Loop Invariants: Part 1

These slides are courtesy of
Paolo Bucci, Bruce Weide, et al.
The Ohio State University's
Department of Computer Science
With some minor modifications by Dr. Holly

Reasoning About Function Calls

- What a function does is described by its ***contract***
  - Precondition: a property that is true *before* the call is made – stated in the *requires* clause
  - Postcondition: a property that is true *after* the call returns – stated in the *ensures* clause

Reasoning About Function Calls

- What a function does is described by its ***contract***
  - Precondition: a property that is true *before* the call is made – stated in the *requires* clause
  - Postcondition: a property that is true *after* the call returns – stated in the *ensures* clause
- It is known as an ***external contract*** because it appears with the operation's header and is used by a client

Reasoning About Loops

- What a **`while`** loop does is described by its ***loop invariant***
  - Invariant: a property that is true *every time* the code reaches a certain point
  - In the case of a loop invariant, it is when program execution reaches the loop test

Reasoning About Loops

- What a **`while`** loop does is described by its ***loop invariant***
  - Invariant: a property that is true *every time* the code reaches a certain point
  - In the case of a loop invariant, when program execution reaches the loop condition test
- This is known as an ***internal contract*** because it appears in an operation's implementation and helps us reason about a loop's behavior

Reasoning About Loops

- What a **`while`** loop does is described by its ***loop invariant***
  - Invariant: a property that is true *every time* the code reaches a certain point in the loop
  - In the case of a loop invariant, when loop execution reaches

Just **`while`** loops?
What about **`for`** loops?

The same reasoning approach can be applied to **`for`** loops, but some modifications are required

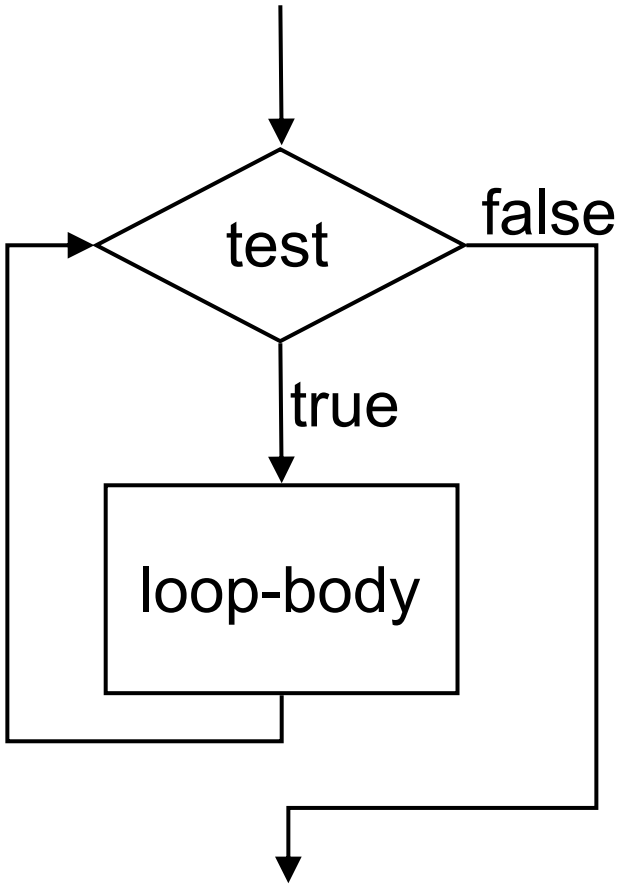
Reasoning About Loops

- What a **`while`** loop does is described by its ***loop invariant***
  - Invariant: a property that is true *every time* the code reaches a certain point in the loop
  - In the case of a loop invariant, when loop execution re

Since a loop invariant is true every time through the loop, it captures what ***does not change***; so it really says what the loop ***does not do***

`while` Statement Control Flow

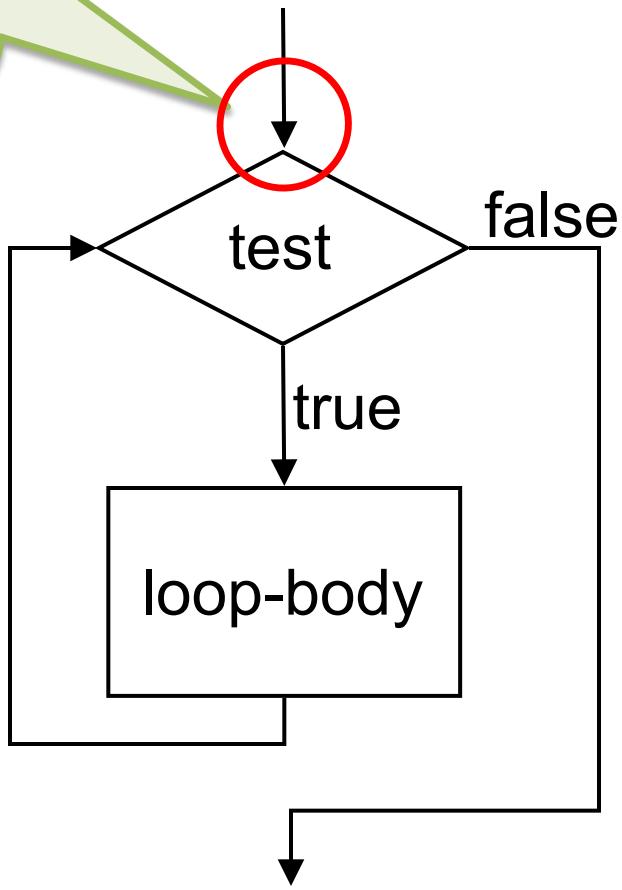
```
while (test) {
  loop-body
}
```



The loop invariant is a property captured in an assertion that is true both here, just before the loop begins...

Control Flow

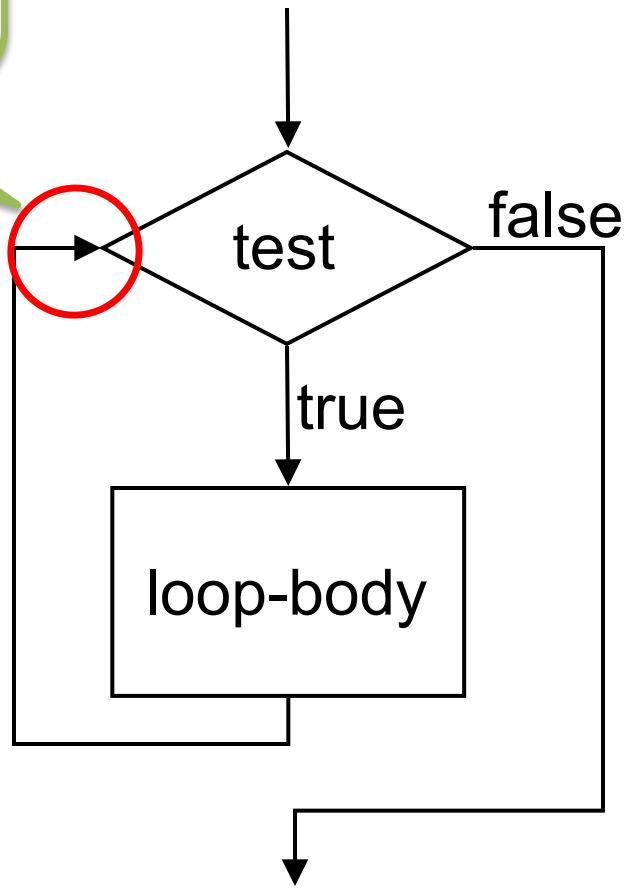
```
while (test) {
  loop-body
}
```



Control Flow

... and here, just after *every* execution of the loop body.

```
while (test) {
  loop-body
}
```



Example

```
void append(QueueOfT& r, QueueOfT& g)
//! Concatenates ("appends") g to the end of r
//! updates: r
//! clears: g
//! ensures: r = #r * #g
```

Example

```
void append(QueueOfT& r, QueueOfT& g)
//! Concatenates ("appends") g to the end of r
//! updates: r
//! clears: g
//! ensures: r = #r * #g
```

Here is a client:

```
1  #include "Wrapper.h"
2  #include "Queue\Queue1.hpp"
3
4  typedef Queue1<Integer> IntegerQueue;
5
6  int main(int argc, char* argv[])
7  {
8     IntegerQueue r1, g1;
9
10    // code that adds items to r1 and g1
11    // ...
12    // ...
13    append(r1, g1);
14 }
```

append's Function Body

```
while (g.length() > 0) {
   T y;
   g.dequeue(y);
   r.enqueue(y);
} // end while
```

append's Function Body

```
while (g.length() > 0) {
   T y;
   g.dequeue(y)
   r.enqueue(y
} // end w
```

What is true about the variables involved in the loop every time we test the loop condition?

Tracing Table	*r = < 1, 2, 3 >* *g = < 4, 5, 6 >*
`while (g.length() > 0) {`	
`  g.dequeue(y);`	
`  r.enqueue(y);`	
`} // end while`	

Tracing Table	*r = < 1, 2, 3 >* *g = < 4, 5, 6 >*
`while (g.length() > 0) {`	
`  g.dequeue(y);`	
`  r.enqueue(y);`	
`} // end while`	

Original incoming values to the loop

Tracing Table	***r = < 1, 2, 3 >*** ***g = < 4, 5, 6 >***
`while (g.length() > 0) {`	
`  g.dequeue(y);`	
`  r.enqueue(y);`	
`} // end while`	

What is true about the variables involved in the loop the *first* time we test the loop condition?

Tracing Table	*r = < 1, 2, 3 >* *g = < 4, 5, 6 >*
`while (g.length() > 0) {`	
	r = < 1, 2, 3 > *g = < 4, 5, 6 >*
`  g.dequeue(y);`	
	r = < 1, 2, 3 > *g = < 5, 6 >* *y = 4*
`  r.enqueue(y);`	
	r = < 1, 2, 3, 4 > *g = < 5, 6 >* *y = 4*
`} // end while`	

Tracing Table	*r = < 1, 2, 3 >* *g = < 4, 5, 6 >*
`while (g.length() > 0) {`	
	r = < 1, 2, 3 > *g = < 4, 5, 6 >*
`g.dequeue(y);`	
	r = < 1, 2, 3 > *g = < 5, 6 >* *y = 4*
`r.enqueue(y);`	
	r = < 1, 2, 3, 4 > *g = < 5, 6 >* *y = 4*
`} // end while`	

Tracing Table	*r = < 1, 2, 3 >* *g = < 4, 5, 6 >*
`while (g.length() > 0) {`	
	r = < 1, 2, 3 > *g = < 4, 5, 6 >*
`g.dequeue(y);`	
	*r = < 1, 2, 3 >* *g = < 5, 6 >* *y = 4*
`r.enqueue(y);`	
	r = < 1, 2, 3, 4 > *g = < 5, 6 >* *y = 4*
`} // end while`	

Tracing Table	*r = < 1, 2, 3 >* *g = < 4, 5, 6 >*
`while (g.length() > 0) {`	
	r = < 1, 2, 3 > *g = < 4, 5, 6 >*
`g.dequeue(y);`	
	r = < 1, 2, 3 > *g = < 5, 6 >* *y = 4*
`r.enqueue(y);`	
	*r = < 1, 2, 3, 4 >* *g = < 5, 6 >* *y = 0*
`} // end while`	

Tracing Table	`r = < 1, 2, 3 >` `g = < 4, 5, 6 >`
`while (g.length() > 0) {`	
	`r = < 1, 2, 3 >`
`g.dequeue(y);`	
	`y = 4`
`r.enqueue(y);`	
	`r = < 1, 2, 3, 4 >` ***`g = < 5, 6 >`*** ***`y = 4`***
`} // end while`	

What is true about *r* and *g* the *first* and *second* time we test the loop condition?

Tracing Table	`r = < 1, 2, 3 >` `g = < 4, 5, 6 >`
`while (g.length() > 0) {`	
`g.dequeue(y);`	
	`y = ?`
`r.enqueue(y);`	

	`r = < 1, 2, 3, 4 >`
	`g = < 5, 6 >`
	`y = 0`

`} // end while`	

Since `y` is declared inside the loop body, the value of `y` is not involved in the loop invariant because *there is no* `y` when we first encounter the loop!

Tracing Table	*r = < 1, 2, 3 >* *g = < 4, 5, 6 >*
while (g.length() > 0) {	
	r = < 1, 2, 3, 4 > *g = < 5, 6 >*
g.dequeue(y);	
	r = < 1, 2, 3, 4 > *g = < 6 >* *y = 5*
r.enqueue(y);	
	r = < 1, 2, 3, 4, 5 > *g = < 6 >* *y = 0*
} // end while	

r and *g* just before evaluating the test for the 3rd time

Tracing Table	`r = < 1, 2, 3 >` `g = < 4, 5, 6 >`
`while (g.length() > 0) {`	
	`r = < 1, 2, 3, 4 >`
`g.dequeue(y);`	
`r.enqueue(y);`	
	`r = < 1, 2, 3, 4, 5 >` ***`g = < 6 >`*** ***`y = 0`***
`} // end while`	

What is true the *first*, *second*, *and third* times we test the loop condition?

Tracing Table	*r = < 1, 2, 3 >* *g = < 4, 5, 6 >*
`while (g.length() > 0) {`	
	r = < 1, 2, 3, 4, 5 > *g = < 6 >*
`g.dequeue(y);`	
	r = < 1, 2, 3, 4, 5 > *g = < >* *y = 6*
`r.enqueue(y);`	
	r = < 1, 2, 3, 4, 5, 6 > *g = < >* *y = 0*
`} // end while`	

r and *g* just before evaluating the test for the 4th time

Tracing Table	`r = < 1, 2, 3 >` `g = < 4, 5, 6 >`
`while (g.length() > 0) {`	
	`r = < 1, 2, 3, 4, 5 >`
`g.dequeue(y);`	
	`y = 0`
`r.enqueue(y);`	
	`r = < 1, 2, 3, 4, 5, 6 >` **`g = < >`** **`y = 0`**
`} // end while`	

What is true the *first*, *second*, *third*, and *fourth* times we test the loop condition?

Tracing Table	`r = < 1, 2, 3 >` `g = < 4, 5, 6 >`
`while (g.length() > 0) {`	
	`r = < 1, 2, 3, 4, 5 >`
`g.dequeue(y);`	
`r.enqueue(y);`	

	`r = < 1, 2, 3, 4,  >`
	`g = < >`
	`y = 0`

`} // end while`	

Whatever is true the last time we test the loop condition is also true here, after the loop finally terminates.

Some Things That Do Not Change

1. “The lengths of the strings are non-negative”, that does not change

   `|r| >= 0 and |g| >= 0` does not change

Some Things That Do Not Change

1. "The lengths of the strings are non-negative", that does not change

   `|r| >= 0 and |g| >= 0` does not change

   – This is true, but it goes without saying because the length of any string is always non-negative

Some Things That Do Not Change

2. "The sum of the lengths of the strings" does not change

`|r| + |g|` does not change

Some Things That Do Not Change

2. "The sum of the lengths of the strings" does not change

   `|r| + |g|` does not change

   – This is true and a useful observation about *append*'s loop that will help us to reach a stronger loop invariant that captures more about what stays the same

Some Things That Do Not Change

3. “The concatenation of the strings” does not change

`r * g` does not change

Some Things That Do Not Change

3. "The concatenation of the strings" does not change

`r * g` does not change

– This is a ***stronger*** more useful observation

– It *implies* the previous observation about the sum of the lengths:

- if `r * g` does not change,
  then `|r| + |g|` also does not change
  but not vice versa

How To Express an Invariant

- How do we say “the concatenation of the strings does not change”?
  - We need to talk values of variables at 2 different times

How To Express an Invariant

- How do we say “the concatenation of the strings does not change”?
  - We need to talk values of variables at 2 different times:
    - The ***original incoming values*** of the variables, their values just prior to the *first* time they are tested (we’ll prepend the variables with #)
    - The ***current values*** of the variables

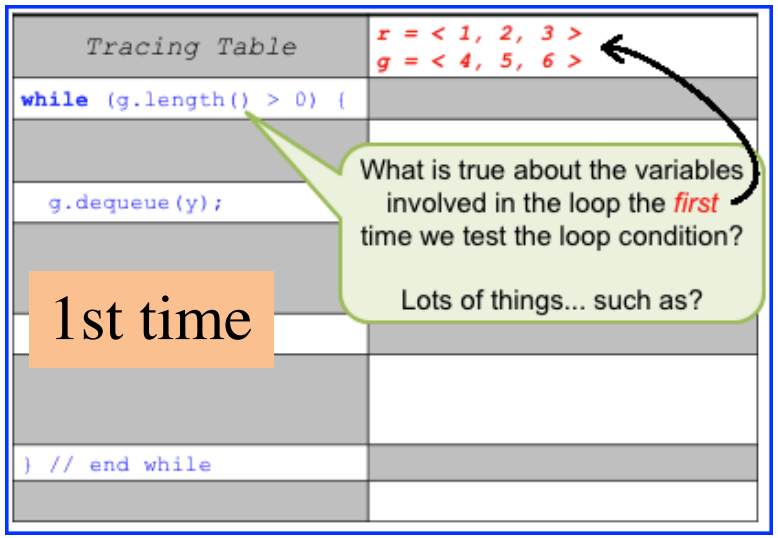
How To Express an Invariant

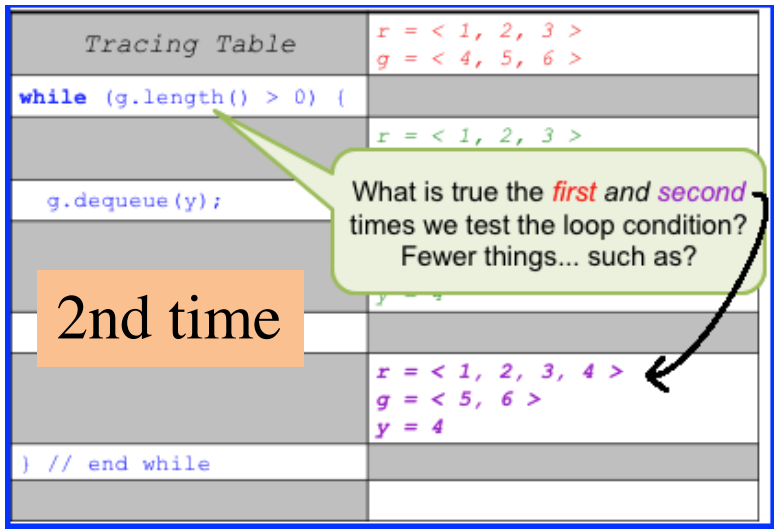
- How do we say "the concatenation of the strings does not change"?
  - We need to talk values of variables at 2 different times:
    - The ***original incoming values*** of the variables, their values just prior to the *first* time they are tested (we'll prepend the variables with `#`)
    - The ***current values*** of the variables

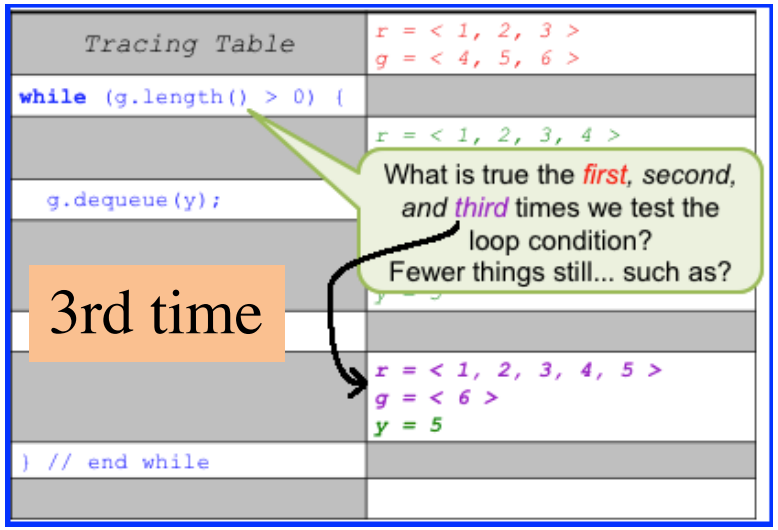
  In this example: `r * g = #r * #g`

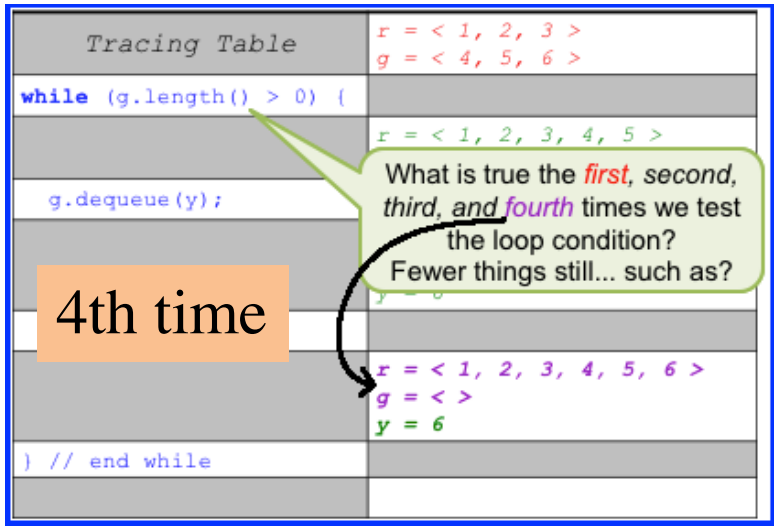
Check using the Tracing Table

`r * g = #r * #g` does not change



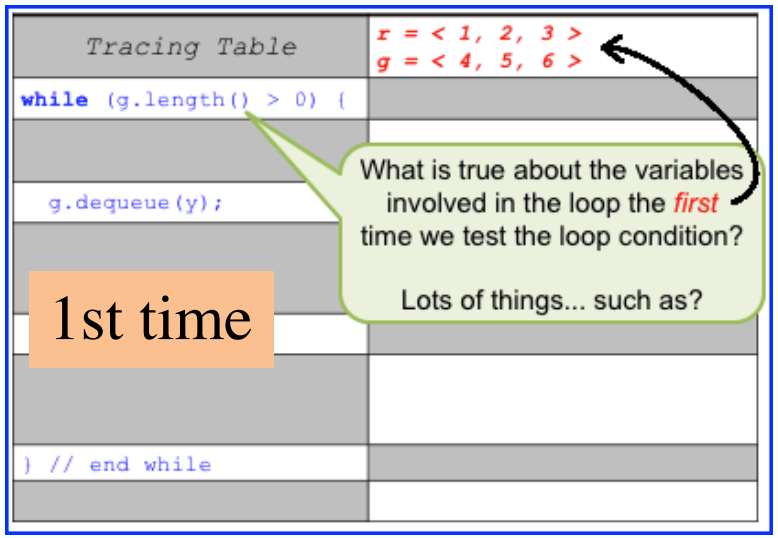






Check the Tracing Table

`r * g = #r * #g` does not change



Here:

`#r = < 1, 2, 3 >`

`#g = < 4, 5, 6 >`

And:

`r = < 1, 2, 3 >`

`g = < 4, 5, 6 >`

So:

`r * g = #r * #g`

Check the Tracing Table

`r * g = #r * #g` does not change

Here:

`#r = < 1, 2, 3 >`

`#g = < 4, 5, 6 >`

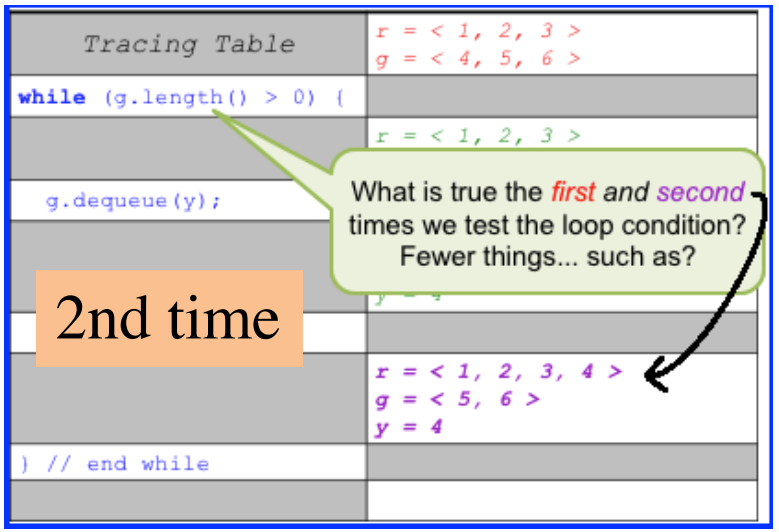
And:

`r = < 1, 2, 3, 4 >`

`g = < 5, 6 >`

So:

`r * g = #r * #g`



Check the Tracing Table

`r * g = #r * #g` does not change

Here:

`#r = < 1, 2, 3 >`

`#g = < 4, 5, 6 >`

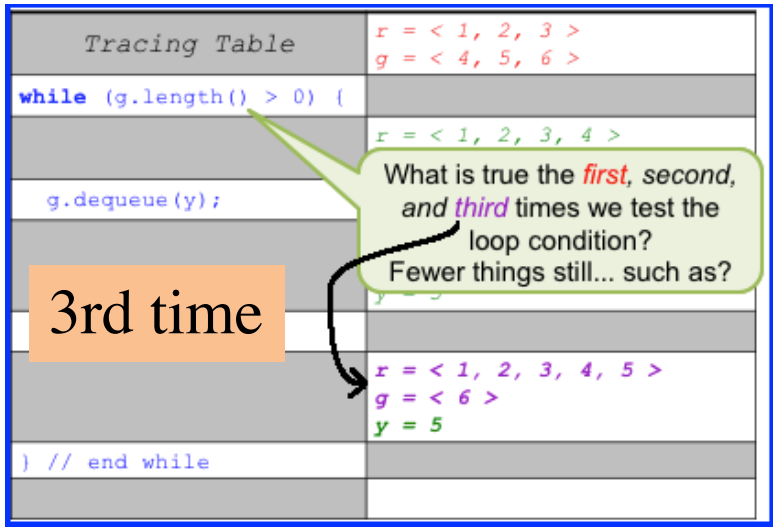
And:

`r = < 1, 2, 3, 4, 5 >`

`g = < 6 >`

So:

`r * g = #r * #g`



Check the Tracing Table

`r * g = #r * #g` does not change

Here:

`#r = < 1, 2, 3 >`

`#g = < 4, 5, 6 >`

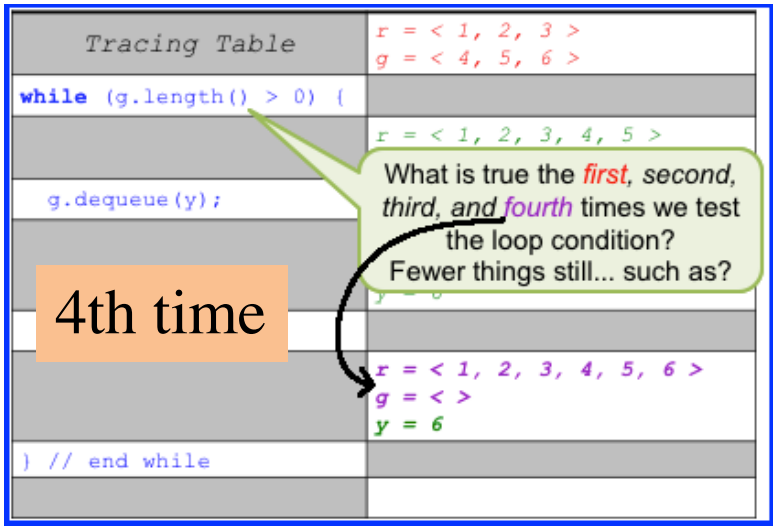
And:

`r = < 1, 2, 3, 4, 5, 6 >`

`g = < >`

So:

`r * g = #r * #g`



append's Function Body

```
while (g.length() > 0) {
    //! updates g, r
    //! maintains r * g = #r * #g
    //! decreases |g|
    T y;
    g.dequeue(y);
    r.enqueue(y);
} // end while
```

append's Function Body

```
while (g.length() > 0) {
    //! updates g, r
    //! maintains |r| * |g| = #r * #g
    //! decreases |g|
    T y;
    g.dequeue(y);
    r.enqueue(y);
} // end while
```

The *updates* keyword introduces the list of variables whose values might change in some iteration

append's Function Body

```
while (g.length() > 0) {
    //! updates g, r
    //! maintains #r * g = #r * #g
    //! decreases |g|
    T y;
    g.dequeue(y);
    r.enqueue(y);
} // end while
```

Any variable in the scope that is not listed as an ***updates-mode*** variable is, by default, a ***restores-mode*** variable, meaning the loop body does not change its value

append's Function Body

```
while (g.length() > 0) {
    //! updates g, r
    //! maintains r * g = #r * #g
    //! decreases |g|
    T y;
    g.dequeue(y);
    r.enqueue(y);
} // end while
```

The *maintains* keyword introduces the claim that the following loop has listed invariant property

append's Function Body

```
while (g.length() > 0) {
    //! updates g, r
    //! maintains r * g = #r * #g
    //! decreases |g|
    T y;
    g.dequeue(y);
    r.enqueue(y);
} // end while
```

The *decreases* keyword introduces the *progress metric*, which describes how the loop makes progress toward the exit condition

Next Up

- See how to use a *loop invariant*
- A little bit about the *progress metric* and loop termination