \textsc{ptr}$

$\Longleftarrow \boxed{\textsc{sum} = 1 + \cdots + \textsc{ptr} \text{ and } \textsc{ptr} \not< n}$

return $\textsc{sum}$

Summing integers: a variation

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\Longleftarrow \boxed{n \geq 1}$

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow 1$

$\Longleftarrow \boxed{\text{SUM} = \text{PTR}}$

while $\text{PTR} < n$ **do**

  $\Longleftarrow \boxed{\text{SUM} = 1 + \cdots + \text{PTR}}$

  $\text{PTR} \leftarrow \text{PTR} + 1$

  $\text{SUM} \leftarrow \text{SUM} + \text{PTR}$

$\Longleftarrow \boxed{\text{SUM} = 1 + \cdots + \text{PTR} \text{ and } \text{PTR} \not< n}$

return SUM

Problem: $\text{SUM} = 1 + \cdots + \text{PTR}$ and $\text{PTR} \not< n$
does not imply $\text{SUM} = 1 + \cdots + n$

Strengthening the loop invariant

New loop invariant:

$\textsc{sum} = 1 + \cdots + \textsc{ptr}$ and $\textsc{ptr} \leq n$

New loop invariant:

$\textsc{sum} = 1 + \cdots + \textsc{ptr}$ and $\textsc{ptr} \leq n$

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\textsc{ptr} \leftarrow 1$

$\textsc{sum} \leftarrow 1$

while $\textsc{ptr} < n$ **do**

  $\textsc{ptr} \leftarrow \textsc{ptr} + 1$

  $\textsc{sum} \leftarrow \textsc{sum} + \textsc{ptr}$

return $\textsc{sum}$

Strengthening the loop invariant

New loop invariant:

$\textsc{sum} = 1 + \cdots + \textsc{ptr}$ and $\textsc{ptr} \leq n$

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\Longleftarrow$ $\boxed{n \geq 1}$

$\textsc{ptr} \leftarrow 1$

$\textsc{sum} \leftarrow 1$

$\Longleftarrow$ $\boxed{\textsc{sum} = \textsc{ptr} \text{ and } \textsc{ptr} \leq n}$

while $\textsc{ptr} < n$ **do**

$\Longleftarrow$ $\boxed{\textsc{sum} = 1 + \cdots + \textsc{ptr} \text{ and } \textsc{ptr} \leq n}$

$\textsc{ptr} \leftarrow \textsc{ptr} + 1$

$\textsc{sum} \leftarrow \textsc{sum} + \textsc{ptr}$

$\Longleftarrow$ $\boxed{\textsc{sum} = 1 + \cdots + \textsc{ptr} \text{ and } \textsc{ptr} \leq n \text{ and } \textsc{ptr} \not< n}$

return $\textsc{sum}$

Strengthening the loop invariant

New loop invariant:

$\textsc{sum} = 1 + \cdots + \textsc{ptr}$ and $\textsc{ptr} \leq n$

Input: number $n \geq 1$

Output: sum of first n positive numbers

$\Longleftarrow$ $\boxed{n \geq 1}$

$\textsc{ptr} \leftarrow 1$

$\textsc{sum} \leftarrow 1$

$\Longleftarrow$ $\boxed{\textsc{sum} = \textsc{ptr} \text{ and } \textsc{ptr} \leq n}$

while $\textsc{ptr} < n$ **do**

$\Longleftarrow$ $\boxed{\textsc{sum} = 1 + \cdots + \textsc{ptr} \text{ and } \textsc{ptr} \leq n}$

$\textsc{ptr} \leftarrow \textsc{ptr} + 1$

$\textsc{sum} \leftarrow \textsc{sum} + \textsc{ptr}$

$\Longleftarrow$ $\boxed{\textsc{sum} = 1 + \cdots + \textsc{ptr} \text{ and } \textsc{ptr} \leq n \text{ and } \textsc{ptr} \not< n}$

return $\textsc{sum}$

$\Longleftarrow$ $\boxed{\text{sum of first } n \text{ numbers returned}}$

The annotated variation

Input: number $n \geq 1$

Output: sum of first n positive numbers

```
// assert: n ≥ 0
```

$\text{PTR} \leftarrow 1$

$\text{SUM} \leftarrow 1$

// **assert:** $\text{SUM} = \text{PTR}$

while $\text{PTR} < n$ **do**

  // **assert:** $\text{SUM} = \sum_{k=1}^{k=\text{PTR}} k$ and $\text{PTR} \leq n$

  $\text{PTR} \leftarrow \text{PTR} + 1$

  $\text{SUM} \leftarrow \text{SUM} + \text{PTR}$

// **assert:** $\text{SUM} = \sum_{k=1}^{k=\text{PTR}} k$ and $\text{PTR} \leq n$ and $\text{PTR} \not< n$

return SUM

// sum of first n positive numbers returned

Input: number $n \geq 1$

Output: sum of first n positive numbers

`// assert:` $n \geq 0$

$\textsc{ptr} \leftarrow 1$

$\textsc{sum} \leftarrow 1$

`// assert:` $\textsc{sum} = \textsc{ptr}$

while $\textsc{ptr} \leq n$ **do**

> `// assert:` $\textsc{sum} = \sum_{k=1}^{k=\textsc{ptr}} k$ `and` $\textsc{ptr} \leq n+1$
>
> $\textsc{ptr} \leftarrow \textsc{ptr} + 1$
>
> $\textsc{sum} \leftarrow \textsc{sum} + \textsc{ptr}$

`// assert:` $\textsc{sum} = \sum_{k=1}^{k=\textsc{ptr}} k$ `and` $\textsc{ptr} \leq n+1$ $\textsc{ptr} \not\leq n$

return $\textsc{sum}$

`// sum of first` n `positive numbers returned`

What we've seen so far

Ways of propogating assertions using valid Floyd-Hoare triples:

1. Basic logic (we'll come back to this at the end of the module)
2. Use the sequential rule
3. Use the **while**-loop invariant theorem

What we've seen so far

Ways of propogating assertions using valid Floyd-Hoare triples:

1. Basic logic (we'll come back to this at the end of the module)
2. Use the sequential rule
3. Use the **while**-loop invariant theorem

What about **for**-loops?

What do we want?

The theorem for **while**:

$\Longleftarrow \boxed{\text{Inv}}$

while *Bool* **do**

> $\Longleftarrow \boxed{\text{Inv}}$
>
> Body

. . .

can be extended to

$\Longleftarrow \boxed{\text{Inv}}$

while *Bool* **do**

> $\Longleftarrow \boxed{\text{Inv}}$
>
> Body

$\Longleftarrow \boxed{\text{Inv and not Bool}}$

What do we want?

The theorem for **while**:

$\Longleftarrow$ Inv

while *Bool* **do**

| $\Longleftarrow$ Inv
| Body

…

can be extended to

$\Longleftarrow$ Inv

while *Bool* **do**

| $\Longleftarrow$ Inv
| Body

$\Longleftarrow$ Inv and not Bool

What we want:

$\Longleftarrow$ Inv(0)

for $i \leftarrow 1$ **to** n **do**

| ???

…

can be extended to:

$\Longleftarrow$ Inv(0)

for $i \leftarrow 1$ **to** n **do**

| ???

$\Longleftarrow$ Inv(n)

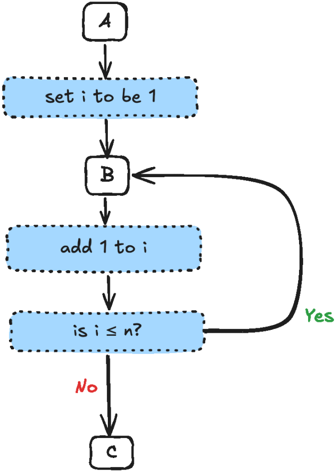
For-loop invariance theorem, in pictures

A

for $i \leftarrow 1$ **to** n **do**

  B

C



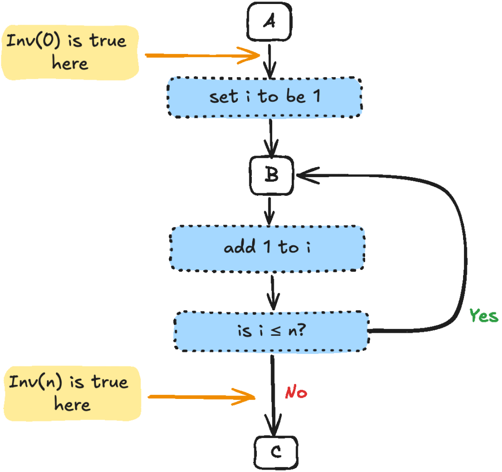
A

for $i \leftarrow 1$ **to** n **do**

  B

C

What we want



For-loop invariance theorem, in pictures

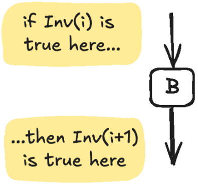
A

for $i \leftarrow 1$ **to** n **do**

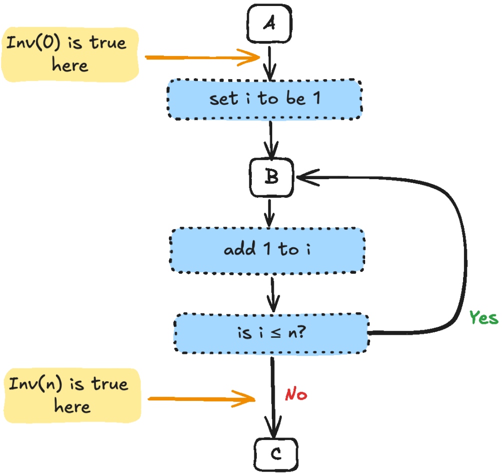
  B

C

How we get it



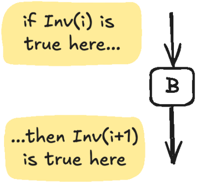
What we want



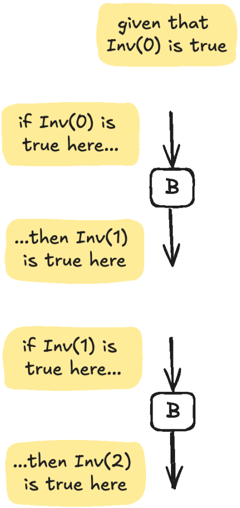
For-loop invariance theorem, in pictures

A
for $i \leftarrow 1$ **to** n **do**
  B
C

How we get it



Why this works



For-loop invariance theorem

for $i \leftarrow 1$ **to** n **do** Body

Suppose Inv(i) is a mathematical statement about i. If:

1. Inv(0) is true before the For-statement starts; and
2. {Inv(i)} Body {Inv($i + 1$)}

   In English: Inv(i) is preserved by the Body

Then:

when the for-loop terminates the property Inv(n) is true.

for $i \leftarrow 1$ **to** n **do** Body

Example Inv(i): $\textsc{sum} = \sum_{k=1}^{i} A[k]$

Suppose Inv(i) is a mathematical statement about i. If:

1. Inv(0) is true before the For-statement starts; and
2. {Inv(i)} Body {Inv($i+1$)}

   In English: Inv(i) is preserved by the Body

Then:

when the for-loop terminates the property Inv(n) is true.

Example placement of valid assertions

Input: Array $A[1 \dots n]$
Output: sum of elements in A

$\textsc{sum} \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  $\textsc{sum} \leftarrow \textsc{sum} + A[i]$
return $\textsc{sum}$

Example placement of valid assertions

Input: Array $A[1 \dots n]$

Output: sum of elements in A

$\textsc{sum} \leftarrow 0$

for $i \leftarrow 1$ **to** n **do**

  $\textsc{sum} \leftarrow \textsc{sum} + A[i]$

return $\textsc{sum}$

Example placement of valid assertions

Input: Array $A[1 \ldots n]$

Output: sum of elements in A

$\textsc{sum} \leftarrow 0$

$\Longleftarrow \boxed{\textsc{sum} = 0}$

for $i \leftarrow 1$ **to** n **do**

  $\textsc{sum} \leftarrow \textsc{sum} + A[i]$

return $\textsc{sum}$

Example placement of valid assertions

Input: Array $A[1 \ldots n]$

Output: sum of elements in A

$\textsc{sum} \leftarrow 0$

$\Longleftarrow \boxed{\textsc{sum} = 0}$

for $i \leftarrow 1$ **to** n **do**

$\quad \textsc{sum} \leftarrow \textsc{sum} + A[i]$

$\quad \Longleftarrow \boxed{\textsc{sum} = A[1] + \cdots + A[i]}$

return $\textsc{sum}$

Example placement of valid assertions

Input: Array $A[1 \ldots n]$

Output: sum of elements in A

$\textsc{sum} \leftarrow 0$

$\Longleftarrow \boxed{\textsc{sum} = 0}$

for $i \leftarrow 1$ **to** n **do**

$\quad \textsc{sum} \leftarrow \textsc{sum} + A[i]$

$\quad \Longleftarrow \boxed{\textsc{sum} = A[1] + \cdots + A[i]}$

$\Longleftarrow \boxed{\textsc{sum} = A[1] + \cdots + A[n]}$

return $\textsc{sum}$

Example placement of valid assertions

Input: Array $A[1 \dots n]$

Output: sum of elements in A

$\textsc{sum} \leftarrow 0$

$\Longleftarrow$ $\boxed{\textsc{sum} = 0}$

for $i \leftarrow 1$ **to** n **do**

  $\textsc{sum} \leftarrow \textsc{sum} + A[i]$

  $\Longleftarrow$ $\boxed{\textsc{sum} = A[1] + \cdots + A[i]}$

$\Longleftarrow$ $\boxed{\textsc{sum} = A[1] + \cdots + A[n]}$

return $\textsc{sum}$

$\Longleftarrow$ $\boxed{\text{sum of elements in } A \text{ returned}}$

Input: Array $A[1 \ldots n]$
Output: sum of elements in A

$\textsc{sum} \leftarrow 0$
// **assert:** $\textsc{sum} = 0$
for $i \leftarrow 1$ **to** n **do**
    $\textsc{sum} \leftarrow \textsc{sum} + A[i]$
    // **assert:** $\textsc{sum} = A[1] + \cdots + A[i]$
// **assert:** $\textsc{sum} = A[1] + \cdots + A[n]$

return $\textsc{sum}$
// **assert:** sum of elements in A returned

Using for-loop invariance theorem, in general

$\Longleftarrow \boxed{\text{Inv}(0)}$

for $i \leftarrow 1$ **to** n **do**

  Body

  $\Longleftarrow \boxed{\text{Inv}(i)}$

...

can be extended to:

$\Longleftarrow \boxed{\text{Inv}(0)}$

for $i \leftarrow 1$ **to** n **do**

  Body

  $\Longleftarrow \boxed{\text{Inv}(i)}$

$\Longleftarrow \boxed{\text{Inv}(n)}$

Summing up

Never write down a looping construct unless

- you have a convincing argument to show it terminates
- you have established a relevant invariant
- you document your program with this information

Summing up

Never write down a looping construct unless

- you have a convincing argument to show it terminates
- you have established a relevant invariant
- you document your program with this information

- The arguments do not have to be very formal
- But they have to be convincing
- Design algorithms and invariant simultaneously
- A priori justification is better than a posteriori justification

A couple more examples

What does this algorithm do ?

Input: Array $A[1 \ldots n]$
Output: ????

for $i \leftarrow 2$ **to** n **do**
    $\textsc{key} \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j > 0$ *and* $A[j] > \textsc{key}$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    $A[j+1] \leftarrow \textsc{key}$
return A

Difficult to know unless you have designed the program

Easy to know if you have designed the program

Insertion sort:

Input: Array $A[1 \dots n]$
Output: Array A sorted

for $i \leftarrow 2$ **to** n **do**
    `// maintains` $A[1 \dots i]$ `sorted`
    $\text{KEY} \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j > 0$ *and* $A[j] > \text{KEY}$ **do**
        `// inserts` $A[i]$ `correctly into` $A[1 \dots (i-1)]$
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    $A[j+1] \leftarrow \text{KEY}$
return A

Another example

Input: A number $k \geq 0$

Output: 3(k+1)

$\Longleftarrow \boxed{k \geq 0}$

$\mathrm{X} \leftarrow 0$

$\mathrm{Y} \leftarrow 0$

$\Longleftarrow \boxed{\mathrm{Y} = 3\mathrm{X} \text{ and } \mathrm{X} \leq k + 1}$

while $\mathrm{X} \leq k$ **do**

  $\Longleftarrow \boxed{\mathrm{Y} = 3\mathrm{X} \text{ and } \mathrm{X} \leq k + 1}$

  $\mathrm{X} \leftarrow \mathrm{X} + 1$

  $\mathrm{Y} \leftarrow \mathrm{Y} + 3$

$\Longleftarrow \boxed{\mathrm{Y} = 3\mathrm{X} \text{ and } \mathrm{X} \leq k + 1 \text{ and } \mathrm{X} > k}$

return Y

$\Longleftarrow \boxed{\mathrm{Y} = 3(k + 1)}$