Basic ACSL Features

Jochen Burghardt

Oct. 21th, 2010

Overview

From unit tests to a function contract

Function to be tested:

```
// returns the index of an occurrence of v in a[]

extern int find(int n,const int a[n],int v);
```

From unit tests to a function contract

Function to be tested:

```
// returns the index of an occurrence of v in a[]

extern int find(int n,const int a[n],int v);
```

Unit test 1:

```
const int const a1[5] = { 9,7,8,9,6 };
int const f1 = find(5,a1,8);
assert(f1 == 2);
```

From unit tests to a function contract

Function to be tested:

```
// returns the index of an occurrence of v in a[]

extern int find(int n,const int a[n],int v);
```

Unit test 2:

```
const int const a1[5] = { 9,7,8,9,6 };
int const f2 = find(5,a1,15);
assert(f2 ==   );
```

From unit tests to a function contract

Function to be tested:

```
// returns the index of an occurrence of v in a[],
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Unit test 2:

```
const int const a1[5] = { 9,7,8,9,6 };
int const f2 = find(5,a1,15);
assert(f2 == -1);
```

(Informal description needs to be refined)

From unit tests to a function contract

Function to be tested:

```
// returns the index of an occurrence of v in a[],
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Unit test 3:

```
const int const a1[5] = { 9,7,8,9,6 };
int const f3 = find(5,a1,9);
assert(f3 == 0);
```

From unit tests to a function contract

Function to be tested:

```
// returns the index of an occurrence of v in a[],
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Unit test 3:

```
const int const a1[5] = { 9,7,8,9,6 };
int const f3 = find(5,a1,9);
assert(f3 == 0 || f3 == 3);
```

(Search order used by `find()` is unknown)

From unit tests to a function contract

Function to be tested:

```
// returns the index of an occurrence of v in a[],
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Unit test assertions:

- `assert(f1 == 2);`
- `assert(f2 == -1);`
- `assert(f3 == 0 || f3 == 3);`

From unit tests to a function contract

Function to be tested:

```
// returns the index of an occurrence of v in a[],
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Unit test assertions:

- ~~`assert(f1 == 2);`~~
- ~~`assert(f2 == -1);`~~
- ~~`assert(f3 == 0 || f3 == 3);`~~

Create a unique `assert` expression working for all tests
(It will be used as ASCL function contract in verification)

Create a unique `assert` expression working for all tests

Create a unique `assert` expression working for all tests

Function to be tested:

```
// returns the index of an occurrence of v in a[]
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Create a unique `assert` expression working for all tests

Function to be tested:

```
// returns the index of an occurrence of v in a[]
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert(a[f] == v);
```

Create a unique `assert` expression working for all tests

Function to be tested:

```
// returns the index of an occurrence of v in a[]
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert(a[f] == v);
```

May cause exception when `find() < 0` or `find() >= n`

Create a unique `assert` expression working for all tests

Function to be tested:

```
// returns the index of an occurrence of v in a[]
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert(0<=f && f<n && a[f] == v);
```

Create a unique `assert` expression working for all tests

Function to be tested:

```
// returns the index of an occurrence of v in a[]
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert(0<=f && f<n && a[f] == v);
```

Does not consider “not-found” case

Create a unique `assert` expression working for all tests

Function to be tested:

```
// returns the index of an occurrence of v in a[]
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert(f == -1 || (0<=f && f<n && a[f] == v));
```

Create a unique `assert` expression working for all tests

Function to be tested:

```
// returns the index of an occurrence of v in a[]
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert(f == -1 || (0<=f && f<n && a[f] == v));
```

Wrong implementation always returning -1 would pass

Create a unique `assert` expression working for all tests

Function to be tested:

```
// returns the index of an occurrence of v in a[]
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert((f == -1 &&  "v not in a" ) || ( 0<=f && f<n && a[f] == v));
```

Create a unique `assert` expression working for all tests

Function to be tested:

```
// returns the index of an occurrence of v in a[]
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert((f == -1 &&  "v not in a" ) || ( 0<=f && f<n && a[f] == v));
```

Create a unique `assert` expression working for all tests

Function to be tested:

```
// returns the index of an occurrence of v in a[]
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert((f == -1 &&  "v not in a" ) || ( 0<=f && f<n && a[f] == v));
```

Cannot be expressed easily in C

Create a unique `assert` expression working for all tests

Function to be tested:

```
// returns the index of an occurrence of v in a[]
// or -1, if not found
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert((f == -1 && "v not in a" ) || ( 0<=f && f<n && a[f] == v));
```

Cannot be expressed easily in C

We therefor now switch to ACSL

Completing a function contract for `find()`: "in"

Completing a function contract for `find()`: "in"

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Completing a function contract for `find()`: "in"

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert((f == -1 && "v not in a") || (0<=f && f<n && a[f] == v));
```

Completing a function contract for `find()`: "in"

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert((f == -1 && "v not in a") || (0<=f && f<n && a[f] == v));
```

- ACSL supports full first-order predicate logic

Completing a function contract for `find()`: "in"

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert((f == -1 && "v not in a") || (0<=f && f<n && a[f] == v));
```

- ACSL supports full first-order predicate logic
- We use its existential quantifier to formalize "v in a"

Completing a function contract for `find()`: "in"

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert((f == -1 && "v not in a") || (0<=f && f<n && a[f] == v));
```

- ACSL supports full first-order predicate logic
- We use its existential quantifier to formalize "v in a"
- **Syntax:** `\exists TYPE VAR; BOOL_EXPR`

Completing a function contract for `find()`: "in"

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert((f == -1 && "v not in a") || (0<=f && f<n && a[f] == v));
```

- ACSL supports full first-order predicate logic
- We use its existential quantifier to formalize "`v` in `a`"
- **Syntax:** `\exists TYPE VAR; BOOL_EXPR`
- **Meaning:** If `VAR` is set to some appropriate value within `TYPE`, then `BOOL_EXPR` becomes true

Completing a function contract for `find()`: "in"

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert((f == -1 && "v not in a") || (0<=f && f<n && a[f] == v));
```

- ACSL supports full first-order predicate logic
- We use its existential quantifier to formalize "`v` in `a`"
- **Syntax:** `\exists TYPE VAR; BOOL_EXPR`
- **Meaning:** If `VAR` is set to some appropriate value within `TYPE`, then `BOOL_EXPR` becomes true
- **Example:** `\exists int i; a[i] == v`

Completing a function contract for `find()`: "in"

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert((f == -1 && "v not in a") || (0<=f && f<n && a[f] == v));
```

- ACSL supports full first-order predicate logic
- We use its existential quantifier to formalize "`v` in `a`"
- **Syntax:** `\exists TYPE VAR; BOOL_EXPR`
- **Meaning:** If `VAR` is set to some appropriate value within `TYPE`, then `BOOL_EXPR` becomes true
- **Example:** `\exists int i; a[i] == v` means "`a[i]` equals `v` for (at least) some `int` value of `i`"

Completing a function contract for `find()`: "in"

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert((f == -1 && ! (\exists int i; a[i] == v))
    || (0<=f && f<n && a[f] == v));
```

- ACSL supports full first-order predicate logic
- We use its existential quantifier to formalize "`v` in `a`"
- **Syntax:** `\exists TYPE VAR; BOOL_EXPR`
- **Meaning:** If `VAR` is set to some appropriate value within `TYPE`, then `BOOL_EXPR` becomes true
- **Example:** `\exists int i; a[i] == v` means "`a[i]` equals `v` for (at least) some `int` value of `i`"
- We may express "`v` not in `a`" by `!(\exists int i; a[i] == v)`

Completing a function contract for `find()`: "in"

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert((f == -1 && ! (\exists int i; a[i] == v))
    || (0<=f && f<n && a[f] == v));
```

- ACSL supports full first-order predicate logic
- We use its existential quantifier to formalize "`v` in `a`"
- **Syntax:** `\exists TYPE VAR; BOOL_EXPR`
- **Meaning:** If `VAR` is set to some appropriate value within `TYPE`, then `BOOL_EXPR` becomes true
- **Example:** `\exists int i; a[i] == v` means "`a[i]` equals `v` for (at least) some `int` value of `i`"
- We may express "`v` not in `a`" by `!(\exists int i; a[i] == v)`
- Since our code is no longer valid C code; we will now fully convert it to proper ACSL

Completing a function contract for `find()`: embedding ACSL syntax

Completing a function contract for `find()`: embedding ACSL syntax

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert(   (f == -1 && ! (\exists int i; a[i] == v))
       || (0<=f && f<n && a[f] == v));
```

Completing a function contract for `find()`: embedding ACSL syntax

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
assert(   (f == -1 && ! (\exists int i; a[i] == v))
       || (0<=f && f<n && a[f] == v));
```

- Any C compiler would complain about the `\exists`

Completing a function contract for `find()`: embedding ACSL syntax

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
/*@ assert(   (f == -1 && ! (\exists int i; a[i] == v))
          || (0<=f && f<n && a[f] == v)); */
```

- Any C compiler would complain about the `\exists`
- Therefor, ACSL is embedded in special comments `/*@...*/` or `//@...`, which are invisible to a C compiler

Completing a function contract for `find()`: embedding ACSL syntax

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
/*@ assert(   (f == -1 && ! (\exists int i; a[i] == v))
          || (0<=f && f<n && a[f] == v)); */
```

- Any C compiler would complain about the `\exists`
- Therefor, ACSL is embedded in special comments `/*@...*/` or `//@...`, which are invisible to a C compiler
- It is no longer neccessary to include a call to `find()` in the code

Completing a function contract for `find()`: embedding ACSL syntax

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
int const f = find(n,a,v);
/*@ assert(   (f == -1 && ! (\exists int i; a[i] == v))
          || (0<=f && f<n && a[f] == v)); */
```

- Any C compiler would complain about the `\exists`
- Therefor, ACSL is embedded in special comments `/*@...*/` or `//@...`, which are invisible to a C compiler
- It is no longer neccessary to include a call to `find()` in the code
- We use the special variable `\result` to refer to the returned value

Completing a function contract for `find()`: embedding ACSL syntax

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
/*@ assert(   (\result == -1 && ! (\exists int i; a[i] == v))
           || (0<=\result && \result<n && a[\result] == v)); */
```

- Any C compiler would complain about the `\exists`
- Therefor, ACSL is embedded in special comments `/*@...*/` or `//@...`, which are invisible to a C compiler
- It is no longer neccessary to include a call to `find()` in the code
- We use the special variable `\result` to refer to the returned value

Completing a function contract for `find()`: embedding ACSL syntax

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

Unique `assert` expression:

```
/*@ assert(   (\result == -1 && ! (\exists int i; a[i] == v))
           || (0<=\result && \result<n && a[\result] == v)); */
```

- Any C compiler would complain about the `\exists`
- Therefor, ACSL is embedded in special comments `/*@...*/` or `//@...`, which are invisible to a C compiler
- It is no longer neccessary to include a call to `find()` in the code
- We use the special variable `\result` to refer to the returned value
- Instead of the `assert()` macro, ASCL provides the `ensures` clause to express the property a procedure has to satisfy

Completing a function contract for `find()`: embedding ACSL syntax

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

ACSL function contract

```
/*@ ensures    (\result == -1 && ! (\exists int i; a[i] == v))
           || (0<=\result && \result<n && a[\result] == v); */
```

- Any C compiler would complain about the `\exists`
- Therefor, ACSL is embedded in special comments `/*@...*/` or `//@...`, which are invisible to a C compiler
- It is no longer neccessary to include a call to `find()` in the code
- We use the special variable `\result` to refer to the returned value
- Instead of the `assert()` macro, ASCL provides the `ensures` clause to express the property a procedure has to satisfy

Completing a function contract for `find()`: embedding ACSL syntax

```
// returns the index of an occurrence of v in a[], or -1
extern int find(int n,const int a[n],int v);
```

ACSL function contract

```
/*@ ensures    (\result == -1 && ! (\exists int i; a[i] == v))
           || (0<=\result && \result<n && a[\result] == v); */
```

- Any C compiler would complain about the `\exists`
- Therefor, ACSL is embedded in special comments `/*@...*/` or `//@...`, which are invisible to a C compiler
- It is no longer neccessary to include a call to `find()` in the code
- We use the special variable `\result` to refer to the returned value
- Instead of the `assert()` macro, ASCL provides the `ensures` clause to express the property a procedure has to satisfy
- An ASCL contract has to appear before a prototype of the function

Completing a function contract for `find()`: embedding ACSL syntax

```
// returns the index of an occurrence of v in a[], or -1
/*@ ensures    (\result == -1 && ! (\exists int i; a[i] == v))
            || (0<=\result && \result<n && a[\result] == v; */
extern int find(int n,const int a[n],int v);
```

- Any C compiler would complain about the `\exists`
- Therefor, ACSL is embedded in special comments `/*@...*/` or `//@...`, which are invisible to a C compiler
- It is no longer neccessary to include a call to `find()` in the code
- We use the special variable `\result` to refer to the returned value
- Instead of the `assert()` macro, ASCL provides the `ensures` clause to express the property a procedure has to satisfy
- An ASCL contract has to appear before a prototype of the function

Completing a function contract for `find()`: embedding ACSL syntax

```
// returns the index of an occurrence of v in a[], or -1
/*@ ensures    (\result == -1 && ! (\exists int i; a[i] == v))
             || (0<=\result<n && a[\result] == v; */
extern int find(int n,const int a[n],int v);
```

- Any C compiler would complain about the `\exists`
- Therefor, ACSL is embedded in special comments `/*@...*/` or `//@...`, which are invisible to a C compiler
- It is no longer neccessary to include a call to `find()` in the code
- We use the special variable `\result` to refer to the returned value
- Instead of the `assert()` macro, ASCL provides the `ensures` clause to express the property a procedure has to satisfy
- An ASCL contract has to appear before a prototype of the function
- ACSL allows relations chains used from mathematical notation: `0<=\result && \result<n` can be shortened to `0<=\result<n`

Completing a function contract for `find()`: embedding ACSL syntax

```
// returns the index of an occurrence of v in a[], or -1
/*@ ensures    (\result == -1 ==> ! (\exists int i; a[i] == v))
           && (0<=\result<n  ==> a[\result] == v)
           && -1<=\result<n; */
extern int find(int n,const int a[n],int v);
```

- Any C compiler would complain about the `\exists`
- Therefor, ACSL is embedded in special comments `/*@...*/` or `//@...`, which are invisible to a C compiler
- It is no longer neccessary to include a call to `find()` in the code
- We use the special variable `\result` to refer to the returned value
- Instead of the `assert()` macro, ASCL provides the `ensures` clause to express the property a procedure has to satisfy
- An ASCL contract has to appear before a prototype of the function
- ACSL allows relations chains used from mathematical notation: `0<=\result && \result<n` can be shortened to `0<=\result<n`
- By some Boolean algebra, we transform the formula into a conjunction

Completing a function contract for `find()`: embedding ACSL syntax

```
// returns the index of an occurrence of v in a[], or -1
/*@ ensures \result==-1 ==> !(\exists int i; a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n; */
extern int find(int n,const int a[n],int v);
```

- Any C compiler would complain about the `\exists`
- Therefor, ACSL is embedded in special comments `/*@...*/` or `//@...`, which are invisible to a C compiler
- It is no longer neccessary to include a call to `find()` in the code
- We use the special variable `\result` to refer to the returned value
- Instead of the `assert()` macro, ASCL provides the `ensures` clause to express the property a procedure has to satisfy
- An ASCL contract has to appear before a prototype of the function
- ACSL allows relations chains used from mathematical notation: `0<=\result && \result<n` can be shortened to `0<=\result<n`
- By some Boolean algebra, we transform the formula into a conjunction
- A conjunctive `ensures` may be split

Completing a function contract for `find()`: embedding ACSL syntax

```
// returns the index of an occurrence of v in a[], or -1
/*@ ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n; */
extern int find(int n,const int a[n],int v);
```

- Any C compiler would complain about the `\exists`
- Therefor, ACSL is embedded in special comments `/*@...*/` or `//@...`, which are invisible to a C compiler
- It is no longer neccessary to include a call to `find()` in the code
- We use the special variable `\result` to refer to the returned value
- Instead of the `assert()` macro, ASCL provides the `ensures` clause to express the property a procedure has to satisfy
- An ASCL contract has to appear before a prototype of the function
- ACSL allows relations chains used from mathematical notation: `0<=\result && \result<n` can be shortened to `0<=\result<n`
- By some Boolean algebra, we transform the formula into a conjunction
- A conjunctive `ensures` may be split
- We need to restrict the range of `a[]` in the first `ensures`

Completing a function contract for `find()`: entry conditions

```
// returns the index of an occurrence of v in a[], or -1
/*@ ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n; */
extern int find(int n,const int a[n],int v);
```

Completing a function contract for `find()`: entry conditions

```
// returns the index of an occurrence of v in a[], or -1
/*@ ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n; */
extern int find(int n,const int a[n],int v);
```

- Most C compilers ignore an upper-bound expression in an array declaration

Completing a function contract for `find()`: entry conditions

```
// returns the index of an occurrence of v in a[], or -1
/*@ ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n; */
extern int find(int n,const int a[n],int v);
```

- Most C compilers ignore an upper-bound expression in an array declaration
- Many C compilers ignore the `const` keyword

Completing a function contract for `find()`: entry conditions

```
// returns the index of an occurrence of v in a[], or -1
/*@ ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n; */
extern int find(int n,const int a[n],int v);
```

- Most C compilers ignore an upper-bound expression in an array declaration
- Many C compilers ignore the `const` keyword
- Frama-C also ignores both

Completing a function contract for `find()`: entry conditions

```
// returns the index of an occurrence of v in a[], or -1
/*@ ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n; */
extern int find(int n,const int a[n],int v);
```

- Most C compilers ignore an upper-bound expression in an array declaration
- Many C compilers ignore the `const` keyword
- Frama-C also ignores both
- So we better tell it the same facts in it's own language

Completing a function contract for `find()`: entry conditions

```
// returns the index of an occurrence of v in a[], or -1
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n; */
extern int find(int n,const int a[n],int v);
```

- Most C compilers ignore an upper-bound expression in an array declaration
- Many C compilers ignore the `const` keyword
- Frama-C also ignores both
- So we better tell it the same facts in it's own language
- `\valid_range(a,0,n-1)` means that `&a[0],...,&a[n-1]` are valid pointers

Completing a function contract for `find()`: entry conditions

```
// returns the index of an occurrence of v in a[], or -1
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n; */
extern int find(int n,const int a[n],int v);
```

- Most C compilers ignore an upper-bound expression in an array declaration
- Many C compilers ignore the `const` keyword
- Frama-C also ignores both
- So we better tell it the same facts in it's own language
- `\valid_range(a,0,n-1)` means that `&a[0],...,&a[n-1]` are valid pointers
- `assigns \nothing` means that `find` does neither modify its parameters nor any global variables

Completing a function contract for `find()`: entry conditions

```
// returns the index of an occurrence of v in a[], or -1
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n; */
extern int find(int n,const int a[n],int v);
```

- Most C compilers ignore an upper-bound expression in an array declaration
- Many C compilers ignore the `const` keyword
- Frama-C also ignores both
- So we better tell it the same facts in it's own language
- `\valid_range(a,0,n-1)` means that `&a[0],...,&a[n-1]` are valid pointers
- `assigns \nothing` means that `find` does neither modify its parameters nor any global variables
- This is stronger than the `const`, but still true and useful in our example

Summary: writing a function contract in ACSL

- A function contract formally describes what a function is expected to do
- It may be considered as a unique `assert` expression usable for any unit test of the function
- To describe function contracts, ACSL provides a more powerful language than C

Summary: writing a function contract in ACSL

- A function contract formally describes what a function is expected to do
- It may be considered as a unique `assert` expression usable for any unit test of the function
- To describe function contracts, ACSL provides a more powerful language than C

Summary: writing a function contract in ACSL

- A function contract formally describes what a function is expected to do
- It may be considered as a unique `assert` expression usable for any unit test of the function
- To describe function contracts, ACSL provides a more powerful language than C

Summary: writing a function contract in ACSL

- A function contract formally describes what a function is expected to do
- It may be considered as a unique `assert` expression usable for any unit test of the function
- To describe function contracts, ACSL provides a more powerful language than C

Function contracts: Frama-C vs. <assert.h>

Generic function to be tested / verified:

```
extern T0 func(T1 x1,...Tk xk);
```

Function contracts: Frama-C vs. <assert.h>

Generic function to be tested / verified:

```
extern T0 func(T1 x1,...Tk xk);
```

Frama-C function contract:

```
/*@ requires req(x1,...,xk);
     ensures ens(\result,x1,...,xk); */
```

Function contracts: Frama-C vs. <assert.h>

Generic function to be tested / verified:

```
extern T0 func(T1 x1,...Tk xk);
```

Frama-C function contract:

```
/*@ requires req(x1,...,xk);
     ensures ens(\result,x1,...,xk); */
```

Corresponding test-code:

```
setup(&x1,...,&xk);     // obtain sample input data
if (req(x1,...,xk)) {
    T0 const x0 = func(x1,...,xk);
    assert(ens(x0,x1,...,xk));
    printf("Test successful\n");
} else {
    printf("Test didn't apply\n");
}
```

Function contracts: Frama-C vs. <assert.h>

Generic function to be tested / verified:

```
extern T0 func(T1 x1,...Tk xk);
```

Frama-C function contract:

```
/*@ requires req(x1,...,xk);
     ensures ens(\result,x1,...,xk); */
```

Corresponding test-code:

```
setup(&x1,...,&xk);     // obtain sample input data
if (req(x1,...,xk)) {
    T0 const x0 = func(x1,...,xk);
    assert(ens(x0,x1,...,xk));
    printf("Test successful\n");
} else {
    printf("Test didn't apply\n");
}
```

Note:

- `requires true` means: `func` runs on all inputs

Function contracts: Frama-C vs. <assert.h>

Generic function to be tested / verified:

```
extern T0 func(T1 x1,...Tk xk);
```

Frama-C function contract:

```
/*@ requires req(x1,...,xk);
     ensures ens(\result,x1,...,xk); */
```

Corresponding test-code:

```
setup(&x1,...,&xk);     // obtain sample input data
if (req(x1,...,xk)) {
    T0 const x0 = func(x1,...,xk);
    assert(ens(x0,x1,...,xk));
    printf("Test successful\n");
} else {
    printf("Test didn't apply\n");
}
```

Note:

- `ensures true` means: nothing to test / verify

Function contracts: Frama-C vs. <assert.h>

Generic function to be tested / verified:

```
extern T0 func(T1 x1,...Tk xk);
```

Frama-C function contract:

```
/*@ requires req(x1,...,xk);
     ensures ens(\result,x1,...,xk); */
```

Corresponding test-code:

```
setup(&x1,...,&xk);     // obtain sample input data
if (req(x1,...,xk)) {
    T0 const x0 = func(x1,...,xk);
    assert(ens(x0,x1,...,xk));
    printf("Test successful\n");
} else {
    printf("Test didn't apply\n");
}
```

Note:

- Some ACSL expressions cannot be translated to C

Function contracts: Frama-C vs. <assert.h>

Generic function to be tested / verified:

```
extern T0 func(T1 x1,...Tk xk);
```

Frama-C function contract:

```
/*@ requires req(x1,...,xk);
     ensures ens(\result,x1,...,xk); */
```

Corresponding test-code:

```
setup(&x1,...,&xk);     // obtain sample input data
if (req(x1,...,xk)) {
    T0 const x0 = func(x1,...,xk);
    assert(ens(x0,x1,...,xk));
    printf("Test successful\n");
} else {
    printf("Test didn't apply\n");
}
```

Note:

- Some ACSL expressions cannot be translated to C , e.g. `\valid_range()`

Function contracts: Frama-C vs. <assert.h>

Generic function to be tested / verified:

```
extern T0 func(T1 x1,...Tk xk);
```

Frama-C function contract:

```
/*@ requires req(x1,...,xk);
     ensures ens(\result,x1,...,xk); */
```

Corresponding test-code:

```
setup(&x1,...,&xk);     // obtain sample input data
if (req(x1,...,xk)) {
    T0 const x0 = func(x1,...,xk);
    assert(ens(x0,x1,...,xk));
    printf("Test successful\n");
} else {
    printf("Test didn't apply\n");
}
```

Note:

- Some ACSL expressions cannot be translated to C , e.g. `\exists`

Function contracts: Frama-C vs. <assert.h>

Generic function to be tested / verified:

```
extern T0 func(T1 x1,...Tk xk);
```

Frama-C function contract:

```
/*@ requires req(x1,...,xk);
     ensures ens(\result,x1,...,xk); */
```

Corresponding test-code:

```
setup(&x1,...,&xk);     // obtain sample input data
if (req(x1,...,xk)) {
    T0 const x0 = func(x1,...,xk);
    assert(ens(x0,x1,...,xk));
    printf("Test successful\n");
} else {
    printf("Test didn't apply\n");
}
```

Note:

- Some ACSL expressions cannot be translated to C , e.g. `\forall`

Programming vs. specification language

Program	**Specification**
Commands to be executed `i = 0;`	Descriptions of a particular state `/*@ i == 0 */`
~~`i == 0;`~~ Warning: Statement without effect	~~`/*@ i = 0 */`~~ Error: Side effect in specification

`double sqrt(double x) {`
`  // Newton's algorithm`
`  // (too complicated`
`  //  to be shown here)`
`}`
Commands to obtain the square-root
	`/*@ requires x >= 0.0;`
	`  ensures  1.0e-12 >=`
	`  fabs(\result*\result-x);`
	`*/`
	
	Test to cross-check the obtained result

Function contract language summary: Junctors

ACSL expression		meaning	
`!p`	negation	as in C:	`(p?false:true)`
`p&&q`	conjunction	as in C:	`(p?q:false)`
`p||q`	disjunction	as in C:	`(p?true:q)`
`p==>q`	implication	`(p?q:true)`	or similarly: `(!p)||q`
`p<==>q`	equivalence	`p==q`	

Function contract language summary: Junctors

ACSL expression		meaning	
`!p`	negation	as in C:	`(p?false:true)`
`p&&q`	conjunction	as in C:	`(p?q:false)`
`p||q`	disjunction	as in C:	`(p?true:q)`
`p==>q`	implication	`(p?q:true)`	or similarly: `(!p)||q`
`p<==>q`	equivalence	`p==q`	

Examples:

```
/*@ensures !(\exists int i; 0<=i<n&&a[i]==v) ==> \result==-1;
   ensures  (\exists int i; 0<=i<n&&a[i]==v) ==> a[\result]==v;
*/
extern int find(int n,const int a[n],int v);
```

Function contract language summary: Quantifiers

Universal quantifier:

```
\forall int x; p(x)
```

is equivalent to an infinite conjunction:

```
...&& p(-2) && p(-1) && p(0) && p(1) && p(2) && p(3) && ...
```

Function contract language summary: Quantifiers

Universal quantifier:

```
\forall int x; p(x)
```

is equivalent to an infinite conjunction:

```
...&& p(-2) && p(-1) && p(0) && p(1) && p(2) && p(3) && ...
```

Existential quantifier:

```
\exists int x; p(x)
```

is equivalent to an infinite disjunction:

```
...|| p(-2) || p(-1) || p(0) || p(1) || p(2) || p(3) || ...
```

Function contract language summary: Quantifiers

Universal quantifier:

```
\forall int x; p(x)
```

is equivalent to an infinite conjunction:

```
...&& p(-2) && p(-1) && p(0) && p(1) && p(2) && p(3) && ...
```

Existential quantifier:

```
\exists int x; p(x)
```

is equivalent to an infinite disjunction:

```
...|| p(-2) || p(-1) || p(0) || p(1) || p(2) || p(3) || ...
```

- Order of quantifiers **does** matter:
  `\forall man_t m;  \exists woman_t w; loves(m,w)` vs.
  `\exists woman_t w; \forall man_t m;  loves(m,w)`

Function contract language summary: Quantifiers

Universal quantifier:

```
\forall int x; p(x)
```

is equivalent to an infinite conjunction:

```
...&& p(-2) && p(-1) && p(0) && p(1) && p(2) && p(3) && ...
```

Existential quantifier:

```
\exists int x; p(x)
```

is equivalent to an infinite disjunction:

```
...|| p(-2) || p(-1) || p(0) || p(1) || p(2) || p(3) || ...
```

- Order of quantifiers **does** matter:
  `\forall man_t m;  \exists woman_t w; loves(m,w)` vs.
  `\exists woman_t w; \forall man_t m;  loves(m,w)`
- Unique existence **cannot** be expressed with `\exists`:
  "p holds at least once": `\exists int i;  p(i)`
  "p holds at most once": `!(\exists int i,j; p(i) && p(j) && i!=j)`
  or equivalently: `\forall int i,j; p(i) && p(j) ==> i==j`

Function contract language summary: Quantifiers

Universal quantifier:

```
\forall int x; p(x)
```

is equivalent to an infinite conjunction:

```
...&& p(-2) && p(-1) && p(0) && p(1) && p(2) && p(3) && ...
```

Existential quantifier:

```
\exists int x; p(x)
```

is equivalent to an infinite disjunction:

```
...|| p(-2) || p(-1) || p(0) || p(1) || p(2) || p(3) || ...
```

- Use ==> to restrict the range of a universal quantifier:
  ```
  \forall int i; 0<=i<8 ==> a[i] != NULL
  ```

Function contract language summary: Quantifiers

Universal quantifier:

```
\forall int x; p(x)
```

is equivalent to an infinite conjunction:

```
...&& p(-2) && p(-1) && p(0) && p(1) && p(2) && p(3) && ...
```

Existential quantifier:

```
\exists int x; p(x)
```

is equivalent to an infinite disjunction:

```
...|| p(-2) || p(-1) || p(0) || p(1) || p(2) || p(3) || ...
```

- Use ==> to restrict the range of a universal quantifier:
  ```
  \forall int i; 0<=i<8 ==> a[i] != NULL
  ```
- Use && to restrict the range of an existential quantifier:
  ```
  \exists int i; 0<=i<8 && a[i] != NULL
  ```

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
extern int find(int n,const int a[n],int v);
```

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- We provide an implementation

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- We provide an implementation

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- We provide an implementation
- We run Frama-C / Jessie on find.c to generate a verification-condition file

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- We provide an implementation
- We run Frama-C / Jessie on find.c to generate a verification-condition file
- We run the Simplify prover on the verification-condition file

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- We provide an implementation
- We run Frama-C / Jessie on find.c to generate a verification-condition file
- We run the Simplify prover on the verification-condition file
- 5 conditions couldn't be verified in our first attempt

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- We run Frama-C / Jessie on find.c to generate a verification-condition file
- We run the Simplify prover on the verification-condition file
- 5 conditions couldn't be verified in our first attempt
- We try to eliminate the reasons, starting with the easiest one ("variant decreases")

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Frama-C tries to prove the termination of each loop

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Frama-C tries to prove the termination of each loop
- For now, we have to provide an expression denoting a measure of "how many iterations remain to be done at most"

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Frama-C tries to prove the termination of each loop
- For now, we have to provide an expression denoting a measure of "how many iterations remain to be done at most"
- More formally, an expression `E` such that always `E>0` and `E` properly decreases with every loop cycle

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- For now, we have to provide an expression denoting a measure of "how many iterations remain to be done at most"
- More formally, an expression `E` such that always `E>0` and `E` properly decreases with every loop cycle
- We choose `n-i`

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- For simple loops like in our example, the choice of `E` might be automated in future

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- For simple loops like in our example, the choice of `E` might be automated in future
- In general, automatically chosing an appropriate `E` is impossible for theoretical reasons [Tur36]

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Running Frama-C / Jessie / Simplify again, we note that 4 unproven conditions remain

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Running Frama-C / Jessie / Simplify again, we note that 4 unproven conditions remain
- We now consider “pointer dereferencing”

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Running Frama-C / Jessie / Simplify again, we note that 4 unproven conditions remain
- We now consider "pointer dereferencing"
- Frama-C couldn't verify that `i>=0` when computing `a[i]`

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Frama-C couldn't verify that `i>=0` when computing `a[i]`
- In fact, a tiny induction proof is needed for this:

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Frama-C couldn't verify that `i>=0` when computing `a[i]`
- In fact, a tiny induction proof is needed for this:
  - `i>=0` holds after 0 loop cycles, i.e. immediately after the assignment `i=0;`

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Frama-C couldn't verify that `i>=0` when computing `a[i]`
- In fact, a tiny induction proof is needed for this:
  - `i>=0` holds after 0 loop cycles, i.e. immediately after the assignment `i=0`;
  - if `i>=0` holds after k loop cycles, then it also holds after $k+1$ cycles, since `i` is increased only

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Since theorem provers are still unable to guess inductive invariants, we have to tell Frama-C explicitely that `i>=0`

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
        loop invariant 0<=i<=n;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Since theorem provers are still unable to guess inductive invariants, we have to tell Frama-C explicitely that `i>=0`
- We use a `loop invariant` clause for this end

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
        loop invariant 0<=i<=n;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Since theorem provers are still unable to guess inductive invariants, we have to tell Frama-C explicitely that `i>=0`
- We use a `loop invariant` clause for this end
- We write "`i<=n`" to catch also the very last cycle

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
        loop invariant 0<=i<=n;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- We run Frama-C / Jessie / Simplify again, and only one unproven condition remains, referring to the first postcondition (`ensures`)

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
        loop invariant 0<=i<=n;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- We have to tell Frama-C the core idea of our algorithm:

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
        loop invariant 0<=i<=n;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- We have to tell Frama-C the core idea of our algorithm:
  "no element of `a[]` inspected so far matched `v`"

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
        loop invariant 0<=i<=n;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- We have to tell Frama-C the core idea of our algorithm:
  "no element of `a[]` inspected so far matched `v`"
  formally: `\forall int j; 0<=j<i ==> a[j] != v;`

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
        loop invariant 0<=i<=n;
        loop invariant \forall int j; 0<=j<i ==> a[j]!=v;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- We use another `loop invariant` clause for this

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
        loop invariant 0<=i<=n;
        loop invariant \forall int j; 0<=j<i ==> a[j]!=v;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- We run Frama-C / Jessie / Simplify again, and now everything could be proved

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
        loop invariant 0<=i<=n;
        loop invariant \forall int j; 0<=j<i ==> a[j]!=v;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Our implementation of `find` has been verified

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
        loop invariant 0<=i<=n;
        loop invariant \forall int j; 0<=j<i ==> a[j]!=v;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Our implementation of `find` has been verified
- Every unit test of `find` on arguments satisfying the `requires` clause is guaranteed to pass an `assert` macro corresponding to the `ensures` clause

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
        loop invariant 0<=i<=n;
        loop invariant \forall int j; 0<=j<i ==> a[j]!=v;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Why not "`0<=j<=i`" in the loop invariant?

Verifying a function implementation against a contract

```
/*@ requires 0 <= n && \valid_range(a,0,n-1);
    assigns \nothing;
    ensures \result==-1 ==> !(\exists int i; 0<=i<n && a[i]==v);
    ensures 0<=\result<n ==> a[\result] == v;
    ensures -1<=\result<n;
 */
int find(int n,const int a[n],int v)
{   int i;
    /*@ loop   variant n - i;
        loop invariant 0<=i<=n;
        loop invariant \forall int j; 0<=j<i ==> a[j]!=v;
    */
    for (i=0; i<n; ++i)
        if (a[i] == v)
            return i;
    return -1;
}
```

- Why not "`0<=j<=i`" in the loop invariant?
- This depends on where the loop invariants holds: before or after the incrementing `++i`

Where does a loop invariant hold?

```
// statements before loop
//@ loop invariant
for (  i=0;   i<n;   ++i  ) {
   // body statements
}
// statements after loop
```

- Control flow of a `for` loop

Where does a loop invariant hold?

```
// statements before loop
//@ loop invariant
for (  i=0;   i<n;   ++i  ) {
   // body statements
}
// statements after loop
```

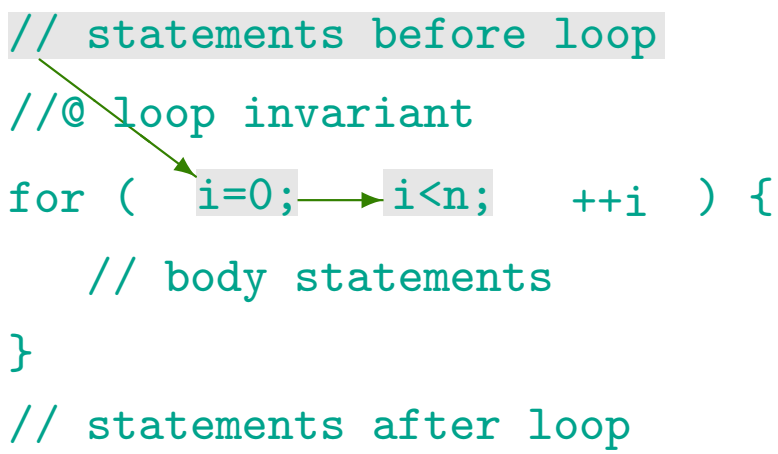
- Control flow of a `for` loop

Where does a loop invariant hold?

```
// statements before loop
//@ loop invariant
for (   i=0;    i<n;    ++i  ) {
   // body statements
}
// statements after loop
```

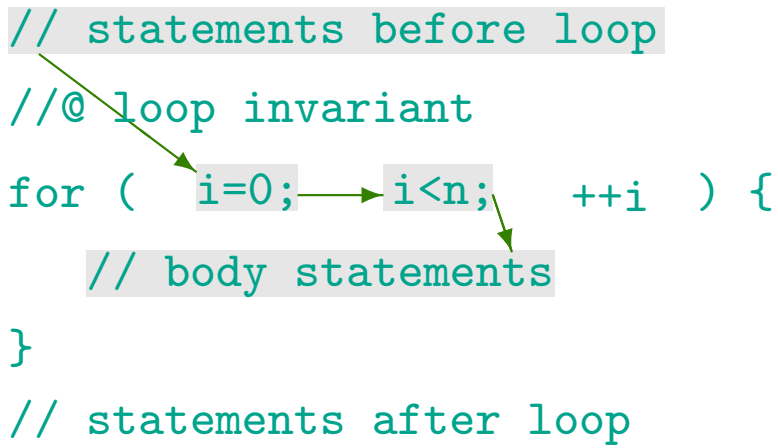
- Control flow of a `for` loop

Where does a loop invariant hold?



- Control flow of a `for` loop

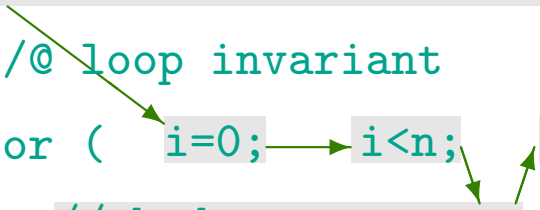
Where does a loop invariant hold?



- Control flow of a `for` loop

Where does a loop invariant hold?

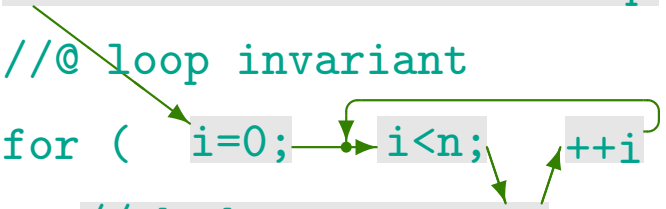
```
// statements before loop
//@ loop invariant
for (   i=0;   i<n;   ++i  ) {
    // body statements
}
// statements after loop
```



- Control flow of a `for` loop

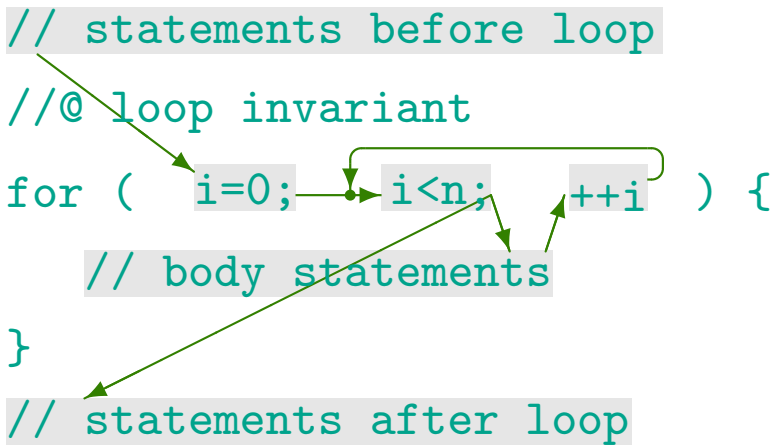
Where does a loop invariant hold?

```
// statements before loop
//@ loop invariant
for (   i=0;    i<n;    ++i  ) {
    // body statements
}
// statements after loop
```



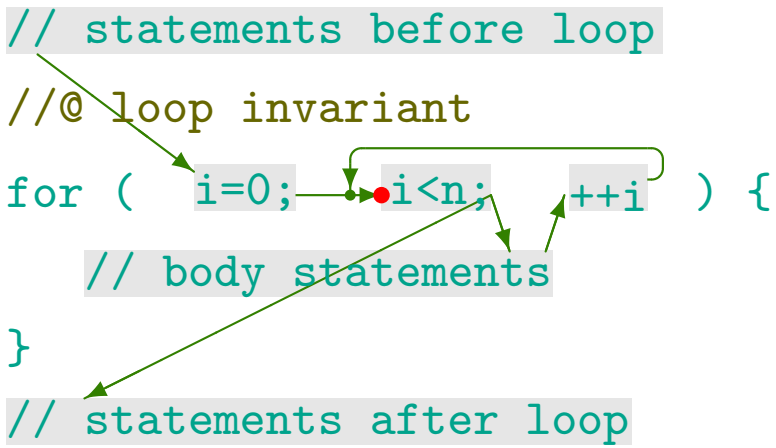
- Control flow of a `for` loop

Where does a loop invariant hold?



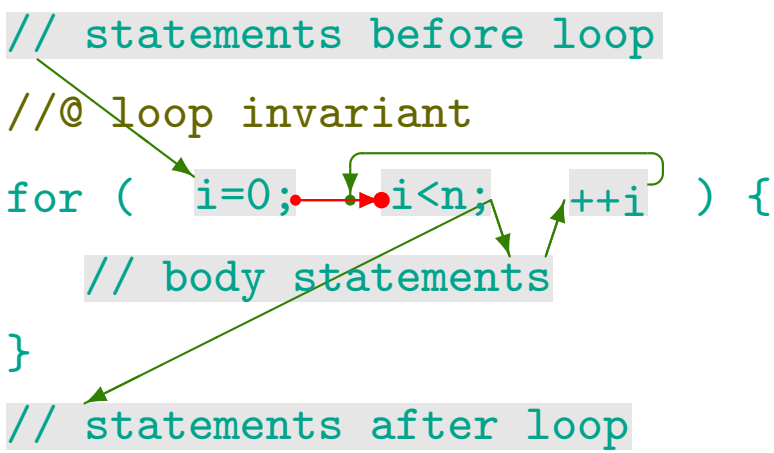
- Control flow of a `for` loop
- "*Where*" means: "*at which arcs of the control flow graph*" rather than: "*at which line/column of the source text*"

Where does a loop invariant hold?



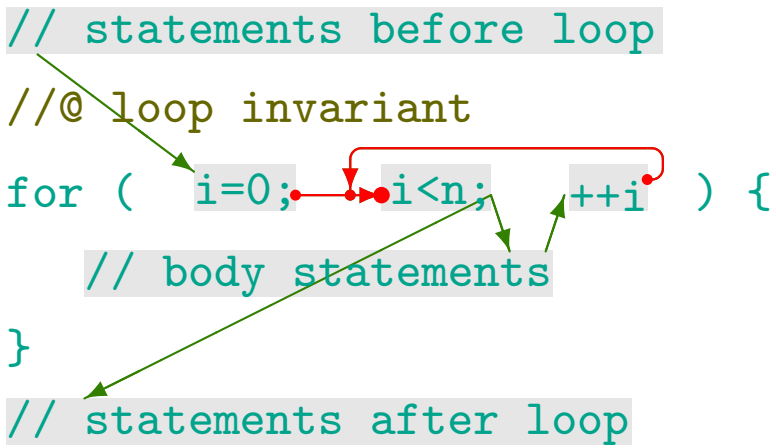
- The loop invariant has to hold immediately before `test`,

Where does a loop invariant hold?



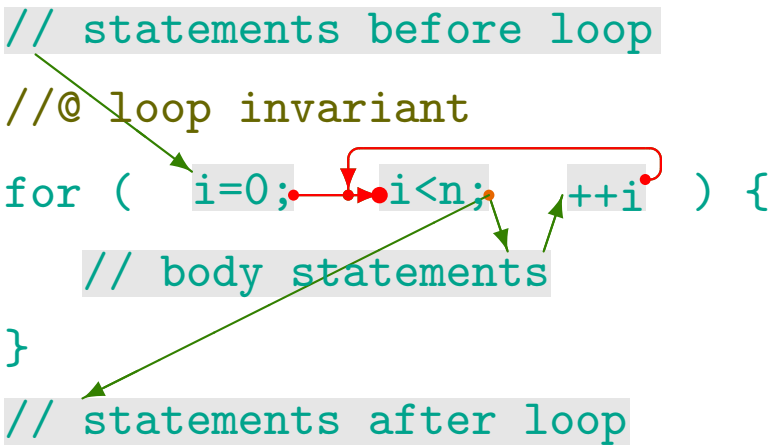
- The loop invariant has to hold immediately before `test`,
- hence also immediately after `init`

Where does a loop invariant hold?



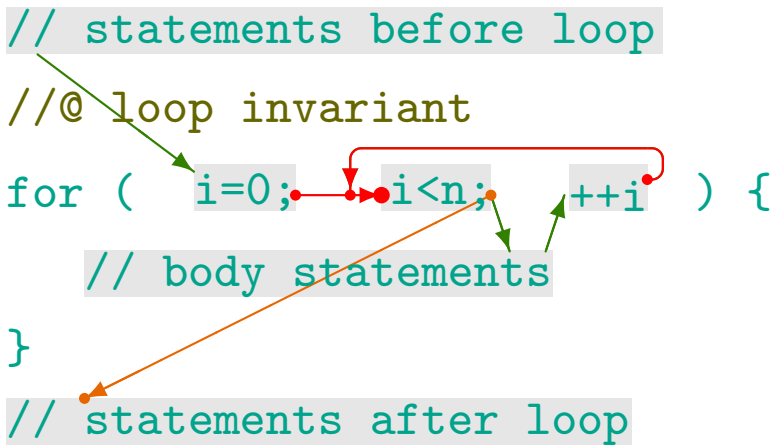
- The loop invariant has to hold immediately before `test`,
- hence also immediately after `init` and `incr`.

Where does a loop invariant hold?



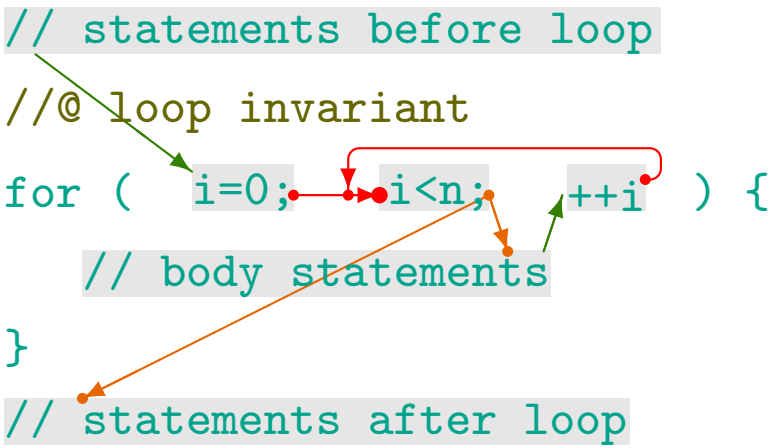
- The loop invariant has to hold immediately before `test`,
- hence also immediately after `init` and `incr`.
- If `test` has no side effects,
  the loop invariant has to hold also immediately after `test`,

Where does a loop invariant hold?



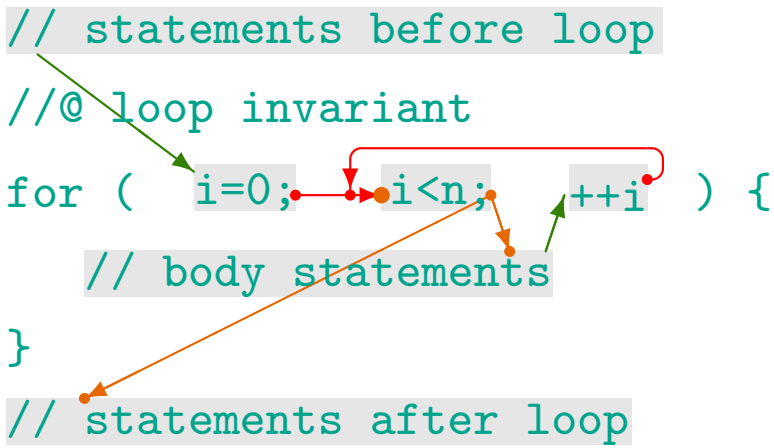
- The loop invariant has to hold immediately before `test`,
- hence also immediately after `init` and `incr`.
- If `test` has no side effects,
  the loop invariant has to hold also immediately after `test`,
- hence also immediately after the loop

Where does a loop invariant hold?



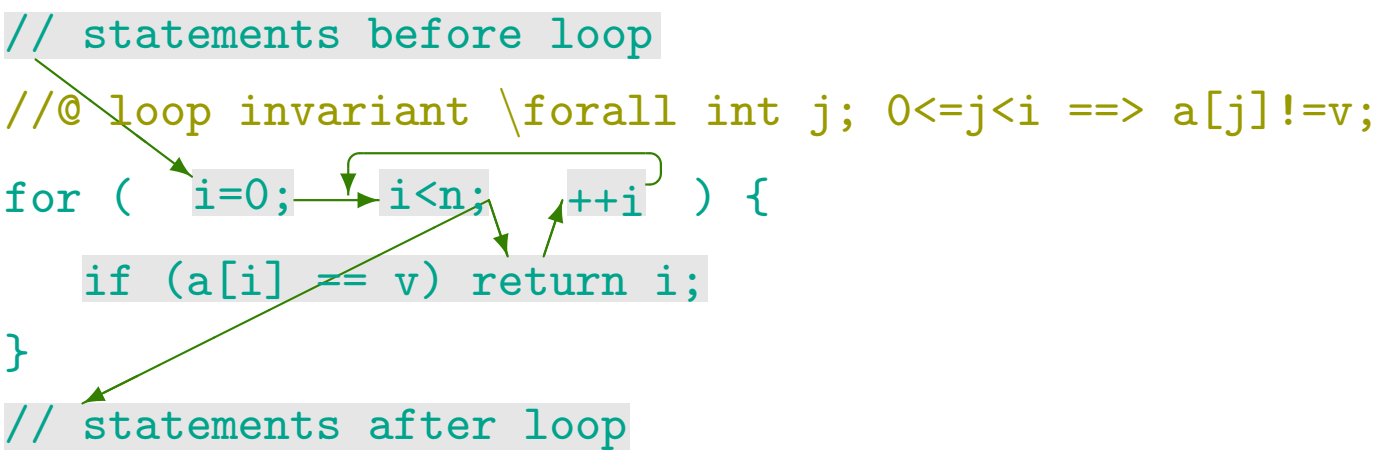
- The loop invariant has to hold immediately before `test`,
- hence also immediately after `init` and `incr`.
- If `test` has no side effects,
  the loop invariant has to hold also immediately after `test`,
- hence also immediately after the loop and before `body`.

Where does a loop invariant hold?



- The loop invariant has to hold immediately before `test`,
- hence also immediately after `init` and `incr`.
- If `test` has no side effects,
  the loop invariant has to hold also immediately after `test`,
- hence also immediately after the loop and before `body`.
- It does usually **not** hold before `init`, inside or after `body` or before `incr`!

Where does a loop invariant hold?

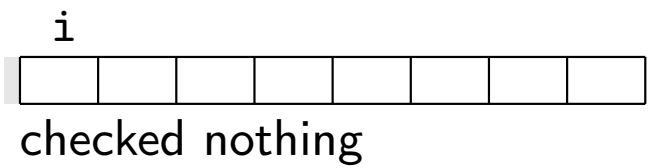


- Considering our `find` example, its 2nd loop invariant

Where does a loop invariant hold?

```
// statements before loop
//@ loop invariant \forall int j; 0<=j<i ==> a[j]!=v;
for (   i=0;      i<n;      ++i   ) {
    if (a[i] == v) return i;
}
// statements after loop
```

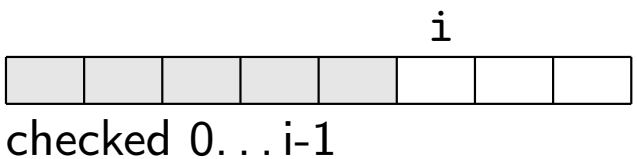
- Considering our `find` example, its 2nd loop invariant
- adequately describes the state after `init`



Where does a loop invariant hold?

```
// statements before loop
//@ loop invariant \forall int j; 0<=j<i ==> a[j]!=v;
for (   i=0;      i<n;      ++i  ) {
    if (a[i] == v) return i;
}
// statements after loop
```

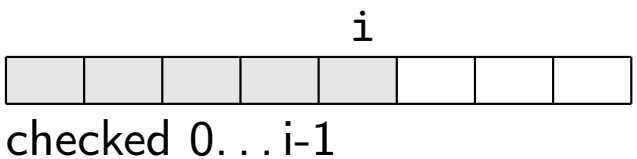
- Considering our `find` example, its 2nd loop invariant
- adequately describes the state after `init`
- and after `incr`



Where does a loop invariant hold?

```
// statements before loop
//@ loop invariant \forall int j; 0<=j<i ==> a[j]!=v;
for (   i=0;      i<n;      ++i   ) {
    if (a[i] == v) return i;
}
// statements after loop
```

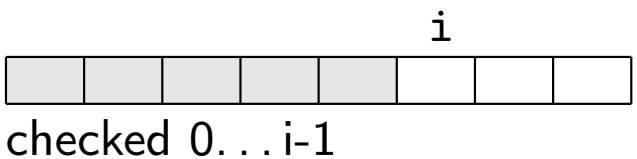
- Considering our `find` example, its 2nd loop invariant
- adequately describes the state after `init`
- and after `incr`,
- but it would be too weak to describe the state before `incr`.



Where does a loop invariant hold?

```
// statements before loop
//@ loop invariant \forall int j; 0<=j<i ==> a[j]!=v;
for (   i=0;      i<n;      ++i   ) {
    if (a[i] == v) return i;
}
// statements after loop
```

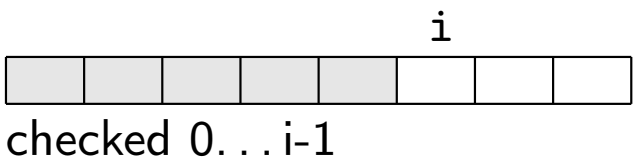
- Considering our `find` example, its 2nd loop invariant
- adequately describes the state after `init`
- and after `incr`



Where does a loop invariant hold?

```
// statements before loop
//@ loop invariant \forall int j; 0<=j<i ==> a[j]!=v;
for (   i=0;      i<n;      ++i   ) {
    if (a[i] == v) return i;
}
// statements after loop
```

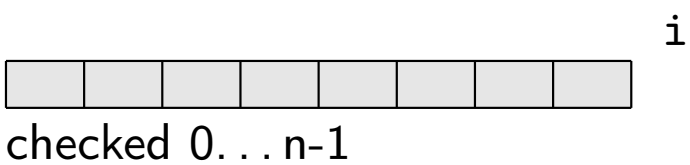
- Considering our `find` example, its 2nd loop invariant
- adequately describes the state after `init`
- and after `incr`.
- If the loop terminates normally,



Where does a loop invariant hold?

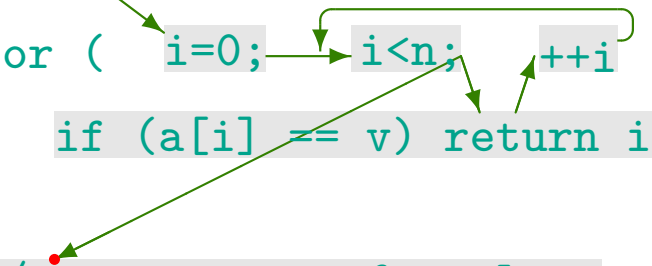
```
// statements before loop
//@ loop invariant \forall int j; 0<=j<i ==> a[j]!=v;
for (   i=0;      i<n;      ++i   ) {
    if (a[i] == v) return i;
}
// statements after loop
```

- Considering our `find` example, its 2nd loop invariant
- adequately describes the state after `init`
- and after `incr`.
- If the loop terminates normally,
- we know that all array elements have been checked (without success)

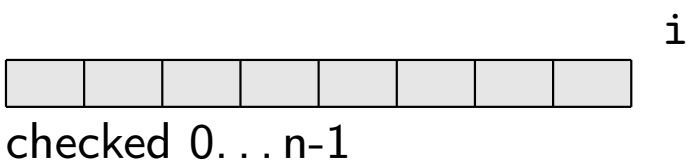


Where does a loop invariant hold?

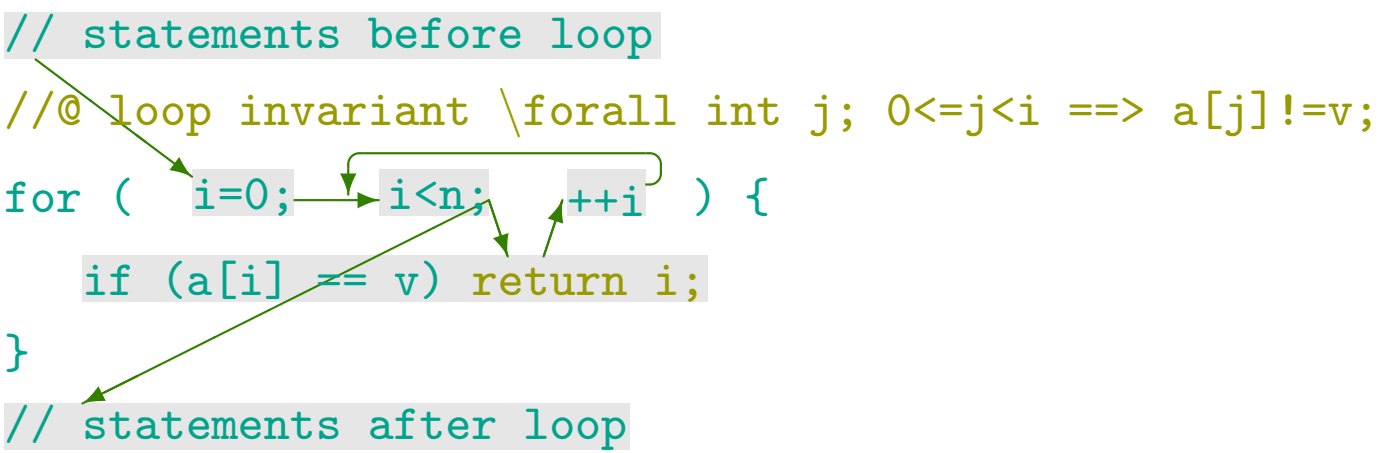
```
// statements before loop
//@ loop invariant \forall int j; 0<=j<i ==> a[j]!=v;
for (   i=0;      i<n;      ++i   ) {
    if (a[i] == v) return i;
}
// statements after loop
```



- Considering our `find` example, its 2nd loop invariant
- adequately describes the state after `init`
- and after `incr`.
- If the loop terminates normally,
- we know that all array elements have been checked (without success)
- and hence returning -1 will satisfy our contract.

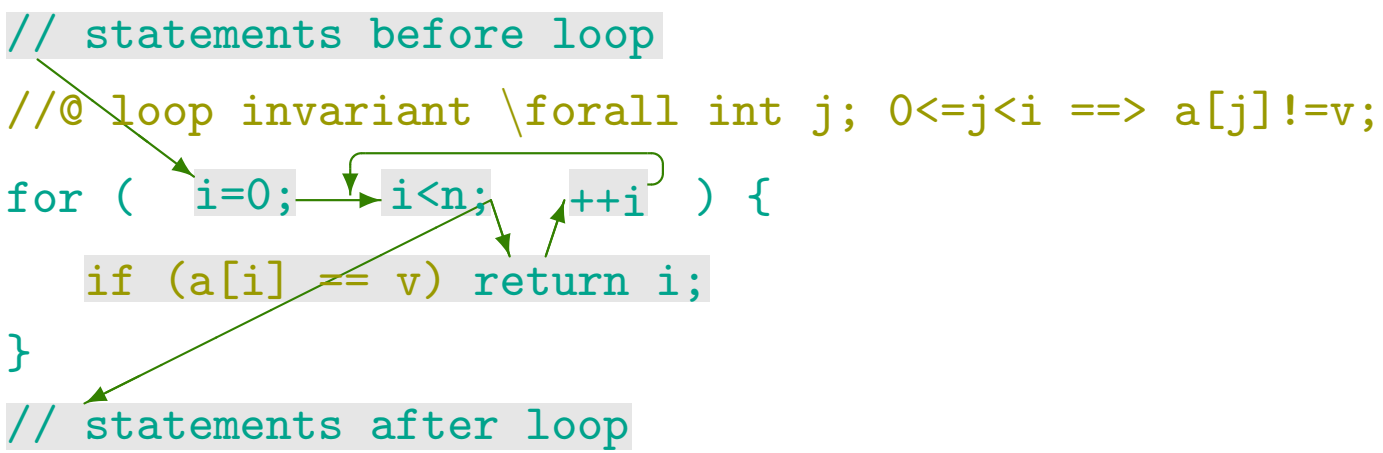


Where does a loop invariant hold?



- Considering our `find` example, its 2nd loop invariant
- adequately describes the state after `init`
- and after `incr`.
- If the loop terminates normally,
- we know that all array elements have been checked (without success)
- and hence returning -1 will satisfy our contract.
- If the loop is terminated by the `return`,

Where does a loop invariant hold?



- Considering our `find` example, its 2nd loop invariant
- adequately describes the state after `init`
- and after `incr`.
- If the loop terminates normally,
- we know that all array elements have been checked (without success)
- and hence returning -1 will satisfy our contract.
- If the loop is terminated by the `return`,
- we obtain the postcondition directly from the preceding `if`.

References in `requires` and `ensures` clauses

```
double updateMean(int *sum,int *cnt,int new)
{
    *sum += new;
    *cnt += 1;
    return (double)*sum / (double)*cnt;      // mean value so far
}
```

- Example: incrementally computing the mean value of an `int` sequence

References in `requires` and `ensures` clauses

```
/*@




 */
double updateMean(int *sum,int *cnt,int new)
{
    *sum += new;
    *cnt += 1;
    return (double)*sum / (double)*cnt;      // mean value so far
}
```

- Example: incrementally computing the mean value of an `int` sequence
- What should the function contract look like?

References in `requires` and `ensures` clauses

```
/*@
    requires \valid(sum) && \valid(cnt);



 */
double updateMean(int *sum,int *cnt,int new)
{
    *sum += new;
    *cnt += 1;
    return (double)*sum / (double)*cnt;      // mean value so far
}
```

- Example: incrementally computing the mean value of an `int` sequence
- What should the function contract look like?
- Require that pointers `sum` and `cnt` point to valid `int`s

References in `requires` and `ensures` clauses

```
/*@
    requires \valid(sum) && \valid(cnt);
    requires 0 <= *cnt;


 */
double updateMean(int *sum,int *cnt,int new)
{
    *sum += new;
    *cnt += 1;
    return (double)*sum / (double)*cnt;      // mean value so far
}
```

- Example: incrementally computing the mean value of an `int` sequence
- What should the function contract look like?
- Require that pointers `sum` and `cnt` point to valid `int`s
- Require `*cnt >= 0` to prevent zero division (overflow-issue ignored)

References in `requires` and `ensures` clauses

```
/*@
    requires \valid(sum) && \valid(cnt);
    requires 0 <= *cnt;


 */
double updateMean(int *sum,int *cnt,int new)
{
    *sum += new;
    *cnt += 1;
    return (double)*sum / (double)*cnt;      // mean value so far
}
```

- Example: incrementally computing the mean value of an `int` sequence
- What should the function contract look like?
- Require that pointers `sum` and `cnt` point to valid `int`s
- Require `*cnt >= 0` to prevent zero division (overflow-issue ignored)
- How to express updates of `sum` and `cnt`?

References in `requires` and `ensures` clauses

```
/*@
    requires \valid(sum) && \valid(cnt);
    requires 0 <= *cnt;
    ensures  *cnt = *cnt + 1;
    ensures  *sum = *sum + new;
 */
double updateMean(int *sum,int *cnt,int new)
{
    *sum += new;
    *cnt += 1;
    return (double)*sum / (double)*cnt;      // mean value so far
}
```

- Example: incrementally computing the mean value of an `int` sequence
- How to express updates of `sum` and `cnt`?

References in `requires` and `ensures` clauses

```
/*@
    requires \valid(sum) && \valid(cnt);
    requires 0 <= *cnt;
    ensures  *cnt = *cnt + 1;
    ensures  *sum = *sum + new;
 */
double updateMean(int *sum,int *cnt,int new)
{
    *sum += new;
    *cnt += 1;
    return (double)*sum / (double)*cnt;      // mean value so far
}
```

- Example: incrementally computing the mean value of an `int` sequence
- How to express updates of `sum` and `cnt`?
- No assigments allowed in contract

References in `requires` and `ensures` clauses

```
/*@
    requires \valid(sum) && \valid(cnt);
    requires 0 <= *cnt;
    ensures  *cnt == *cnt + 1;
    ensures  *sum == *sum + new;
 */
double updateMean(int *sum,int *cnt,int new)
{
    *sum += new;
    *cnt += 1;
    return (double)*sum / (double)*cnt;      // mean value so far
}
```

- Example: incrementally computing the mean value of an `int` sequence
- How to express updates of `sum` and `cnt`?
- No assigments allowed in contract

References in `requires` and `ensures` clauses

```
/*@
    requires \valid(sum) && \valid(cnt);
    requires 0 <= *cnt;
    ensures  *cnt == *cnt + 1;
    ensures  *sum == *sum + new;
 */
double updateMean(int *sum,int *cnt,int new)
{
    *sum += new;
    *cnt += 1;
    return (double)*sum / (double)*cnt;      // mean value so far
}
```

- Example: incrementally computing the mean value of an `int` sequence
- How to express updates of `sum` and `cnt`?
- No assigments allowed in contract
- Obviously nonsense: e.g. first equation implies `0 == 1`

References in `requires` and `ensures` clauses

```
/*@
    requires \valid(sum) && \valid(cnt);
    requires 0 <= *cnt;
    ensures  *cnt == *cnt + 1;
    ensures  *sum == *sum + new;
 */
double updateMean(int *sum,int *cnt,int new)
{
    *sum += new;
    *cnt += 1;
    return (double)*sum / (double)*cnt;      // mean value so far
}
```

- Example: incrementally computing the mean value of an `int` sequence
- How to express updates of `sum` and `cnt`?
- No assigments allowed in contract
- Obviously nonsense: e.g. first equation implies `0 == 1`
- Use `\old(·)` to refer to the value of an expression on function entry

References in `requires` and `ensures` clauses

```
/*@
    requires \valid(sum) && \valid(cnt);
    requires 0 <= *cnt;
    ensures  *cnt == \old(*cnt) + 1;
    ensures  *sum == \old(*sum) + new;
 */
double updateMean(int *sum,int *cnt,int new)
{
    *sum += new;
    *cnt += 1;
    return (double)*sum / (double)*cnt;      // mean value so far
}
```

- Example: incrementally computing the mean value of an `int` sequence
- How to express updates of `sum` and `cnt`?
- No assigments allowed in contract
- Obviously nonsense: e.g. first equation implies `0 == 1`
- Use `\old(·)` to refer to the value of an expression on function entry

References in `requires` and `ensures` clauses

```
/*@
    requires \valid(sum) && \valid(cnt);
    requires 0 <= *cnt;
    ensures  *cnt == \old(*cnt) + 1;
    ensures  *sum == \old(*sum) + new;
 */
double updateMean(int *sum,int *cnt,int new)
{
    *sum += new;
    *cnt += 1;
    return (double)*sum / (double)*cnt;      // mean value so far
}
```

- Rules:

References in `requires` and `ensures` clauses

```
/*@
    requires \valid(sum) && \valid(cnt);
    requires 0 <= *cnt;
    ensures  *cnt == \old(*cnt) + 1;
    ensures  *sum == \old(*sum) + new;
 */
double updateMean(int *sum,int *cnt,int new)
{
    *sum += new;
    *cnt += 1;
    return (double)*sum / (double)*cnt;      // mean value so far
}
```

- Rules:
- In `requires`, an expression refers to its value on function entry

References in `requires` and `ensures` clauses

```
/*@
    requires \valid(sum) && \valid(cnt);
    requires 0 <= *cnt;
    ensures  *cnt == \old(*cnt) + 1;
    ensures  *sum == \old(*sum) + new;
 */
double updateMean(int *sum,int *cnt,int new)
{
    *sum += new;
    *cnt += 1;
    return (double)*sum / (double)*cnt;      // mean value so far
}
```

- Rules:
- In `requires`, an expression refers to its value on function entry
- In `ensures`, an expression refers to its value on function exit

References in `requires` and `ensures` clauses

```
/*@
    requires \valid(sum) && \valid(cnt);
    requires 0 <= *cnt;
    ensures  *cnt == \old(*cnt) + 1;
    ensures  *sum == \old(*sum) + new;
 */
double updateMean(int *sum,int *cnt,int new)
{
    *sum += new;
    *cnt += 1;
    return (double)*sum / (double)*cnt;      // mean value so far
}
```

- Rules:
- In `requires`, an expression refers to its value on function entry
- In `ensures`, an expression refers to its value on function exit
- In `ensures`, an expression within `old(·)` refers to its value on function entry

Another example: Euklid's gcd algorithm

```
/*@ requires A > 0 && B > 0;
    ensures \forall int i;
        div(i,A) && div(i,B) <==> div(i,\result);
*/                  // div(x,y):  x is a divisor of y
int gcd(int A, int B) {
    int const a = A, b = B;
    /*@ loop variant a + b;
        loop invariant a > 0 && b > 0;
        loop invariant \forall int i;
            div(i,a) && div(i,b) <==> div(i,A) && div(i,B);
    */
    while (a != b)
        if (a > b)
             a -= b;
        else
             b -= a;
    return a;
}
```

Another example: Euklid's gcd algorithm

```
/*@ requires A > 0 && B > 0;
    ensures \forall int i;
        div(i,A) && div(i,B) <==> div(i,\result);
*/                  // div(x,y):  x is a divisor of y
int gcd(int A, int B) {
    int const a = A, b = B;
    /*@ loop variant a + b;
        loop invariant a > 0 && b > 0;
        loop invariant \forall int i;
            div(i,a) && div(i,b) <==> div(i,A) && div(i,B);
    */
    while (a != b)
        if (a > b)
            a -= b;
        else
            b -= a;
    return a;
}
```

- First algorithm at all: 300 BC [Fit08, Bk.7,Pr.1]

Another example: Euklid's gcd algorithm

```
/*@ requires A > 0 && B > 0;
    ensures \forall int i;
        div(i,A) && div(i,B) <==> div(i,\result);
*/                  // div(x,y):  x is a divisor of y
int gcd(int A, int B) {
    int const a = A, b = B;
    /*@ loop variant a + b;
        loop invariant a > 0 && b > 0;
        loop invariant \forall int i;
            div(i,a) && div(i,b) <==> div(i,A) && div(i,B);
    */
    while (a != b)
        if (a > b)
            a -= b;
        else
            b -= a;
    return a;
}
```

- First algorithm at all: 300 BC [Fit08, Bk.7,Pr.1]
- Shows complementary set of `C`:
  `while` loop, numeric computation,
  nontrivial `loop variant`, infinite quantifier range

Another example: Euklid's gcd algorithm

```
/*@ requires A > 0 && B > 0;
    ensures \forall int i;
        div(i,A) && div(i,B) <==> div(i,\result);
*/                  // div(x,y):  x is a divisor of y
int gcd(int A, int B) {
    int const a = A, b = B;
    /*@ loop variant a + b;
        loop invariant a > 0 && b > 0;
        loop invariant \forall int i;
            div(i,a) && div(i,b) <==> div(i,A) && div(i,B);
    */
    while (a != b)
        if (a > b)
            a -= b;
        else
            b -= a;
    return a;
}
```

- Example computation:

a:	**72**	**30**		**18**	**6**
b:	**42**		**12**		**6**

Another example: Euklid's gcd algorithm

```
/*@ requires A > 0 && B > 0;
    ensures \forall int i;
        div(i,A) && div(i,B) <==> div(i,\result);
*/                  // div(x,y):  x is a divisor of y
int gcd(int A, int B) {
    int const a = A, b = B;
    /*@ loop variant a + b;
        loop invariant a > 0 && b > 0;
        loop invariant \forall int i;
            div(i,a) && div(i,b) <==> div(i,A) && div(i,B);
    */
    while (a != b)
        if (a > b)
            a -= b;
        else
            b -= a;
    return a;
}
```

- Example computation:

a:	**72**	**30**		**18**	**6**
b:	**42**		**12**		**6**

- $6 = 2 \cdot 3$ is in fact the greatest common divisor of $72 = 2 \cdot 2 \cdot 2 \cdot 3 \cdot 3$ and $42 = 2 \cdot 3 \cdot 7$

Another example: Euklid's gcd algorithm

```
/*@ requires A > 0 && B > 0;
    ensures \forall int i;
        div(i,A) && div(i,B) <==> div(i,\result);
*/                  // div(x,y):  x is a divisor of y
int gcd(int A, int B) {
    int const a = A, b = B;
    /*@ loop variant a + b;
        loop invariant a > 0 && b > 0;
        loop invariant \forall int i;
            div(i,a) && div(i,b) <==> div(i,A) && div(i,B);
    */
    while (a != b)
        if (a > b)
            a -= b;
        else
            b -= a;
    return a;
}
```

- Example computation:

a:	**72**	**30**		**18**	**6**
b:	**42**		**12**		**6**

- $6 = 2 \cdot 3$ is in fact the greatest common divisor of $72 = 2 \cdot 2 \cdot 2 \cdot 3 \cdot 3$ and $42 = 2 \cdot 3 \cdot 7$
- Contract: "The divisors of `\result` are exactly the common divisors of `A` and `B`"

Another example: Euklid's gcd algorithm

```
/*@ requires A > 0 && B > 0;
    ensures \forall int i;
        div(i,A) && div(i,B) <==> div(i,\result);
*/                  // div(x,y):  x is a divisor of y
int gcd(int A, int B) {
    int const a = A, b = B;
    /*@ loop variant a + b;
        loop invariant a > 0 && b > 0;
        loop invariant \forall int i;
            div(i,a) && div(i,b) <==> div(i,A) && div(i,B);
    */
    while (a != b)
        if (a > b)
            a -= b;
        else
            b -= a;
    return a;
}
```

- $6 = 2 \cdot 3$ is in fact the greatest common divisor of $72 = 2 \cdot 2 \cdot 2 \cdot 3 \cdot 3$ and $42 = 2 \cdot 3 \cdot 7$
- Contract: "The divisors of `\result` are exactly the common divisors of `A` and `B`"
- `\result` is the greatest divisor of itself, and hence the greatest common divisor of `A` and `B`

Predicate definition of `div`

```
/*@ axiomatic Euklid {
    predicate div(integer a, integer b) =
        (0 < a && 0 <= b && b % a == 0);
    axiom A1:
        \forall integer a, b, c;
            b >= c && div(a,b) && div(a,c) ==> div(a,b-c);
    axiom A2:
        \forall integer a, b, c;
            b >= c && div(a,c) && div(a,b-c) ==> div(a,b);
    }
*/
```

References


Richard Fitzpatrick, editor.
Euklid's Elements of Geometry.
Austin/TX, 2008.


C.A.R. Hoare.
An axiomatic basis for computer programming.
C.ACM, 12:576–583, 1969.


A.M. Turing.
On computable numbers, with an application to the Entscheidungsproblem.
Proc. London Math. Soc., Ser. 2, 42:230–265, 1936.