ACSL By Example

Towards a Verified C Standard Library

Version 11.1.1
for
Frama-C (Sodium)
June 2015

Jochen Burghardt
Jens Gerlach
Timon Lapawczyk

Former Authors

Andreas Carben
Liangliang Gu
Kerstin Hartig
Hans Pohl
Juan Soto
Kim Völlinger

Fraunhofer
FOKUS

The research leading to these results has received funding from the STANCE project[1] within European Union's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 317753.[2]

This body of work was completed within the Device-Soft project, which was supported by the Programme Inter Carnot Fraunhofer from BMBF (Grant 01SF0804) and ANR.[3]

[1]See `http://www.stance-project.eu`
[2]project duration: 2012–2015
[3]project duration: 2009–2012

Foreword

This report provides various examples for the formal specification, implementation, and deductive verification of C programs using the ANSI/ISO-C Specification Language (ACSL [1]) and the WP plug-in [2] of Frama-C [3] (Framework for Modular Analysis of C programs). The report at hand has been revised and refers to the *Sodium* release from March 2015 of Frama-C.

We have chosen our examples from the C++ standard library whose initial version is still known as the *Standard Template Library* (STL).[4] The STL contains a broad collection of *generic* algorithms that work not only on C arrays but also on more elaborate containers, i.e., data structures. For the purposes of this document we have selected representative algorithms, and converted their implementation from C++ function templates to C functions that work on arrays of type **int**.[5]

We will continue to extend and refine this report by describing additional STL algorithms and data structures. Thus, step by step, this document will evolve from an ACSL tutorial to a report on a formally specified and deductively verified standard library for ANSI/ISO-C. Moreover, should ACSL be extended to a C++ specification language, our work may be extended to a deductively verified C++ Standard Library.

You may email suggestions, errors or greetings(!) to

jens.gerlach@fokus.fraunhofer.de

In particular, we encourage you to check vigilantly whether our formal specifications capture the essence of the informal description of the STL algorithms.

We appreciate your feedback and hope that this document helps foster the adoption of deductive verification techniques.

Acknowledgement

Many members from the Frama-C community provided valuable input and comments during the course of the development of this document. In particular, we wish to thank our project partners Patrick Baudin, Loïc Correnson, Zaynah Dargaye, Florent Kirchner, Virgile Prevosto and Armand Puccetti from CEA LIST and Pascal Cuoq from TrustInSoft[6].

We also like to express our gratitude to Claude Marché (LRI/INRIA) and Yannick Moy (AdaCore) for their helpful comments and detailed suggestions for improvement.

[4] See `http://www.sgi.com/tech/stl/`
[5] We are not directly using **int** in the source code but rather `value_type` which is a **typedef** for **int**.
[6] http://trust-in-soft.com

Changes for Version 11.1.1 (June 2015)

For changes in previous versions see Appendix A.

- add Chapter 7 on numeric algorithms
    - move `iota` algorithm to numeric algorithms (Section 7.1)
    - add `accumulate` algorithm (Section 7.2)
    - add `inner_product` algorithm (Section 7.3)
    - add `partial_sum` algorithm (Section 7.4)
    - add `adjacent_difference` algorithm (Section 7.5)

Contents

List of Logic Specifications

List of Figures

List of Tables

1. Introduction

The Framework for Modular Analyses of C, Frama-C [3], is a suite of software tools dedicated to the analysis of C source code. Its development efforts are conducted and coordinated at two French public institutions: CEA LIST [4], a laboratory of applied research on software-intensive technologies, and INRIA Saclay[5], the French National Institute for Research in Computer Science and Control in collaboration with LRI [6], the Laboratory for Computer Science at Université Paris-Sud.

ACSL (ANSI/ISO-C Specification Language) [1] is a formal language to express behavioral properties of C programs. This language can specify a wide range of functional properties by adding annotations to the code. It allows to create function contracts containing preconditions and postconditions. It is possible to define type and global invariants as well as logic specifications, such as predicates, lemmas, axioms or logic functions. Furthermore, ACSL allows statement annotations such as assertions or loop annotations.

Within Frama-C, the WP plug-in [2] enables deductive verification of C programs that have been annotated with ACSL. The WP plug-in uses Hoare-style weakest precondition computations to formally prove ACSL properties of C code. Verification conditions are generated and submitted to external automatic theorem provers or interactive proof assistants.

The Verification Group at Fraunhofer FOKUS[7] see the great potential for deductive verification using ACSL. However, we recognize that for a novice there are challenges to overcome in order to effectively use the WP plug-in for deductive verification. In order to help users gain confidence, we have written this tutorial that demonstrates how to write annotations for existing C programs. This document provides several examples featuring a variety of annotated functions using ACSL. For an in-depth understanding of ACSL, we strongly recommend users to read the official Frama-C introductory tutorial [8] first. The principles presented in this paper are also documented in the ACSL reference document [9].

1.1. Structure of this document

The functions presented in this document were selected from the C++ standard library. The original C++ implementation was stripped from its generic implementation and mapped to C arrays of type `value_type`.

Chapter 2 provides a short introduction into the Hoare Calculus.

We have grouped various standard algorithms algorithms in chapters as follows:

- non-mutating algorithms (Chapter 3)
- maximum/minimum algorithms (Chapter 4)

- binary search algorithms (Chapter 5)
- mutating algorithms (Chapter 6)
- numeric algorithms (Chapter 7)

The order of these chapters reflects their increasing complexity.

Using the example of a stack, we tackle in Chapter 8 the problem of how a data type and its associated C functions can be specified with ACSL and automatically verified with Frama-C.

1.2. Types, arrays, ranges and valid indices

This section describe several general conventions and basic definitions we use throughout this document.

1.2.1. Types

In order to keep algorithms and specifications as general as possible, we use abstract type names on almost all occasions. We currently defined the following types:

```
typedef int value_type;

typedef unsigned int size_type;

typedef int bool;
```

Programmers who know the types associated with C++ standard library containers will not be surprised that `value_type` refers to the type of values in an array whereas `size_type` will be used for the indices of an array.

This approach allows one to modify e.g. an algorithm working on an `int` array to work on a `char` array by changing only one line of code, viz. the `typedef` of `value_type`. Moreover, we believe in better readability as it becomes clear whether a variable is used as an index or as a memory for a copy of an array element, just by looking at its type.

The latter reason also applies to the use of `bool`. To denote values of that type, we `#defined` the identifiers `false` and `true` to be `0` and `1`, respectively. While any non-zero value is accepted to denote `true` in ACSL like in C the algorithms shown in this tutorial will always produce `1` for `true`. Due to the above definitions, the ACSL truth-value constant `\false` and `\true` can be used interchangeably with our `false` and `true`, respectively, in ACSL clauses, but not in C code.

1.2.2. Array and ranges

The C Standard describes an array as a "contiguously allocated nonempty set of objects" [10, §6.2.5.20]. If `n` is a constant integer expression with a value greater than zero, then

```
int a[n];
```

describes an array of type `int`. In particular, for each `i` that is greater than or equal to 0 and less than `n`, we can dereference the pointer `a+i`.

Let the following prototype represent a function, whose first argument is the address to a range and whose second argument is the length of this range.

```
void example(value_type* a, size_type n);
```

To be very precise, we have to use the term *range* instead of *array*. This is due to the fact, that functions may be called with empty ranges, i.e., with `n == 0`. Empty arrays, however, are not permitted according to the definition stated above. Nevertheless, we often use the term *array* and *range* interchangeably.

1.2.3. Specification of valid ranges in ACSL

The following ACSL fragment expresses the precondition that the function `example` expects that for each `i`, such that `0 <= i < n`, the pointer `a+i` may be safely dereferenced.

```
/*@
    requires 0 <= n;
    requires \valid(a+(0.. n-1));
*/
void example(value_type* a, size_type n);
```

In this case we refer to each index `i` with `0 <= i < n` as a *valid index* of `a`.

ACSL's built-in predicates `\valid(a + (0.. n))` and `\valid_read(a + (0.. n))` refer to all addresses `a+i` where `0 <= i <= n`. However, the array notation `int a[n]` of the C programming language refers only to the elements `a+i` where `i` satisfies `0 <= i < n`. Users of ACSL must therefore use the range notation `a+(0.. n-1)` in order to express a valid array of length `n`.

2. The Hoare calculus

In 1969, C.A.R. Hoare introduced a calculus for formal reasoning about properties of imperative programs [11], which became known as "Hoare Calculus".

The basic notion is

```
//@ assert P;
Q;
//@ assert R;
```

where `P` and `R` denote logical expressions and `Q` denotes a source-code fragment. Informally, this means "If `P` holds before the execution of `Q`, then `R` will hold after the execution". Usually, `P` and `R` are called "precondition" and "postcondition" of `Q`, respectively. The syntax for logical expressions is described in [9, Section 2.2] in full detail. For the purposes of this tutorial, the notions shown in Table 2.1 are sufficient. Note that they closely resemble the logical and relational operators in C.

`!P`	negation	"`P` is not true"
`P && Q`	conjunction	"`P` is true and `Q` is true"
`P || Q`	disjunction	"`P` is true or `Q` is true"
`P ==> Q`	implication	"if `P` is true, then `Q` is true"
`P <==> Q`	equivalence	"if, and only if, `P` is true, then `Q` is true"
`x < y == z`	relation chain	"`x` is less than `y` and `y` is equal to `z`"
`\forall int x; P(x)`	universal quantifier	"`P(x)` is true for every **`int`** value of `x`"
`\exists int x; P(x)`	existential quantifier	"`P(x)` is true for some **`int`** value of `x`"

Table 2.1.: Some ACSL formula syntax

The Listings 2.2 and 2.3 shows three example source-code fragments and annotations.

```
//@ assert x % 2 == 1;
++x;
//@ assert x % 2 == 0;
```

(a)

```
//@ assert 0 <= x <= y;
++x;
//@ assert 0 <= x <= y + 1;
```

(b)

Listing 2.2: Example source code fragments and annotations

```
//@ assert true;
while (--x != 0)
    sum += a[x];
//@ assert x == 0;
```

Listing 2.3: Loop source code fragments and annotations

Their informal meanings are as follows:

Listing 2.2 (a) "If `x` has an odd value before execution of the code `++x` then `x` has an even value thereafter."

Listing 2.2 (b) "If the value of `x` is in the range $\{0, \ldots, \texttt{y}\}$ before execution of the same code, then `x`'s value is in the range $\{0, \ldots, \texttt{y} + 1\}$ after execution."

Listing 2.3 "Under any circumstances, the value of `x` is zero after execution of the loop code."

Any C programmer will confirm that these properties are valid.[7] The examples were chosen to demonstrate also the following issues:

- For a given code fragment, there does not exist one fixed pre- or postcondition. Rather, the choice of formulas depends on the actual property to be verified, which comes from the application context. The two examples in Listing 2.2 share the same code fragment, but have different pre- and postconditions.
- The postcondition need not be the most restricting possible formula that can be derived. In Listing 2.3, it is not an error that we stated only that `0 <= x` although we know that even `1 <= x`.
- In particular, pre- and postconditions need not contain all variables appearing in the code fragment. Neither `sum` nor `a[]` is referenced in the formulas of Listing 2.3.
- We can use the predicate **`true`** to denote the absence of a properly restricting precondition, as we did in Listing 2.3.
- It is not possible to express by pre- and postconditions that a given piece of code will always terminate. Listing 2.3 only states that *if* the loop terminates, then `x == 0` will hold. In fact, if `x` has a negative value on entry, the loop will run forever. However, if the loop terminates, `x == 0` will hold, and that is what Listing 2.3 claims.

  Usually, termination issues are dealt with separately from correctness issues. Termination proofs may, however, refer to properties stated (and verified) using the Hoare Calculus.

Hoare provided the rules shown in Listing 2.4 to 2.14 in order to reason about programs. We will comment on them in the following sections.

[7] We leave the important issues of overflow aside for a moment.

2.1. The assignment rule

We start with the rule that is probably the least intuitive of all Hoare-Calculus rules, viz. the assignment rule. It is depicted in Listing 2.4, where "`P {x↦e}`" denotes the result of substituting each occurrence of `x` in `P` by `e`.

```
//@ assert P {x |--> e};
x = e;
//@ assert P;
```

Listing 2.4: The assignment rule

For example,

if	`P`	is	`x > 0 && a[2* x ] == 0` ,
then	`P {x ↦ y+1}`	is	`y+1 > 0 && a[2*(y+1)] == 0` .

Hence, we get Listing 2.5 as an example instance of the assignment rule. Note that parentheses are required in the index expression to get the correct `2*(y+1)` rather than the faulty `2*y+1`.

```
//@ assert y+1 > 0 && a[2*(y+1)] == 0;
x = y+1;
//@ assert x > 0 && a[2*x] == 0;
```

Listing 2.5: An assignment rule example instance

Note that several different expressions `P` may result in the same expression `P {x |--> e}`. For example, all four expressions

```
  x > 0 && a[2*  x  ] == 0 ,
  x > 0 && a[2*(y+1)] == 0 ,
y+1 > 0 && a[2*  x  ] == 0 ,
```

and `y+1 > 0 && a[2*(y+1)] == 0` ,

result in `y+1 > 0 && a[2*(y+1)] == 0`
after applying `{x |--> y+1}`.

For this reason, the same precondition and statement may result in several different postconditions (All four above expressions are valid postconditions in Listing 2.5, for example). However, given a postcondition and a statement, there is only one precondition that corresponds.

When first confronted with Hoare's assignment rule, most people are tempted to think of a simpler and more intuitive alternative, shown in Listing 2.6.

```
//@ assert P;
x = e;
//@assert P && x ==
   e;
```

Listing 2.6: Simpler, but *faulty* assignment rule

Listing 2.7–2.9 show some example instances of this faulty rule.

```
//@ assert y > 0;
x = y+1;
//@ assert y > 0 && x == y+1;
```

Listing 2.7: An example instance of the faulty rule from Listing 2.6

While Listing 2.7 happens to be ok, Listing 2.8 and 2.9 lead to postconditions that are obviously nonsensical formulas.

```
//@ assert true;
x = x+1;
//@assert x == x+1;
```

Listing 2.8: An example instance of the faulty rule from Listing 2.6

The reason is that in the assignment in Listing 2.8 the left-hand side variable `x` also appears in the right-hand side expression `e`, while the assignment in Listing 2.9 just destroys the property from its precondition.

```
//@ assert x < 0;
x = 5;
//@ assert x < 0 && x == 5;
```

Listing 2.9: An example instance of the faulty rule from Listing 2.6

Note that the correct example Listing 2.7 can as well be obtained as an instance of the correct rule from Listing 2.4, since replacing `x` by `y+1` in its postcondition yields `y > 0 && y+1 == y+1` as precondition, which is logically equivalent to just `y > 0`.

2.2. The sequence rule

The sequence rule, shown in Listing 2.10, combines two code fragments `Q` and `S` into a single one `Q ; S`. Note that the postcondition for `Q` must be identical to the precondition of `S`. This just reflects the sequential execution ("first do `Q`, then do `S`") on a formal level. Thanks to this rule, we may "annotate" a program with interspersed formulas, as it is done in Frama-C.

```
//@ assert P
   ;
Q;
//@ assert R
   ;
```

and

```
//@ assert R
   ;
S;
//@ assert T
   ;
```

⇝

```
//@ assert P
   ;
Q ; S;
//@ assert T
   ;
```

Listing 2.10: The sequence rule

2.3. The implication rule

The implication rule, shown in Listing 2.11, allows us at any time to weaken a postcondition and to sharpen a precondition. We will provide application examples together with the next rule.

```
//@ assert P
   ;
Q;
//@ assert R
   ;
```

⇝

```
//@ assert P`;
Q;
//@ assert R`;
```

if `P` ==> P`
and `R ==> R``

Listing 2.11: The implication rule

2.4. The choice rule

The choice rule, depicted in Listing 2.12, is needed to verify `if(...){...} else{...}` statements. Both branches must establish the same postcondition, viz. `S`. The implication rule can be used to weaken differing postconditions `S1` of a `then` branch and `S2` of an `else` branch into a unified postcondition `S1||S2`, if necessary. In each branch, we may use what we know about the condition `B`, e.g. in the `else` branch, that it is false. If the `else` branch is missing, it can be considered as consisting of an empty sequence, having the postcondition `P && !B`.

Listing 2.13 shows an example application of the choice rule. The variable `i` may be used as an index into a ring buffer `int a[n]`. The shown code fragment just advances the index `i` appropriately. We verified that `i` remains a valid index into `a[]` provided it was valid before. Note the use

```
//@ assert P &&
    B;
Q;
//@ assert S;
```

and

```
//@ assert P && !
    B;
R;
//@ assert S;
```

⇝

```
//@ assert
   P;
if (B) {
    Q;
} else {
    R;
}
//@ assert
   S;
```

Listing 2.12: The choice rule

of the implication rule to establish preconditions for the assignment rule as needed, and to unify the postconditions of the `then` and **`else`** branch, as required by the choice rule.

`//@ assert 0 <= i < n`	given precondition
`if (i < n-1){`	
`  //@ assert 0 <= i < n - 1;`	using that the condition `i < n-1;` holds in the
`  //@ assert 1 <= i+1 < n;`	`then` part by the implication rule
`  i = i+1;`	
`  //@ assert 1 <= i < n;`	by the assignment rule
`  //@ assert 0 <= i < n;`	weakened by the implication rule
`} else {`	
`  //@ assert 0 <= i == n-1 < n;`	using that then condition `i < n-1` fails in the
`  //@ assert 0 == 0 && 0 < n;`	**`else`** part weakened by the implication rule
`  i = 0;`	
`  //@ assert i == 0 && 0 < n;`	by the assignment rule
`  //@ assert 0 <= i < n;`	weakened by the implication rule
`}`	
`//@ assert 0 <= i < n;`	by the choice rule from the `then` and the **`else`** part

Listing 2.13: An example application of the choice rule

2.5. The loop rule

The loop rule, shown in Listing 2.14, is used to verify a **while** loop. This requires to find an appropriate formula, `P`, which is preserved by each execution of the loop body. `P` is also called a loop invariant.

```
//@ assert P &&
      B;
S;
//@ assert P;
```

$\rightsquigarrow$

```
//@ assert P;
while (B)
{
     S;
}
//@ assert ! B && P
    ;
```

Listing 2.14: The loop rule

To find it requires some intuition in many cases; for this reason, automatic theorem provers usually have problems with this task.

As said above, the loop rule does not guarantee that the loop will always eventually terminate. It merely assures us that, if the loop has terminated, the postcondition holds. To emphasis this, the properties verifiable with the Hoare Calculus are usually called "partial correctness" properties, while properties that include program termination are called "total correctness" properties.

As an example application, let us consider an abstract ring-buffer loop as shown in Listing 2.15.

`//@ assert 0 < n;`	given precondition
`//@ assert 0 <= 0 < n;`	follows trivially
`int i = 0;`	
`//@ assert 0 <= i < n;`	by the assignment rule
`while(!done){`	
`  //@ assert 0 <= i < n && !done;`	may be assumed by the loop rule
`  a[i] = getchar();`	
`  //@ assert 0 <= i < n && !done;`	required property of `getchar`
`  //@ assert 0 <= i < n;`	weakened by the implication rule
`  if (i < n-1)i++; else i = 0;`	
`  //@ assert 0 <= i < n;`	as seen above (Listing 2.13)
`  process(a, i, &done);`	
`  //@ assert 0 <= i < n;`	required property of `process`
`}`	
`//@ assert 0 <= i < n;`	by the loop rule

Listing 2.15: An abstract ring buffer loop

Listing 2.15 shows a verification proof for the index `i` lying always within the valid range `[0..n-1]` during, and after, the loop. It uses the proof from Listing 2.13 as a sub-part. Note the following issues:

- To reuse the proof from Listing 2.13, we had to drop the conjunct `!done`, since we didn't consider it in Listing 2.13. In general, we may *not* infer

`//@ assert P && S;`
`Q;`
`//@ assert R && S;`
	from	`//@ assert P;`
		`Q;`
		`//@ assert R;`

  since the code fragment `Q` may just destroy the property `S`. This is obvious for `Q` being the fragment from Listing 2.13, and `S` being e.g. `i != 0`.

  Suppose for a moment that `process` had been implemented in a way such that it refuses to set `done` to **`true`** unless it is **`false`** at entry. In this case, we would really need that `!done` still holds after execution of Listing 2.13. We would have to do the proof again, looping-through an additional conjunct `!done`.

- We have similar problems to carry the property `0 <= i < n && !done` and `0 <= i < n` over the statement `a[i] = getchar()` and `process(a, i, &done)`, respectively. We need to specify that neither `getchar` nor `process` is allowed to alter the value of `i` or `n`. In ACSL, there is a particular language construct `assigns` for that purpose, which is introduced in Section 6.1 on Page 70.

- In our example, the loop invariant can be established between any two statements of the loop body. However, this need not be the case in general. The loop rule only requires the invariant holds before the loop and at the end of the loop body. For example, `process` could well change the value of `i`[8] and even `n` intermediately, as long as it re-establishes the property `0 <= i < n` immediately prior to returning.

- The loop invariant, `0 <= i < n`, is established by the proof in Listing 2.13 also after termination of the loop. Thus, e.g., a final `a[i] = '\0'` after the loop would be guaranteed not to lead to a bounds violation.

- Even if we would need the property `0 <= i < n` to hold only immediately before the assignment `a[i] = getchar()`, since, e.g., `process`'s body didn't use `a` or `i`, we would still have to establish `0 <= i < n` as a loop invariant by the loop rule, since there is no other way to obtain any property inside a loop body. Apart from this formal reason it is obvious that `0 <= i < n` wouldn't hold during the second loop iteration unless we re-established it at the end of the first one, and that is just what the while rule requires.

[8] We would have to change the call to `process(a, &i, &done)` and the implementation of `process` appropriately. In this case we couldn't rely on the above-mentioned `assigns` clause for `process`.

2.6. Derived rules

The above rules don't cover all kinds of statements allowed in C. However, missing C-statements can be rewritten into a form that is semantically equivalent and covered by the Hoare rules.

For example,

```
switch (E) {
    case E1: Q1; break; ...
    case En: Qn; break;
    default: Q0; break;
}
```

is semantically equivalent to

```
if (E == E1) {
    Q1;
} else ... if (E == En) {
    Qn;
} else {
    Q0;
}
```

if `E` doesn't have side-effects. While the `if-else` form is usually slower in terms of execution speed on a real computer, this doesn't matter for verification purposes, which are separate from execution issues.

Similarly, `for (P; Q; R) {S}` can be re-expressed as `P; while (Q) {S ; R}`, and so on.

It is then possible to derive a Hoare rule for each kind of statement not previously discussed, by applying the classical rules to the corresponding re-expressed code fragment. However, we do not present these derived rules here.

Although procedures cannot be re-expressed in the above way if they are (directly or mutually) recursive, it is still possible to derive Hoare rules for them. This requires the finding of appropriate "procedure invariants" similar to loop invariants. Non-recursive procedures can, of course, just be inlined to make the classical Hoare rules applicable.

Note that `goto` cannot be rewritten in the above way; in fact, programs containing `goto` statements cannot be verified with the Hoare Calculus. See [12] for a similar calculus that can deal with arbitrary flowcharts, and hence arbitrary jumps. In fact, Hoare's work was based on that calculus. Later calculi inspired from Hoare's work have been designed to re-integrate support for arbitrary jumps. However, in this tutorial, we will not discuss example programs containing a `goto`.

3. Non-mutating algorithms

In this chapter, we consider *non-mutating* algorithms, i.e., algorithms that neither change their arguments nor any objects outside their scope. This requirement can be formally expressed with the following *assigns clause*:

```
assigns \nothing;
```

Each algorithm in this chapter therefore uses this assigns clause in its specification.

The specifications of these algorithms are not very complex. Nevertheless, we have tried to arrange them so that the earlier examples are simpler than the later ones. All algorithms work on one-dimensional arrays ("ranges").

`equal` (Section 3.1 on Page 26) compares two ranges element-by-element. Here, we will present to versions to specify to specify such a function.

`mismatch` (Section 3.2 on Page 31) returns the smallest index where two ranges differ. An implementation of `equal` using `mismatch` is also presented.

`find` (Section 3.3 on Page 33) provides *sequential* or *linear search* and returns the smallest index at which a given value occurs in a range. In Section 3.4, on Page 35, a predicate is introduced in order to simplify the specification.

`find_first_of` (Section 3.5, on Page 37) provides similar to `find` a *sequential search*, but unlike `find` it does not search for a particular value, but for the least index of a given range which occurs in another range.

`adjacent_find` (Section 3.6 on Page 39) can be used to find equal neighbors in an array.

`search` (Section 3.7, on Page 41) finds a subsequence that is identical to a given sequence when compared element-by-element and returns the position of the first occurrence.

`count` (Section 3.8, on Page 44) returns the number of occurrences of a given value in a range. Here we will employ some user-defined axioms to formally specify `count`.

3.1. The `equal` algorithm

The `equal` algorithm in the C++ Standard Library compares two generic sequences. For our purposes we have modified the generic implementation[9] to that of an array of type `value_type`. The signature now reads:

```
bool equal(const value_type* a, size_type n, const value_type* b);
```

The function returns **true** if `a[i] == b[i]` holds for each `0 <= i < n`. Otherwise, `equal` returns **false**.

3.1.1. Formal specification of `equal`

The ACSL specification of `equal` is shown in Listing 3.1. We discuss the specification now line by line.

```
/*@
  requires \valid_read(a + (0..n-1));
  requires \valid_read(b + (0..n-1));

  assigns \nothing;

  behavior all_equal:
    assumes \forall integer i; 0 <= i < n ==> a[i] == b[i];
    ensures \result;

  behavior some_not_equal:
    assumes \exists integer i; 0 <= i < n && a[i] != b[i];
    ensures !\result;

  complete behaviors;
  disjoint behaviors;
*/
bool equal(const value_type* a, size_type n, const value_type* b);
```

Listing 3.1: Formal specification of `equal`

The first part of our specification are the preconditions, which must be satisfied before the algorithm is executed. Those requirements can be specified with the `requires`-clause in ACSL. In case of the `equal` algorithm it is needed that `n` is non-negative (not specified) and that the pointers `a` and `b` point to n contiguously allocated objects of type `value_type` (see also Section 1.2).

In the second part of our specification we make a statement about objects and arguments that the function is allowed to change. Since `equal` is a non-mutating algorithm and does not modify any memory location outside its scope we just define `assigns \nothing` (see Page 25).

Finally, we define the postconditions, which must hold after the `equal` algorithm is finished. Corresponding to the informal description from the STL documentation, we have two behaviors:

[9]See `http://www.sgi.com/tech/stl/equal.html`.

The behavior `all_equal` applies if an element-wise comparison of the two ranges yields that they are all equal (this si formalized in the foirst `assumes` clause. In this case the function `equal` is expected to return **`true`**; we express this by “`ensures \result`”. The behavior `some_not_equal` applies if there is at least one valid index `i` where the elements `a[i]` and `b[i]` differ (second `assumes` clause). In this case the function `equal` is expected to return **`false`**, expressed as “`ensures !\result`”.

The negation of the formula

```
\forall integer i; 0 <= i < n ==> a[i] == b[i];
```

in behavior `all_equal` is just the formula

```
\exists integer i; 0 <= i < n && a[i] != b[i];
```

in behavior `some_not_equal`. Therefore, these two behaviors complement each other. Also note that the variable `i` is not of type **`int`**, but of type `integer`. While the former type comprises finitely many (often just 4294967296) distinct numbers available on the target platform hardware, the latter type contains numbers of arbitrary size, and is allowed only in ACSL specifications. Using type `integer` becomes a real issue e.g. in Sect. 7.1.

The `complete behaviors`-clause in Listing 3.1 expresses the fact that for all ranges `a` and `b` that satisfy the preconditions of the contract *at least one* of the specified named behaviors, in this case `all_equal` and `some_not_equal`, applies.

The `disjoint behaviors`-clause in Listing 3.1 formalizes the fact that for all ranges `a` and `b` that satisfy the preconditions of the contract *at most one* of the specified named behaviors, in this case `all_equal` and `some_not_equal`, applies.

3.1.2. The `EqualRanges` predicate

The fact that two arrays `a[0]..a[n-1]` and `b[0]..b[n-1]` are equal when compared element by element, is a property we might need again in other specifications, as it describes a very basic behavior.

The motto *don't repeat yourself* is not just good programming practice.[10] It is also true for concise and easy to understand specifications. We will therefore introduce specification elements that we can apply to the `equal` algorithm as well as to other specifications and implementations with the described behavior.

In Listing 3.2 we introduce the predicate `EqualRanges`.

This predicate formalizes the fact that the arrays `a[0]..a[n-1]` and `b[0]..b[n-1]` are equal when compared element by element. The letters `A` and `B` are *labels*[11] that must be supplied when using the predicate `EqualRanges`. We use labels in the definition of `EqualRanges` to extend its applicability. The expression `\at(a[i], A)` means that `a[i]` is evaluated at the label `A`. Frama-C defines several standard labels, e.g. `Old` and `Post`, a programmer can use to refer to the pre-state or post-state, respectively, of a function. For more details on labels we refer to the ACSL specification [9, p. 39].

[10]Compare `http://en.wikipedia.org/wiki/Don't_repeat_yourself`.

[11]Labels are used in C to name the target of the *goto* jump statement.

```
/*@
  predicate
    EqualRanges{A,B}(value_type* a, integer n, value_type* b) =
      \forall integer i; 0 <= i < n ==> \at(a[i], A) == \at(b[i], B);

  predicate
    EqualRanges{A,B}(value_type* a, integer n) =
      \forall integer i; 0 <= i < n ==> \at(a[i], A) == \at(a[i], B);
*/
```

Listing 3.2: The predicate `EqualRanges`

Using this predicate we can reformulate the specification of `equal` in Listing 3.1 as shown in Listing 3.3. Here we use the predefined label `Here`. When used in an `ensures` clause the label `Here` refers to the pre-state of a function.

```
/*@
  requires \valid_read(a + (0..n-1));
  requires \valid_read(b + (0..n-1));

  assigns \nothing;

  ensures \result <==> EqualRanges{Here,Here}(a, n, b);
*/
bool equal(const value_type* a, size_type n, const value_type* b);
```

Listing 3.3: Formal specification of `equal` using the `EqualRanges` predicate

Note that the equivalence is needed in the ensures clause. Putting an equality instead is not legal in ACSL, because `EqualRanges` is a predicate.

3.1.3. Implementation of `equal`

Listing 3.4 shows one way to implement the function `equal`. In our description, we concentrate on the *loop annotations*.

The first loop *invariant* is needed to prove that all accesses to `a` and `b` occur with valid indices. However, we may *not* require simply

```
  loop invariant 0 <= i < n;
```

since the very last loop iteration would violate this formula. Therefore, we have to weaken the formula to that shown in the implementation of Listing 3.4, which is preserved by *all* iterations of the loop. Note that `0 <= i < n` is still valid immediately before the array accesses in, since we may assume there in addition that the loop condition `i < n` holds. However, `0 <= i < n` is invalid after completion of the loop, while the loop invariant is guaranteed to hold there, too, cf. the loop rule in Figure 2.14 on Page 21.

Most important is the last loop *invariant*. It complies with the postcondition of the specification in Listing 3.3 and is needed to prove that for each iteration all elements of `a` and `b` up to that iteration

```
bool equal(const value_type* a, size_type n, const value_type* b)
{
  /*@
    loop invariant 0 <= i <= n;
    loop invariant \forall integer k; 0 <= k < i ==> a[k] == b[k];
    loop assigns i;
    loop variant n-i;
  */
  for (size_type i = 0; i < n; i++)
  {
    if (a[i] != b[i])
    {
      return false;
    }
  }

  return true;
}
```

Listing 3.4: Implementation of `equal`

step are equal. The loop assigns-clause in Listing 3.4 expresses that only the loop index is modified in any iteration. This is in accordance with the fact that `equal` is a *non-mutating* algorithm. The loop *variant* is needed to generate correct verification conditions for the termination of the **`for`**-loop. In order to prove the termination of the loop, Frama-C needs to know an expression whose value is decreased by each and every loop cycle and is always positive[12][9, Subsections 2.4.2, 2.5.1]. For a **`for`** loop as simple as that the expression `n-i` is sufficient for that purpose. Again, we can use the predicate `EqualRanges` in order to simplify the second loop invariant, which complies our postcondition. Listing 3.5 shows the modified implementation using the predicate `EqualRanges`.

[12]Except for possibly the very last iteration.

```
bool equal(const value_type* a, size_type n, const value_type* b)
{
  /*@
    loop invariant 0 <= i <= n;
    loop invariant EqualRanges{Here,Here}(a, i, b);
    loop assigns   i;
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i)
  {
    if (a[i] != b[i])
    {
      return false;
    }
  }

  return true;
}
```

Listing 3.5: Implementation of `equal` using the `EqualRanges` predicate

3.2. The mismatch algorithm

The `mismatch` algorithm is closely related to the negation of `equal` from Section 3.1. Its signature reads

```
size_type mismatch(const value_type* a, int n,
                   const value_type* b);
```

The function `mismatch` returns the smallest index where the two ranges `a` and `b` differ. If no such index exists, that is, if both ranges are equal then `mismatch` returns the length `n` of the two ranges.[13]

3.2.1. Formal specification of mismatch

We use the `EqualRanges` predicate defined in Listing 3.2 also for the formal specification of `mismatch` that is shown in Listing 3.6.

Note in particular the use of `EqualRanges` in the specification shown in Listing 3.6 in order to express that `mismatch` returns the *smallest* index where the two arrays differ. Note also that the completeness and disjointedness of the behaviors `all_equal` and `some_not_equal` has now become immediately obvious, since their `assumes` clauses are just literal negations of each other.

```
/*@
  requires \valid_read(a + (0..n-1));
  requires \valid_read(b + (0..n-1));

  assigns \nothing;

  behavior all_equal:
    assumes EqualRanges{Here,Here}(a, n, b);
    ensures \result == n;

  behavior some_not_equal:
    assumes !EqualRanges{Here,Here}(a, n, b);
    ensures 0 <= \result < n;
    ensures a[\result] != b[\result];
    ensures EqualRanges{Here,Here}(a, \result, b);

  complete behaviors;
  disjoint behaviors;
*/
size_type mismatch(const value_type* a, size_type n,
                   const value_type* b);
```

Listing 3.6: Formal specification of `mismatch`

[13] See also `http://www.sgi.com/tech/stl/mismatch.html`.

3.2.2. Implementation of mismatch

Listing 3.7 shows an implementation of mismatch that we have enriched with some loop annotations to support the deductive verification.

```
size_type mismatch(const value_type* a, size_type n,
                   const value_type* b)
{
  /*@
    loop invariant 0 <= i <= n;
    loop invariant EqualRanges{Here,Here}(a, i, b);
    loop assigns i;
    loop variant n-i;
  */
  for (size_type i = 0; i < n; i++)
    if (a[i] != b[i])
    {
      return i;
    }

  return n;
}
```

Listing 3.7: Implementation of mismatch

We use the predicate EqualRanges as shown in Listing 3.7 in order to express that all indices k that are less than the current index i satisfy the condition a[k] == b[k]. This is necessary to prove that mismatch indeed returns the smallest index where the two ranges differ.

3.2.3. Implementation of equal by calling mismatch

Listing 3.8 shows an implementation of the equal algorithm by a simple call of mismatch.[14]

```
/*@
  requires \valid_read(a + (0..n-1));
  requires \valid_read(b + (0..n-1));

  assigns \nothing;

  ensures \result <==> EqualRanges{Here,Here}(a, n, b);
*/
bool equal(const value_type* a, size_type n, const value_type* b);
```

Listing 3.8: Implementation of equal with mismatch

[14]See also the note on the relationship of equal and mismatch on http://www.sgi.com/tech/stl/equal.html.

3.3. The `find` algorithm

The `find` algorithm in the C++ standard library implements *sequential search* for general sequences.[15] We have modified the generic implementation, which relies heavily on C++ templates, to that of a range of type `value_type`. The signature now reads:

```
size_type find(const value_type* a, size_type n, value_type val);
```

The function `find` returns the least *valid* index `i` of `a` where the condition `a[i] == val` holds. If no such index exists then `find` returns the length `n` of the array.

3.3.1. Formal specification of `find`

The formal specification of `find` in ACSL is shown in Listing 3.9.

```
/*@
  requires  \valid_read(a + (0..n-1));

  assigns  \nothing;

  behavior some:
    assumes  \exists integer i; 0 <= i < n && a[i] == val;
    ensures  0 <= \result < n;
    ensures  a[\result] == val;
    ensures  \forall integer i; 0 <= i < \result ==> a[i] != val;

  behavior none:
    assumes  \forall integer i; 0 <= i < n ==> a[i] != val;
    ensures  \result == n;

  complete behaviors;
  disjoint behaviors;
*/
size_type find(const value_type* a, size_type n, value_type val);
```

Listing 3.9: Formal specification of `find`

The `requires`-clause indicates that `n` is non-negative and that the pointer `a` points to *n* contiguously allocated objects of type `value_type` (see Section 1.2).

The `assigns`-clause indicates that `find` (as a non-mutating algorithm), does not modify any memory location outside its scope (see Page 25).

We have subdivided the specification of `find` into two behaviors (named `some` and `none`). The behavior `some` applies if the sought-after value is contained in the array. We express this condition by using the `assumes`-clause. The next line expresses that if the assumptions of the behavior are satisfied then `find` will return a valid index. The algorithm also ensures that the returned (valid) index `i`, `a[i] == val` holds. Therefore we define this property in the second postcondition of

[15] See `http://www.sgi.com/tech/stl/find.html`.

behavior `some`. Finally, it is important to express that `find` return the smallest index `i` for which `a[i] == val` holds (see last postcondition of behavior `some`).

The behavior `none` covers the case that the sought-after value is *not* contained in the array (see `assumes`-clause of behavior `none` in Listing 3.9). In this case, `find` must return the length `n` of the range `a`.

Note that the formula in the `assumes`-clause of the behavior `some` is the negation of the `assumes` -clause of the behavior `none`. Therefore, we can express that these two behaviors are *complete* and *disjoint*.

3.3.2. Implementation of **find**

Listing 3.10 shows a straightforward implementation of `find`. The only noteworthy elements of this implementation are the *loop annotations*.

```
size_type find(const value_type* a, size_type n, value_type val)
{
  /*@
    loop invariant 0 <= i <= n;
    loop invariant \forall integer k; 0 <= k < i ==> a[k] != val;
    loop assigns i;
    loop variant n-i;
   */
  for (size_type i = 0; i < n; i++)
    if (a[i] == val)
    {
      return i;
    }

  return n;
}
```

Listing 3.10: Implementation of `find`

The first loop *invariant* is needed to prove that accesses to `a` only occur with valid indices. With the second loop *invariant* is needed for the proof of the postconditions of the behavior `some` (see Listing 3.9). It expresses that for each iteration the sought-after value is not yet found up to that iteration step.

Finally, the loop *variant* `n-i` is needed to generate correct verification conditions for the termination of the loop.

3.4. The `find` algorithm reconsidered

In this section we specify the `find` algorithm in a slightly different way when compared to Section 3.3. Our approach is motivated by a considerable number of closely related formulas. We have in Listings 3.9 and 3.10 the following formulas

`\exists integer i; 0 <= i < n && a[i] == val;`
`\forall integer i; 0 <= i < \result ==> a[i] != val;`
`\forall integer i; 0 <= i < n ==> a[i] != val;`
`\forall integer k; 0 <= k < i ==> a[k] != val;`

Note that the first formula is the negation of the third one.

In order to be more explicit about the commonalities of these formulas we define a predicate, called `HasValue` (see Listing 3.11), which describes the situation that there is a valid index `i` such that

`a[i] == val`

holds.

```
/*@
  predicate
    HasValue{A}(value_type* a, integer n, value_type val) =
      \exists integer i; 0 <= i < n && a[i] == val;
*/
```

Listing 3.11: The predicate `HasValue`

Note that we needed to provide a label, viz. `A`, to the predicate, since its evaluation depends on a memory state, viz. then contents of `a[]`. ACSL allows to abbreviate `\at(a[i],A)` by `a[i]` if, as in our predicate body, `A` is the only available label.

With this predicate we can encapsulate all uses of the ∀ and ∃ quantifiers in both the specification of the function contract of `find` and in the loop annotations. The result is shown in Listings 3.12 and 3.13.

3.4.1. Formal specification of `find`

This approach leads to a specification of `find` that is more readable than the one described in Section 3.3.

In particular, it can be seen immediately that the conditions in the assumes clauses of the two behaviors `some` and `none` are mutually exclusive since one is the literal negation of the other. Moreover, the requirement that `find` returns the smallest index can also be expressed using the `HasValue` predicate, as depicted with the last postcondition of behavior `some` as shown in Listing 3.12.

```
/*@
  requires  \valid_read(a + (0..n-1));

  assigns  \nothing;

  behavior some:
    assumes  HasValue(a, n, val);
    ensures  0 <= \result < n;
    ensures  a[\result] == val;
    ensures  !HasValue(a, \result, val);

  behavior none:
    assumes  !HasValue(a, n, val);
    ensures  \result == n;

  complete behaviors;
  disjoint behaviors;
*/
size_type find(const value_type* a, size_type n, value_type val);
```

Listing 3.12: Formal specification of `find` using the `HasValue` predicate

3.4.2. Implementation of `find`

The predicate `HasValue` is also used in the loop annotation inside the implementation of `find`.

```
size_type find(const value_type* a, size_type n, value_type val)
{
  /*@
    loop invariant 0 <= i <= n;
    loop invariant !HasValue(a, i, val);
    loop assigns i;
    loop variant n-i;
   */
  for (size_type i = 0; i < n; i++)
    if (a[i] == val)
    {
      return i;
    }

  return n;
}
```

Listing 3.13: Implementation of `find` with loop annotations based on `HasValue`

3.5. The find_first_of algorithm

The find_first_of algorithm[16] is closely related to find (see Sections 3.3 and 3.4).

```
size_type find_first_of(const value_type* a, size_type m,
                        const value_type* b, size_type n);
```

As in find it performs a sequential search. However, whereas find searches for a particular value, find_first_of returns the least index i such that a[i] is equal to one of the values b[0..n-1].

3.5.1. Formal specification of find_first_of

Similar to our approach in Section 3.4, we define a predicate HasValueOf that formalizes the fact that there are valid indices i for a and j of b such that a[i] == b[j] hold. We have chosen to reuse the predicate HasValue (Listing 3.11) to define HasValueOf (Listing 3.14).

```
/*@
  predicate
  HasValueOf{A}(value_type* a, integer m, value_type* b, integer n) =
    \exists integer i; 0 <= i < m && HasValue{A}(b, n, a[i]);
*/
```

Listing 3.14: The predicate HasValueOf

Both the predicates HasValueOf and HasValue occur in the formal specification of find_first_of (see Listing 3.15). Note how similar the specification of find_first_of becomes to that of find (Listing 3.12) when using these predicates.

3.5.2. Implementation of find_first_of

Our implementation of find_first_of is shown in Listing 3.16.

Note the call of the find function shown in the Listing above. In the original implementation[17], find_first_of does not call find but rather inlines it. The reason for this were probably efficiency considerations. We opted for an implementation of find_first_of that emphasizes reuse. Besides, leading to a more concise implementation, we also have to write less loop annotations.

[16] See http://www.sgi.com/tech/stl/find_first_of.html.
[17] See http://www.sgi.com/tech/stl/stl_algo.h

```
/*@
  requires \valid_read(a + (0..m-1));
  requires \valid_read(b + (0..n-1));

  assigns \nothing;

  behavior found:
    assumes HasValueOf(a, m, b, n);
    ensures bound:  0 <= \result < m;
    ensures result: HasValue(b, n, a[\result]);
    ensures first:  !HasValueOf(a, \result, b, n);

  behavior not_found:
    assumes !HasValueOf(a, m, b, n);
    ensures result: \result == m;

  complete behaviors;
  disjoint behaviors;
*/
size_type find_first_of(const value_type* a, size_type m,
                        const value_type* b, size_type n);
```

Listing 3.15: Formal specification of `find_first_of`

```
size_type find_first_of (const value_type* a, size_type m,
                         const value_type* b, size_type n)
{
  /*@
    loop invariant bound: 0 <= i <= m;
    loop invariant not_found: !HasValueOf(a, i, b, n);
    loop assigns i;
    loop variant m-i;
  */
  for (size_type i = 0; i < m; i++)
  {
    if (find(b, n, a[i]) < n)
    {
      return i;
    }
  }

  return m;
}
```

Listing 3.16: Implementation of `find_first_of`

3.6. The `adjacent_find` algorithm

The `adjacent_find` algorithm[18]

```
size_type adjacent_find(const value_type* a, size_type n);
```

returns the smallest valid index i, such that i+1 is also a valid index and such that

```
a[i] == a[i+1]
```

holds. The `adjacent_find` algorithm returns `n` if no such index exists.

3.6.1. Formal specification of `adjacent_find`

As in the case of other search algorithms, we first define a predicate `HasEqualNeighbors` (see Listing 3.17) that captures the essence of finding two adjacent indices at which the array holds equal values.

```
/*@
  predicate
    HasEqualNeighbors{A}(value_type* a, integer n) =
      \exists integer i; 0 <= i < n-1 && a[i] == a[i+1];
*/
```

Listing 3.17: The predicate `HasEqualNeighbors`

```
/*@
  requires \valid_read(a + (0..n-1));

  assigns \nothing;

  behavior some:
    assumes  HasEqualNeighbors(a, n);
    ensures  0 <= \result < n-1;
    ensures  a[\result] == a[\result+1];
    ensures  !HasEqualNeighbors(a, \result);

  behavior none:
    assumes  !HasEqualNeighbors(a, n);
    ensures  \result == n;

  complete behaviors;
  disjoint behaviors;
*/
size_type adjacent_find(const value_type* a, size_type n);
```

Listing 3.18: Formal specification of `adjacent_find`

[18] See `http://www.sgi.com/tech/stl/adjacent_find.html`

We use the predicate HasEqualNeighbors to define the formal specification of adjacent_find (see Listing 3.18).

3.6.2. Implementation of adjacent_find

The implementation of adjacent_find, including loop (in)variants is shown in Listing 3.19. Please note the use of the predicate HasEqualNeighbors in the loop invariant to match the similar postcondition of behavior some.

```
size_type
adjacent_find(const value_type* a, size_type n)
{
  if (n == 0) return n;

  /*@
    loop invariant 0 <= i < n;
    loop invariant !HasEqualNeighbors(a, i+1);
    loop assigns i;
    loop variant n-i;
  */
  for (size_type i = 0; i < n - 1; i++)
  {
    if (a[i] == a[i + 1])
    {
      return i;
    }
  }

  return n;
}
```

Listing 3.19: Implementation of adjacent_find

3.7. The `search` algorithm

The `search` algorithm in the C++ standard library finds a subsequence that is identical to a given sequence when compared element-by-element. For our purposes we have modified the generic implementation to that of an array of type `value_type`.[19] The signature now reads:

```
size_type search(const value_type* a, size_type m,
                 const value_type* b, size_type n)
```

The function `search` returns the first index `i` of the array `a` where the condition `a[i+k]==b[k]` for each `0 <= k < n` holds. If no such index exists then `search` returns the length `m` of the array `a`. Figure 3.20 tries to illustrate the requirement of `search` that `b[0..n-1]` cannot be found in the subrange `a[0..\result+n-2]`.

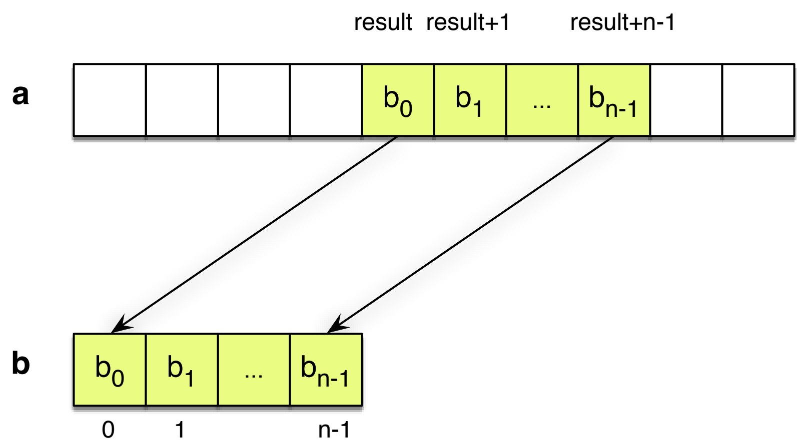


Figure 3.20.: Matching `b[0..n-1]` in `a[0..m-1]`

3.7.1. Formal specification of `search`

Our specification of `search` starts with introducing the predicate `HasSubRange` in Listing 3.21. This predicate formalizes the fact that the sequence `a` contains a subsequence which is identical to the sequence `b`.

The ACSL specification of `search` is shown in Listing 3.22. The behavior `has_match` applies if the sequence `a` contains a subsequence, which is identical to the sequence `b`. We express this condition with `assumes` by using the predicate `HasSubRange`.

The first `ensures` clause of behavior `has_match` indicates that the return value must be in the range `[0..m-n]`. The second one expresses that `search` returns the smallest index where `b` can be found in `a`. Finally, in the last line under behavior `has_match` we indicate that the sequence `a` contains a subsequence (from the position `\result`), which is identical to the sequence `b`.

[19]See `http://www.sgi.com/tech/stl/search.html`.

```
/*@
  predicate
    HasSubRange{A}(value_type* a, integer m,
                   value_type* b, integer n) =
      \exists size_type k;
         (0 <= k <= m-n) && EqualRanges{A,A}(a+k, n, b);
*/
```

Listing 3.21: The predicate HasSubRange

```
/*@
  requires \valid_read(a + (0..m-1));
  requires \valid_read(b + (0..n-1));

  assigns \nothing;

  ensures (n == 0 || m == 0) ==> \result == 0;

  behavior has_match:
    assumes HasSubRange(a, m, b, n);
    ensures 0 <= \result <= m-n;
    ensures EqualRanges{Here,Here}(a+\result, n, b);
    ensures !HasSubRange(a, \result+n-1, b, n);

  behavior no_match:
    assumes !HasSubRange(a, m, b, n);
    ensures \result == m;

  complete behaviors;
  disjoint behaviors;
*/
size_type search(const value_type* a, size_type m,
                 const value_type* b, size_type n);
```

Listing 3.22: Formal specification of search

The behavior no_match covers the case that there is no such subsequence in sequence a, which equals to the sequence b. In this case, search must return the length m of the range a. In any case, if the ranges a or b are empty, then the return value will be 0. We express this fact with the following line:

```
ensures (n == 0 || m == 0) ==> \result == 0;
```

The formula in the assumes clause of the behavior has_match is the negation of the assumes clause of the behavior no_match. Therefore, we can express that these two behaviors are *complete* and *disjoint*.

3.7.2. Implementation of `search`

Our implementation of `search` is shown in Listing 3.23. It follows the C++ standard library implementation in being easy to understand, but needing an order of magnitude of `m*n` operations. In contrast, the sophisticated algorithm from [13] needs only `m+n` operations.[20]

```
size_type search(const value_type* a, size_type m,
                 const value_type* b, size_type n)
{
  if ((n == 0) || (m == 0))
  {
    return 0;
  }

  if (n > m)
  {
    return m;
  }

  /*@
    loop invariant 0 <= i <= m-n+1;
    loop invariant !HasSubRange(a, n+i-1, b, n);
    loop assigns i;
    loop variant m-i;
  */
  for (size_type i = 0; i <= m - n; i++)
  {
    if (equal(a + i, n, b)) // Is there a match?
    {
      return i;
    }
  }

  return m;
}
```

Listing 3.23: Implementation of `search`

The second loop *invariant* is needed for the proof of the postconditions of the behavior `has_match` (see Listing 3.22). It expresses that for each iteration the subsequence, which equals to the sequence `b`, is not yet found up to that iteration step.

[20]This question has been also discussed by the C++ standardization committee, see `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3905.html`

3.8. The `count` algorithm

The `count` algorithm in the C++ standard library counts the frequency of occurrences for a particular element in a sequence. For our purposes we have modified the generic implementation[21] to that of arrays of type `value_type`. The signature now reads:

```
size_type count(const value_type* a, size_type n, value_type val);
```

Informally, the function returns the number of occurrences of `val` in the array `a`.

3.8.1. An axiomatic definition of counting

The specification of `count` will be fairly short because it employs the *logic function* `Count` whose (considerably) longer definition is given in Listing 3.24.[22] We will reuse this axiomatic definition of counting for the specification of other algorithms, e.g., `remove_copy` (Section 6.9).

```
/*@
  axiomatic CountAxiomatic
  {
    logic integer Count{L}(value_type* a, integer n, value_type v)
        reads a[0..n-1];

    axiom CountEmpty: \forall value_type *a, v, integer n;
        n <= 0 ==> Count(a, n, v) == 0;

    axiom CountOneHit: \forall value_type *a, v, integer n;
        (a[n] == v ==> Count(a, n + 1, v) == Count(a, n, v) + 1);

    axiom CountOneMiss: \forall value_type *a, v, integer n;
        (a[n] != v ==> Count(a, n + 1, v) == Count(a, n, v));

    axiom CountRead{L1,L2}: \forall value_type *a, v, integer n;
        EqualRanges{L1,L2}(a, n) ==>
             Count{L1}(a, n, v) == Count{L2}(a, n, v);
  }
*/
```

Listing 3.24: The logic function `Count`

The logic function `Count` in Listing 3.24 determines the number of occurrences of a value `v` in the index range `[0..n-1]` of an array of type `value_type`.

- The ACSL keyword `axiomatic` is used to gather the logic function `Count` and its defining axioms. Note that the interval bound `n` and the return value for `Count` are of type `integer`.
- Axiom `CountEmpty` covers the case of an empty range.

[21] See `http://www.sgi.com/tech/stl/count.html`.
[22] This definition of `Count` is a generalization of the *logic function* `nb_occ` of the ACSL specification [9].

- Axioms `CountOneHit` and `CountOneMiss` reduce counting of a range of length $n+1$ to a range of length n.

- The `reads` clause in the axiomatic definition of `Count` specifies the set of memory locations on which `Count` depends. Specifically, it states that `Count` only depends on the range `a[0..n-1]`. Axiom `CountRead` then ensures that `Count` produces the same result if the values `a[0..n-1]` do not change between two program states indicated by the labels `L1` and `L2`, respectively. We use predicate `EqualRanges` (Listing 3.2) to express this condition. Axiom `CountRead` is necessary if one has to verify *mutating* algorithms that are specified with `Count`, e.g., `remove_copy` in Sections 6.9 and 6.10.

The following properties of `Count` can be verified with the help of the axioms given in Listing 3.24.

```
/*@
  lemma CountOne: \forall value_type *a, v, integer n;
     Count(a, n + 1, v) == Count(a, n, v) + Count(a + n, 1, v);

  lemma CountUnion: \forall value_type *a, v, integer n, k;
     0 <= k ==>
     Count(a, n + k, v) == Count(a, n, v) + Count(a + n, k, v);

  lemma CountBounds: \forall value_type *a, v, integer n;
      0 <= n ==> 0 <= Count(a, n, v) <= n;

  lemma CountMonotonic: \forall value_type *a, v, integer m, n;
      0 <= m <= n ==> Count(a, m, v) <= Count(a, n, v);
*/
```

Listing 3.25: More properties of `Count`

3.8.2. Formal specification of count

Listing 3.26 shows how we use the logic function count to specify `count` in ACSL. Note that our specification also states that the result of `count` is non-negative and less than or equal the says of the array.

```
/*@
  requires \valid_read(a + (0..n-1));

  assigns \nothing;

  ensures \result == Count(a, n, val);
  ensures 0 <= \result <= n;
*/
size_type count(const value_type* a, size_type n, value_type val);
```

Listing 3.26: Formal specification of `count`

3.8.3. Implementation of count

Listing 3.27 shows a possible implementation of `count`. Note that we refer to the logic function `Count` in one of the loop invariants.

```
size_type
count(const value_type* a, size_type n, value_type val)
{
  size_type counted = 0;

  /*@
    loop invariant 0 <= i <= n;
    loop invariant 0 <= counted <= i;
    loop invariant counted == Count(a, i, val);
    loop assigns i, counted;
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i)
  {
    if (a[i] == val)
    {
      counted++;
    }
  }

  return counted;
}
```

Listing 3.27: Implementation of `count`

4. Maximum and minimum algorithms

In this chapter we discuss the formal specification of algorithms that compute the maximum or minimum values of their arguments. As the algorithms in Chapter 3, they also do not modify any memory locations outside their scope. The most important new feature of the algorithms in this chapter is that they compare values using binary operators such as `<`.

We consider in this chapter the following algorithms.

`max_element` (Section 4.2, on Page 50) returns an index to a maximum element in range. Similar to `find` it also returns the smallest of all possible indices. An alternative specification which relies on user-defined predicates will be introduced in Section 4.3, on Page 52).

`max_seq` (Section 4.4, on Page 54) is very similar to `max_element` and will serve as an example of *modular verification*. It returns the maximum value itself rather than an index to it.

`min_element` which can be used to find the smallest element in an array (Section 4.5).

First, however, we discuss in Section 4.1 general properties that must be satisfied by the relational operators.

4.1. A note on relational operators

Note that in order to compare values, the algorithms in the C++ standard library usually rely solely on the *less than* operator < or special function objects.[23] To be precises, the operator < must be a *partial order*,[24] which means that the following rules hold.

$$
\begin{array}{lll}
\text{irreflexivity} & \forall x & : \neg(x < x) \\
\text{asymmetry} & \forall x, y & : x < y \implies \neg(y < x) \\
\text{transitivity} & \forall x, y, z & : x < y \wedge y < z \implies x < z
\end{array}
$$

If you wish check that the operator < of our `value_type`[25] satisfies this properties you can formulate lemmas in ACSL and verify them with Frama-C. (see Listing 4.1).

```
/*@
  lemma LessIrreflexivity:
    \forall value_type a; !(a < a);

  lemma LessAntisymmetry:
    \forall value_type a, b; (a < b) ==> !(b < a);

  lemma LessTransitivity:
    \forall value_type a, b, c; (a < b) && (b < c) ==> (a < c);
*/
```

Listing 4.1: Requirements for a partial order on `value_type`

It is of course possible to specify and implement the algorithms of this chapter by only using operator <. For example, `a <= b` can be written as `a < b || a == b`, or, for our particular ordering on `value_type`, as `!(b < a)`. However, for the purpose of this introductory document we have opted for a more user friendly representation.

Listing 4.2 formulates condition on the semantics of the derived operator >, <= and >=.

```
/*@
  lemma Greater:
    \forall value_type a, b; (a > b) <==> (b < a);

  lemma LessOrEqual:
    \forall value_type a, b; (a <= b) <==> !(b < a);

  lemma GreaterOrEqual:
    \forall value_type a, b; (a >= b) <==> !(a < b);
*/
```

Listing 4.2: Semantics of derived comparison operators

[23]See `http://www.sgi.com/tech/stl/LessThanComparable.html`.
[24]See `http://en.wikipedia.org/wiki/Partially_ordered_set`
[25]See Section 1.2

We also provide a group of predicates that concisely express the comparison of the elements in an array segment with a given value (see Listing 4.3). We will use these predicates both in this chapter and in Chapter binary-search.

```
/*@
  predicate ConstantRange(value_type* a, integer first,
                          integer last, value_type val) =
     \forall integer i; first <= i < last ==> a[i] == val;

  predicate StrictLowerBound(value_type* a, integer first,
                             integer last, value_type val) =
     \forall integer i; first <= i < last ==> val < a[i];

  predicate LowerBound(value_type* a, integer first,
                       integer last, value_type val) =
     \forall integer i; first <= i < last ==> !(a[i] < val);

  predicate StrictUpperBound(value_type* a, integer first,
                             integer last, value_type val) =
     \forall integer i; first <= i < last ==> a[i] < val;

  predicate UpperBound(value_type* a, integer first,
                       integer last, value_type val) =
     \forall integer i; first <= i < last ==> !(val < a[i]);
*/
```

Listing 4.3: Predicates for comparing array elements with a given value

4.2. The `max_element` algorithm

The `max_element` algorithm in the C++ Standard Template Library[26] searches the maximum of a general sequence. The signature of our version of `max_element` reads:

```
size_type max_element(const value_type* a, size_type n);
```

The function finds the largest element in the range `a[0, n)`. More precisely, it returns the unique valid index `i` such that

1. for each index `k` with `0 <= k < n` the condition `a[k] <= a[i]` holds and
2. for each index `k` with `0 <= k < i` the condition `a[k] < a[i]` holds.

The return value of `max_element` is `n` if and only if there is no maximum, which can only occur if `n == 0`.

4.2.1. Formal specification of `max_element`

A formal specification of `max_element` in ACSL is shown in Listing 4.4.

```
/*@
  requires  \valid_read(a + (..n-1));

  assigns \nothing;

  behavior empty:
    assumes n == 0;
    ensures \result == 0;

  behavior not_empty:
    assumes 0 < n;

    ensures 0 <= \result < n;
    ensures \forall integer i; 0 <= i < n ==> a[i] <= a[\result];
    ensures \forall integer i; 0 <= i < \result ==> a[i] < a[\result];

  complete behaviors;
  disjoint behaviors;
*/
size_type max_element(const value_type* a, size_type n);
```

Listing 4.4: Formal specification of `max_element`

We have subdivided the specification of `max_element` into two behaviors (named `empty` and `not_empty`). The behavior `empty` contains the specification for the case that the range contains no elements. The behavior `not_empty` applies if the range has a positive length.

[26] See `http://www.sgi.com/tech/stl/max_element.html`

The second `ensures` clause of behavior `not_empty` indicates that the returned valid index `k` refers to a maximum value of the array. The third one expresses that `k` is indeed the *first* occurrence of a maximum value in the array.

4.2.2. Implementation of `max_element`

Listing 4.5 shows an implementation of `max_element`. In our description, we concentrate on the *loop annotations*.

```
size_type max_element(const value_type* a, size_type n)
{
  if (n == 0)
  {
    return 0;
  }

  size_type max = 0;

  /*@
    loop invariant 0 <= i <= n;
    loop invariant 0 <= max <  n;
    loop invariant \forall integer k; 0 <= k < i   ==> a[k] <= a[max];
    loop invariant \forall integer k; 0 <= k < max ==> a[k] <  a[max];
    loop assigns max, i;
    loop variant n-i;
  */
  for (size_type i = 1; i < n; i++)
  {
    if (a[max] < a[i])
    {
      max = i;
    }
  }

  return max;
}
```

Listing 4.5: Implementation of `max_element`

The second loop invariant is needed to prove the first postcondition of behavior `not_empty` in Listing 4.4. Using the next loop invariant we prove the second postcondition of behavior `not_empty` in Listing 4.4. Finally, the last postcondition of this behavior can be proved with the endmost loop *invariant*.

4.3. The `max_element` algorithm with predicates

In this section we present another specification of the `max_element` algorithm. The main difference is that we employ two user defined predicates. First we define the predicate `MaxElement` by using the previously introduced predicate `UpperBound` (Listing 4.3) by stating that it is an upper bound that belongs to the sequence $a[0..n-1]$.

```
/*@
  predicate MaxElement{L}(value_type* a, integer n, integer max) =
      0 <= max < n && UpperBound(a, 0, n, a[max]);
*/
```

Listing 4.6: Definition of the `MaxElement` predicate

4.3.1. Formal specification of `max_element`

The new formal specification of `max_element` in ACSL is shown in Listing 4.7. Note that we also use the predicate `StrictUpperBound` (Listing 4.3) in order to express that `max_element` returns the *first* maximum position in $[0..n-1]$.

```
/*@
  requires  \valid_read(a + (0..n-1));

  assigns \nothing;

  behavior empty:
    assumes n == 0;

    ensures result:   \result == 0;

  behavior not_empty:
    assumes 0 < n;

    ensures result:   0 <= \result < n;
    ensures maximum:  MaxElement(a, n, \result);
    ensures first:    StrictUpperBound(a, 0, \result, a[\result]);

  complete behaviors;
  disjoint behaviors;
*/
size_type max_element(const value_type* a, size_type n);
```

Listing 4.7: Formal specification of `max_element`

4.3.2. Implementation of `max_element`

Listing 4.8 shows implementation of `max_element` with rewritten loop invariants. In the loop invariants we also employ the predicates `UpperBound` and `StrictUpperBound` that we have used in the specification.

```
size_type max_element(const value_type* a, size_type n)
{
  if (n == 0)
  {
    return 0;
  }

  size_type max = 0;

  /*@
    loop invariant bound:  0 <= i <= n;
    loop invariant min:    0 <= max < n;
    loop invariant lower:  UpperBound(a, 0, i, a[max]);
    loop invariant first:  StrictUpperBound(a, 0, max, a[max]);

    loop assigns max, i;
    loop variant n-i;
  */
  for (size_type i = 0; i < n; i++)
  {
    if (a[max] < a[i])
    {
      max = i;
    }
  }

  return max;
}
```

Listing 4.8: Implementation of `max_element`

4.4. The `max_seq` algorithm

In this section we consider the function `max_seq` (see Chapter 3, [8]) that is very similar to the `max_element` function of Section 4.2. The main difference between `max_seq` and `max_element` is that `max_seq` returns the maximum value (not just the index of it). Therefore, it requires a *non-empty* range as an argument.

Of course, `max_seq` can easily be implemented using `max_element` (see Listing 4.10). Moreover, using only the formal specification of `max_element` in Listing 4.7 we are also able to deductively verify the correctness of this implementation. Thus, we have a simple example of *modular verification* in the following sense:

> Any implementation of `max_element` that is separately proven to implement the contract in Listing 4.7 makes `max_seq` behave correctly. Once the contracts have been defined, the function `max_element` could be implemented in parallel, or just after `max_seq`, without affecting the verification of `max_seq`.

4.4.1. Formal specification of `max_seq`

A formal specification of `max_seq` in ACSL is shown in Listing 4.9.

```
/*@
  requires n > 0;
  requires \valid_read(p + (0..n-1));

  assigns  \nothing;

  ensures \forall integer i; 0 <= i <= n-1 ==> \result >= p[i];
  ensures \exists integer e; 0 <= e <= n-1 &&  \result == p[e];
*/
value_type max_seq(const value_type* p, size_type n);
```

Listing 4.9: Formal specification of `max_seq`

Using the first `requires`-clause we express that `max_seq` needs a *non-empty* range as input. By using the `ensures`-clause we express our postconditions. They formalize that `max_seq` indeed returns the maximum value of the range.

4.4.2. Implementation of max_seq

Listing 4.10 shows the trivial implementation of max_seq using max_element. Since max_seq requires a non-empty range the call of max_element returns an index to a maximum value in the range. The fact that max_element returns the smallest index is of no importance in this context.

```
value_type max_seq(const value_type* p, size_type n)
{
  return p[max_element(p, n)];
}
```

Listing 4.10: Implementation of max_seq

4.5. The `min_element` algorithm

The `min_element` algorithm in the C++ standard library[27] searches the minimum in a general sequence. The signature of our version of `min_element` reads:

```
size_type min_element(const value_type* a, size_type n);
```

The function `min_element` finds the smallest element in the range `a[0..n-1]`. More precisely, it returns the unique valid index `i` such that The return value of `min_element` is `n` if and only if `n == 0`. First we define the predicate `MinElement` by using the previously introduced predicate `LowerBound` (Listing 4.3) by stating that it is an lower bound that belongs to the sequence $a[0..n-1]$.

```
/*@
  predicate MinElement{L}(value_type* a, integer n, integer min) =
    0 <= min < n && LowerBound(a, 0, n, a[min]);
*/
```

Listing 4.11: Definition of the `MinElement` predicate

4.5.1. Formal specification of `min_element`

```
/*@
  requires \valid_read(a + (0..n-1));

  assigns \nothing;

  behavior empty:
    assumes n == 0;

    ensures result:  \result == 0;

  behavior not_empty:
    assumes 0 < n;

    ensures result:   0 <= \result < n;
    ensures minimum:  MinElement(a, n, \result);
    ensures first:    StrictLowerBound(a, 0, \result, a[\result]);

  complete behaviors;
  disjoint behaviors;
*/
size_type min_element(const value_type* a, size_type n);
```

Listing 4.12: Formal specification of `min_element`

[27]See `http://www.sgi.com/tech/stl/min_element.html`.

The ACSL specification of `min_element` is shown in Listing 4.12. Note that we also use the predicate `StrictLowerBound` (Listing 4.3) in order to express that `min_element` returns the *first* minimum position in $[0..n-1]$.

4.5.2. Implementation of `min_element`

Listing 4.13 shows implementation of `min_element` with rewritten loop invariants. In the loop invariants we also employ the predicates `LowerBound` and `StrictLowerBound` that we have used in the specification.

```
size_type min_element(const value_type* a, size_type n)
{
  if (0 == n)
  {
    return n;
  }

  size_type min = 0;

  /*@
    loop invariant bound:   0 <= i   <= n;
    loop invariant min:     0 <= min <  n;
    loop invariant lower:   LowerBound(a, 0, i, a[min]);
    loop invariant first:   StrictLowerBound(a, 0, min, a[min]);

    loop assigns min, i;
    loop variant n-i;
  */
  for (size_type i = 0; i < n; i++)
  {
    if (a[i] < a[min])
    {
      min = i;
    }
  }

  return min;
}
```

Listing 4.13: Implementation of `min_element`

5. Binary search algorithms

In this chapter, we consider the four *binary search* algorithms of the C++ standard library, namely

- `lower_bound` in Section 5.1
- `upper_bound` in Section 5.2
- `equal_range` in Section 5.3
- `binary_search` in Section 5.4.

All binary search algorithms require that their input array is sorted in ascending order. The predicate `Sorted` in Listing 5.1 formalizes these requirements.

```
/*@
  predicate
    Sorted{L}(value_type* a, integer n) =
      \forall integer i, j; 0 <= i < j < n  ==> a[i] <= a[j];
*/
```

Listing 5.1: The predicate `Sorted`

As in the case of the of maximum/minimum algorithms from Chapter 4 the binary search algorithms primarily use the less-than operator < (and the derived operators <=, > and >=) to determine whether a particular value is contained in a sorted range. Thus, different to the `find` algorithm in Section 3.3, the equality operator == will play only a supporting part in the specification of binary search.

In order to make the specifications of the binary search algorithms more compact and (arguably) more readable we use the predicates from Listing 4.3.

5.1. The lower_bound algorithm

The `lower_bound` algorithm is one of the four binary search algorithms of the C++ standard library. For our purposes we have modified the generic implementation[28] to that of an array of type `value_type`. The signature now reads:

```
size_type lower_bound(const value_type* a, size_type n,
                      value_type val);
```

As with the other binary search algorithms `lower_bound` requires that its input array is sorted in ascending order. Specifically, `lower_bound` will return the *largest* index `i` with `0 <= i <= n` such that for each index `k` with `0 <= k < i` the condition `a[k] < val` holds. This specification makes `lower_bound` a bit tricky to use as a search algorithm:

- If `lower_bound` returns `n` then for each index `i` with `0 <= i < n` holds `a[i] < val`. Thus, `val` is not contained in `a`.
- If, however, `lower_bound` returns an index `r` with `0 <= r < n` then we can only be sure that `a[i] < val` holds for `0 <= i < r` and that `val <= a[i]` holds for `r <= i < n`.

5.1.1. Formal specification of lower_bound

The ACSL specification of `lower_bound` is shown in Listing 5.2.

```
/*@
  requires \valid_read(a + (0..n-1));
  requires Sorted(a, n);

  assigns \nothing;

  ensures result:  0 <= \result <= n;
  ensures left:    StrictUpperBound(a, 0, \result, val);
  ensures right:   LowerBound(a, \result, n, val);
*/
size_type
lower_bound(const value_type* a, size_type n, value_type val);
```

Listing 5.2: Formal specification of `lower_bound`

- The preconditions express, by using the predicate `Sorted`, that the values in the (valid) array need to be sorted in ascending order.
- The postconditions formalize the central properties, mentioned above, of the return value of `lower_bound`.

[28] See `http://www.sgi.com/tech/stl/lower_bound.html`.

5.1.2. Implementation of lower_bound

Our implementation of lower_bound is shown in Listing 5.3. Each iteration step narrows down the range that contains the sought-after result. The loop invariants express that in each iteration step all indices less than the temporary left bound left contain values smaller than val and all indices not less than the temporary right bound right contain values not smaller than val.

```
size_type
lower_bound(const value_type* a, size_type n, value_type val)
{
  size_type left = 0;
  size_type right = n;
  size_type middle = 0;

  /*@
    loop invariant bound:  0 <= left <= right <= n;
    loop invariant left:   StrictUpperBound(a, 0,  left, val);
    loop invariant right:  LowerBound(a, right, n, val);

    loop assigns middle, left, right;
    loop variant right - left;
  */
  while (left < right)
  {
    middle = left + (right - left) / 2;

    if (a[middle] < val)
    {
      left = middle + 1;
    }
    else
    {
      right = middle;
    }
  }

  return left;
}
```

Listing 5.3: Implementation of lower_bound

5.2. The upper_bound algorithm

The upper_bound[29] algorithm is a version of the binary_search algorithm closely related to lower_bound of Section 5.1.

The signature reads:

```
size_type upper_bound(const value_type* a, size_type n,
                      value_type val)
```

In contrast to the lower_bound algorithm the upper_bound algorithm locates the *largest* index i with 0 <= i <= n such that for each index k with 0 <= k < i the condition a[k] <= val holds. This means:

- If upper_bound returns n then we can only be sure that for each index 0 <= i < n the relationship a[i] <= val.
- If upper_bound returns an index r with 0 <= r < n then we can be sure that val < a[i] holds for i where r <= i < n. Thus, if upper_bound returns 0 then we know that val is not contained in a.

5.2.1. Formal specification of upper_bound

The ACSL specification of upper_bound is shown in Listing 5.4.

```
/*@
  requires \valid_read(a + (0..n-1));
  requires Sorted(a, n);

  assigns \nothing;

  ensures result:  0 <= \result <= n;
  ensures left:    UpperBound(a, 0, \result, val);
  ensures right:   StrictLowerBound(a, \result, n, val);
*/
size_type
upper_bound(const value_type* a, size_type n, value_type val);
```

Listing 5.4: Formal specification of upper_bound

The specification is quite similar to the specification of lower_bound (see Listing 5.2). The difference can be seen in the postconditions. As we are searching for the upper bound this time, upper_bound has to ensures that

- all indices less than the returned one belong to elements are less than or equal to val
- all indices greater than or equal to the returned one belong to elements that are greater than val.

[29]See http://www.sgi.com/tech/stl/upper_bound.html.

5.2.2. Implementation of upper_bound

Our implementation of upper_bound is shown in Listing 5.5.

The loop invariants express that for each iteration step all indices less than the temporary left bound left contain values not greater than val and all indices not less than the temporary right bound right contain values greater than val.

```
size_type
upper_bound(const value_type* a, size_type n, value_type val)
{
  size_type left = 0;
  size_type right = n;
  size_type middle = 0;

  /*@
    loop invariant bound:  0 <= left <= right <= n;
    loop invariant left:   UpperBound(a, 0, left, val);
    loop invariant right:  StrictLowerBound(a, right, n, val);

    loop assigns middle, left, right;
    loop variant right - left;
  */
  while (left < right)
  {
    middle = left + (right - left) / 2;

    if (a[middle] <= val)
    {
      left = middle + 1;
    }
    else
    {
      right = middle;
    }
  }

  return right;
}
```

Listing 5.5: Implementation of upper_bound

5.3. The equal_range algorithm

The equal_range algorithm is one of the four binary search algorithms of the C++ standard library. For our purposes we have modified the generic implementation[30] to that of an array of type value_type. Moreover, instead of a pair of iterators, our version of equal_range returns a pair of indices. To be more precisely, the return type of equal_range is the struct size_type_pair from Listing 5.6. Thus, the signature of equal_range now reads:

```
size_type_pair equal_range(const value_type* a, size_type n,
                           value_type val);
```

As with the other binary search algorithms equal_range requires that its input array is sorted in ascending order. The specification of equal_range states that it *combines* the results of the algorithms lower_bound (Section 5.1) and upper_bound (Section 5.2).

5.3.1. Formal specification of equal_range

The ACSL specification of equal_range is shown in Listing 5.6.

```
struct spair
{
  size_type first;
  size_type second;
};

typedef struct spair size_type_pair;

/*@
  requires \valid_read(a + (0..n-1));
  requires Sorted(a, n);

  assigns \nothing;

  ensures result:  0 <= \result.first <= \result.second <= n;
  ensures left:    StrictUpperBound(a, 0, \result.first, val);
  ensures middle:  ConstantRange(a, \result.first,
                                 \result.second, val);
  ensures right:   StrictLowerBound(a, \result.second, n, val);
 */
size_type_pair
equal_range(const value_type* a, size_type n, value_type val);
```

Listing 5.6: Formal specification of equal_range

The preconditions express that the values in the (valid) array need to be sorted in ascending order.

The postconditions express that the pair of indices (f, s) returned by equal_range satisfy the following properties:

[30] See http://www.sgi.com/tech/stl/equal_range.html.

- $0 \leq f \leq s \leq n$
- the set of indices $[f, s) = \{i \mid f \leq i < s\}$ is the *largest* set for which $a[i] = \text{val}$ holds

5.3.2. Implementation of `equal_range`

Our implementation of `equal_range` is shown in Listing 5.7. We call the two functions `lower_bound` and `upper_bound` and return their respective results as a pair. However, instead of doing this straightforward, we use the auxiliary function `make_pair`[31] and formulate an assertion for its arguments `first` ≤ `second`. Using this assertion simplifies the task of *automatically* proving the postcondition in Listing 5.6.

```
/*@
    assigns \nothing;

    ensures \result.first  == first;
    ensures \result.second == second;
*/
size_type_pair make_pair(size_type first, size_type second)
{
  size_type_pair pair;
  pair.first  = first;
  pair.second = second;

  return pair;
}

size_type_pair
equal_range(const value_type* a, size_type n, value_type val)
{
  size_type first  = lower_bound(a, n, val);
  size_type second = upper_bound(a, n, val);
  //@ assert aux: second < n  ==>  val < a[second];

  return make_pair(first, second);
}
```

Listing 5.7: Implementation of `equal_range`

In an earlier version of this document we had proven the similar assertion `first <= second` with the interactive theorem prover Coq. After reviewing this proof we formulated the new assertion `aux` that uses a fact from the postcondition of `upper_bound` (Listing 5.4). The benefit of this reformulation is that both the assertion `aux` and the postcondition `first <= second` can now be verified automatically.

[31] This functions is modelled after the C++ template function `std::make_pair`.

5.4. The binary_search algorithm

The binary_search algorithm is one of the four binary search algorithms of the C++ standard library. For our purposes we have modified the generic implementation[32] to that of an array of type value_type. The signature now reads:

```
bool binary_search(const value_type* a, size_type n,
                   value_type  val);
```

Again, binary_search requires that its input array is sorted in ascending order. It will return **true** if there exists an index i in a such that a[i] == val holds.[33]

5.4.1. Formal specification of binary_search

The ACSL specification of binary_search is shown in Listing 5.8.

```
/*@
  requires \valid_read(a + (0..n-1));
  requires Sorted(a, n);

  assigns \nothing;

  ensures result:  \result <==>
                   \exists integer i; 0 <= i < n && a[i] == val;
 */
bool binary_search(const value_type* a, size_type n, value_type val);
```

Listing 5.8: Formal specification of binary_search

Note that we can use our previously introduced predicate HasValue (see Page 35) in Listing 5.9. It is interesting to compare this specification with that of find in Listing 3.12. Both find and binary_search allow to determine whether a value is contained in an array. The fact that the C++ standard library requires that find has *linear* complexity whereas binary_search must have a *logarithmic* complexity can currently not be expressed with ACSL.

[32]See http://www.sgi.com/tech/stl/binary_search.html.
[33]To be more precise: The C++ standard library requires that (a[i] <= val) && (val <= a[i]) holds. For our definition of value_type (see Section 1.2) this means that val equals a[i].

```
/*@
  requires \valid_read(a + (0..n-1));
  requires Sorted(a, n);

  assigns \nothing;

  ensures result:  \result <==> HasValue(a, n, val);
 */
bool binary_search(const value_type* a, size_type n, value_type val);
```

Listing 5.9: Formal specification of `binary_search` using the `HasValue` predicate

5.4.2. Implementation of `binary_search`

Our implementation of `binary_search` is shown in Listing 5.10.

```
bool binary_search(const value_type* a, size_type n, value_type val)
{
  size_type i = lower_bound(a, n, val);
  return i < n && a[i] <= val;
}
```

Listing 5.10: Implementation of `binary_search`

The function `binary_search` first calls `lower_bound` from Section 5.1. Remember that if `lower_bound` returns an index `0 <= i < n` then we can be sure that `val <= a[i]` holds.

6. Mutating algorithms

Let us now turn our attention to another class of algorithms, viz. *mutating* algorithms, i.e., algorithms that change one or more ranges. In Frama-C, you can explicitly specify that, e.g., entries in an array a may be modified by a function f, by including the following *assigns clause* into the f's specification:

```
assigns a[0..length-1];
```

The expression length-1 refers to the value of length when f is entered, see [9, Section 2.3.2]. Below are the seven example algorithms we will discuss next.

swap (Section 6.1 on Page 70) exchanges two values.

fill (Section 6.2 on Page 72) initializes each element of an array by a given fixed value.

swap_ranges (Section 6.3 on Page 74) exchanges the contents of the arrays of equal length, element by element. We use this example to present "modular verification", as swap_ranges reuses the verified properties of swap.

copy (Section 6.4 on Page 76) copies a source array to a destination array.

reverse_copy and reverse (Sections 6.5 and 6.6 on Pages 78 and 80, respectively) reverse an array. Whereas
reverse_copy copies the result to a separate destination array, the reverse algorithm works in place.

rotate_copy (Section 6.7 on Page 82) rotates a source array by m positions and copies the results to a destination array.

replace_copy (Section 6.8 on Page 84) copies a source array to a destination array, but substitutes each occurrence of a given old value by a given **new** value.

remove_copy copies a source array to a destination array, but omits each occurrence of a given value. We provide two specifications for remove_copy:

- First we provide a relatively simple contract that omits, however, an important aspect of the informal specification (see Section 6.9 on Page 86).
- In Section 6.10 (Page 88) we show how the missing part of the specification can be expressed.

6.1. The swap algorithm

The swap algorithm[34] in the C++ standard library exchanges the contents of two variables. Similarly, the iter_swap algorithm[35] exchanges the contents referenced by two pointers. Since C and hence ACSL, does not support an & type constructor ("declarator"), we will present an algorithm that processes pointers and refer to it as swap.

6.1.1. Formal specification of swap

The ACSL specification for the swap function is shown in Listing 6.1.

```
/*@
  requires \valid(p);
  requires \valid(q);

  assigns *p;
  assigns *q;

  ensures *p == \old(*q);
  ensures *q == \old(*p);
*/
void swap(value_type* p, value_type* q);
```

Listing 6.1: Formal specification of swap

The preconditions which formalize by the requires-clause states that both argument pointers to the swap function must be dereferenceable.

The assigns-clauses formalize that the swap algorithm modifies only the entries referenced by the pointers p and q. Nothing else may be altered. In general, when more than one *assigns clause* appears in a function's specification, it permitted to modify any of the referenced locations. However, if no *assigns clause* appears at all, the function is free to modify any memory location, see [9, Section 2.3.2]. To forbid a function to do any modifications outside its scope, a clause

```
  assigns \nothing;
```

must be used, as we practised in the example specifications in Chapter 3.

Upon termination of swap the entries must be mutually exchanged. We can express those postconditions by using the ensures-clause. The expression \old(*p) refers to the pre-state of the function contract, whereas by default, a postcondition refers the values after the functions has been terminated.

[34]See http://www.sgi.com/tech/stl/swap.html.
[35]See http://www.sgi.com/tech/stl/iter_swap.html.

6.1.2. Implementation of `swap`

Listing 6.2 shows the usual straight-forward implementation of `swap`. No interspersed ACSL is needed to get it verified by Frama-C.

```
void swap(value_type* p, value_type* q)
{
  value_type save = *p;
  *p = *q;
  *q = save;
}
```

Listing 6.2: Implementation of `swap`

6.2. The `fill` algorithm

The `fill` algorithm in the C++ Standard Library initializes general sequences with a particular value. For our purposes we have modified the generic implementation[36] to that of an array of type `value_type`. The signature now reads:

```
void fill(value_type* a, size_type n, value_type val);
```

6.2.1. Formal specification of `fill`

Listing 6.3 shows the formal specification of `fill` in ACSL. We can express the postcondition of `fill` simply by using the predicate `ConstantRange` from Listing 4.3.

```
/*@
  requires valid: \valid(a + (0..n-1));

  assigns a[0..n-1];

  ensures constant: ConstantRange(a, 0, n, val);
*/
void fill(value_type* a, size_type n, value_type val);
```

Listing 6.3: Formal specification of `fill`

[36] See `http://www.sgi.com/tech/stl/fill.html`

6.2.2. Implementation of `fill`

Listing 6.4 shows an implementation of `fill`.

```
void fill(value_type* a, size_type n,  value_type val)
{
  /*@
    loop invariant bound:    0 <= i <= n;
    loop invariant constant: ConstantRange(a, 0, i, val);

    loop assigns i, a[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i)
  {
    a[i] = val;
  }
}
```

Listing 6.4: Implementation of `fill`

The loop invariant `bound` is necessary to prove that each access to the range `a` occurs with valid indices. The loop invariant `constant` expresses that for each iteration the array is filled with the value of `val` up to the index `i` of the iteration. Note that we use here again the predicate `ConstantRange` from Listing 4.3.

6.3. The swap_ranges algorithm

The `swap_ranges` algorithm[37] in the C++ standard library exchanges the contents of two expressed ranges element-wise. After translating C++ reference types and iterators to C, our version of the original signature reads:

```
void swap_ranges(value_type* a, size_type n, value_type* b);
```

We do not return a value since it would equal `n`, anyway.

This function refers to the previously discussed algorithm `swap`. Thus, `swap_ranges` serves as another example for "modular verification". The specification of `swap` will be automatically integrated into the proof of `swap_ranges`.

6.3.1. Formal specification of swap_ranges

Listing 6.5 shows an ACSL specification for the `swap_ranges` algorithm.

```
/*@
  requires valid_a: \valid(a + (0..n-1));
  requires valid_a: \valid(b + (0..n-1));
  requires sep:     \separated(a+(0..n-1), b+(0..n-1));

  assigns a[0..n-1];
  assigns b[0..n-1];

  ensures equal_a: EqualRanges{Here,Old}(a, n, b);
  ensures equal_b: EqualRanges{Old,Here}(a, n, b);
*/
void swap_ranges(value_type* a, size_type n, value_type* b);
```

Listing 6.5: Formal specification of `swap_ranges`

The `swap_ranges` algorithm works correctly only if `a` and `b` do not overlap. Because of that fact we use the `separated`-clause to tell Frama-C that `a` and `b` must not overlap.

With the `assigns`-clause we postulate that the `swap_ranges` algorithm alters the elements contained in two distinct ranges, modifying the corresponding elements and nothing else.

The postconditions of `swap_ranges` specify that the content of each element in its post-state must equal the pre-state of its counterpart. We can use the predicate `EqualRanges` (see Listing 3.2) together with the label `Old` and `Here` to express the postcondition of `swap_ranges`. In our specification in Listing 6.5, for example, we specify that the array `a` in the memory state that corresponds to the label `Here` is equal to the array `b` at the label `Old`. Since we are specifying a postcondition `Here` refers to the post-state of `swap_ranges` whereas `Old` refers to the pre-state.

[37] See `http://www.sgi.com/tech/stl/swap_ranges.html`.

6.3.2. Implementation of swap_ranges

Listing 6.6 shows an implementation of swap_ranges together with the necessary loop annotations.

```
void swap_ranges(value_type* a, size_type n, value_type* b)
{
  /*@
    loop invariant bound: 0 <= i <= n;
    loop invariant equal_a: EqualRanges{Here,Pre}(a, i, b);
    loop invariant equal_b: EqualRanges{Here,Pre}(b, i, a);

    loop invariant unchanged_a: Unchanged{Here,Pre}(a, i, n);
    loop invariant unchanged_b: Unchanged{Here,Pre}(b, i, n);

    loop assigns i, a[0..n-1], b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i)
  {
    swap(a+i, b+i);
  }
}
```

Listing 6.6: Implementation of swap_ranges

For the postcondition of the specification in Listing 6.5 to hold, our loop invariants must ensure that at each iteration all of the corresponding elements that have already been visited are swapped.

Note that there are two additional loop invariants which claim that all the elements that have not visited yet equal their original values. This a workaround that allows us to prove the postconditions of swap_ranges despite the fact that the loop assigns is coarser than it should be. The predicate Unchanged from Listing 6.7 is used to express this property.

```
/*@
  predicate
    Unchanged{A,B}(value_type* a, integer first, integer last) =
      \forall integer i; first <= i < last
         ==> \at(a[i], A) == \at(a[i], B);
*/
```

Listing 6.7: The predicate Unchanged

6.4. The copy algorithm

The copy algorithm in the C++ Standard Library implements a duplication algorithm for general sequences. For our purposes we have modified the generic implementation[38] to that of a range of type value_type. The signature now reads:

```
void copy(const value_type* a, size_type n, value_type* b);
```

Informally, the function copies every element from the source range a to the destination range b.

6.4.1. Formal specification of copy

The ACSL specification of copy is shown in Listing 6.8. The copy algorithm expects that the ranges a and b are valid for reading and writing, respectively. Also important is that the ranges do not overlap, this property is expressed with the separated-clause in our specification.

```
/*@
  requires valid_a: \valid_read(a + (0..n-1));
  requires valid_b:      \valid(b + (0..n-1));
  requires sep:      \separated(a + (0..n-1), b + (0..n-1));

  assigns b[0..n-1];

  ensures equal: EqualRanges{Here,Here}(a, n, b);
*/
void copy(const value_type* a, const size_type n, value_type* b);
```

Listing 6.8: Formal specification of copy

Furthermore the function copy assigns the elements from the source range a to the destination range b, modifying the memory of the elements pointed to by b. Again, we can use the EqualRanges predicate from Section 3.1 to express that the array a equals b after copy has been called. Nothing else must be altered. To state this we use the assigns-clause.

Note the similarities in the specifications of copy and swap_ranges (Section 6.3).

[38] See http://www.sgi.com/tech/stl/copy.html.

6.4.2. Implementation of copy

Listing 6.9 shows an implementation of the copy function.

```
void copy(const value_type* a, size_type n, value_type* b)
{
  /*@
    loop invariant bound: 0 <= i <= n;
    loop invariant equal: EqualRanges{Here,Here}(a, i, b);
    loop assigns   i, b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i)
  {
    b[i] = a[i];
  }
}
```

Listing 6.9: Implementation of copy

Here are some remarks on its loop invariants.

For the postcondition to be true, we must ensure that for every element i, the comparison a[i] == b[i] is **true**. This can be expressed by using the EqualRanges predicate.

The assigns clause ensures that nothing but the range b[0..i-1] and the loop variable i is modified. In order to prove the termination of the loop for every possible n we use the loop variant n-i and cover it with an assertion.

6.5. The reverse_copy algorithm

The reverse_copy[39] algorithm of the C++ Standard Library invert the order of elements in a sequence. reverse_copy does not change the input sequence and copies its result to the output sequence. For our purposes we have modified the generic implementations to that of a range of type value_type. The signature now reads:

```
void reverse_copy(const value_type* a, size_type n, value_type* b);
```

6.5.1. Formal specification of reverse_copy

Informally, reverse_copy copies the elements from the array a into array b such that the copy is a reverse of the original array. Thus, after calling reverse_copy the following conditions shall be satisfied.

```
  b[0]    ==  a[n-1]
  b[1]    ==  a[n-2]
   ⋮       ⋮     ⋮
b[n-1]    ==  a[ 0 ]
```

In order to concisely formalize these condition we write the two (overloaded) predicates Reversed that are shown in Listing 6.10.

```
/*@
   predicate
     Reversed{A,B}(value_type* a, integer n, value_type* b,
                   integer first, integer last) =
       \forall integer k; first <= k < last
          ==> \at(a[k], A) == \at(b[n-1-k], B);


   predicate
     Reversed{A,B}(value_type* a, integer n, value_type* b) =
       Reversed{A,B}(a, n, b, 0, n);

*/
```

Listing 6.10: Predicate Reversed

[39] See http://www.sgi.com/tech/stl/reverse_copy.html.

The ACSL specification of reverse_copy is shown in Listing 6.11.

```
/*@
  requires valid_a: \valid_read(a + (0..n-1));
  requires valid_b:      \valid(b + (0..n-1));
  requires sep:       \separated(a + (0..n-1), b + (0..n-1));

  assigns b[0..(n-1)];

  ensures reverse:  Reversed{Here,Here}(a, n, b);
*/
void reverse_copy(const value_type* a, size_type n, value_type* b);
```

Listing 6.11: Formal specification of reverse_copy

The postcondition states that the contents of a was copied reversely to b.

6.5.2. Implementation of reverse_copy

Listing 6.12 shows an implementation of the reverse_copy function.

```
void reverse_copy(const value_type* a, size_type n, value_type* b)
{
  /*@
    loop invariant bound:   0 <= i <= n;
    loop invariant reverse: Reversed{Here,Here}(b, n, a, 0, i);
    loop assigns i, b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i)
  {
    b[i] = a[n-1-i];
  }
}
```

Listing 6.12: Implementation of reverse_copy

For the postcondition to be true, we must ensure that for every element i, the comparison b[i] == a[n-1-i] holds. This is formalized by the loop invariants. You can see that it is very similar to the postcondition in Listing 6.11.

6.6. The `reverse` algorithm

The `reverse`[40] algorithm of the C++ Standard Library invert the order of elements in a sequence. The `reverse` algorithm works in place, meaning, that it modifies its input sequence. For our purposes we have modified the generic implementations to that of a range of type `value_type`. The signature now reads:

```
void reverse(value_type* a, size_type n);
```

6.6.1. Formal specification of `reverse`

The ACSL specification for the `reverse` function is shown in listing 6.13.

```
/*@
  requires valid: \valid(a + (0..n-1));

  assigns a[0..(n-1)];

  ensures reverse: Reversed{Here,Old}(a, n, a);
*/
void reverse(value_type* a, size_type n);
```

Listing 6.13: Formal specification of `reverse`

In the postcondition we use again the predicate `Reversed` from Listing 6.10.

[40]See `http://www.sgi.com/tech/stl/reverse.html`.

6.6.2. Implementation of `reverse`

Listing 6.14 shows an implementation of the `reverse` function where the elements of the first half of the array are swapped with the corresponding elements of the second half. Note the assertion for the variable `half` in the loop body.

```
void reverse(value_type* a, size_type n)
{
  const size_type half = n / 2;

  /*@
    loop invariant bound: 0 <= i <= half;

    loop invariant left:   Reversed{Here,Pre} (a, n, a, 0,  i);
    loop invariant middle: Unchanged{Here,Pre}(a, i, n - i);
    loop invariant right:  Reversed{Here,Pre} (a, n, a, n-i, n);

    loop assigns i, a[0..n-1];
    loop variant half - i;
  */
  for (size_type i = 0; i < half; ++i)
  {
    //@ assert 0 < half ==> i < n-1-i;
    swap(&a[i], &a[n-1-i]);
  }
}
```

Listing 6.14: Implementation of `reverse`

We reuse the predicates `Reversed` (Listing 6.10) and `Unchanged` (Listing 6.7) in order to write concise loop invariants.

6.7. The `rotate_copy` algorithm

The `rotate_copy` algorithm in the C++ Standard Library rotates a sequence by `m` positions and copies the results to another same sized sequence. For our purposes we have modified the generic implementation[41] to that of a range of type `value_type`. The signature now reads:

```
void rotate_copy(const value_type* a, size_type m,
                 size_type n, value_type* b);
```

Informally, the first `m` elements of the array `a` become the last `m` elements of the array `b` whereas the last `n-m` elements of the array `a` become the first `n-m` elements of the array `b`. Figure 6.15 illustrates the effects of `rotate_copy`.

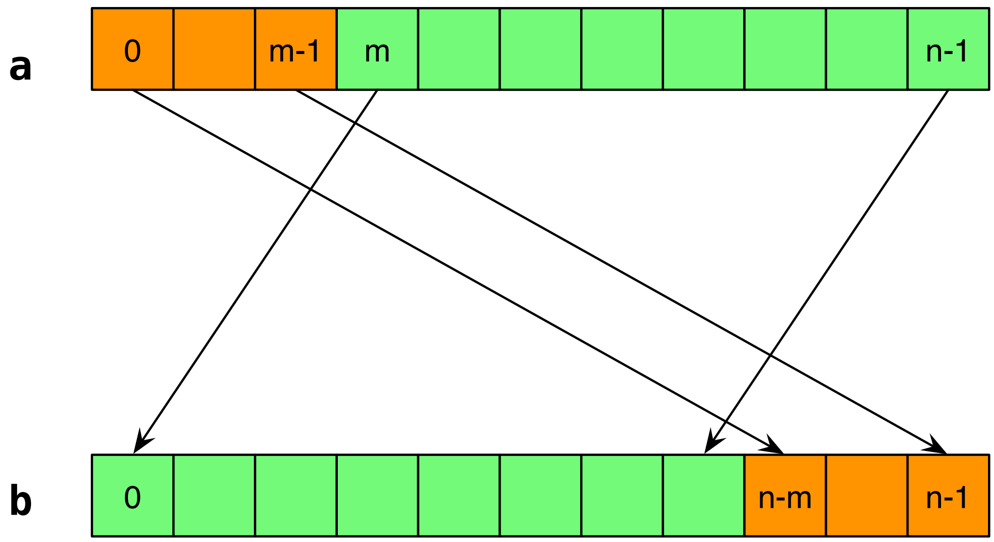


Figure 6.15.: Effects of `rotate_copy`

[41] See `http://www.sgi.com/tech/stl/rotate_copy.html`.

6.7.1. Formal specification of rotate_copy

The ACSL specification of rotate_copy is shown in Listing 6.16.

```
/*@
  requires sub:      0 <= m <= n;
  requires valid_a: \valid_read(a + (0..n-1));
  requires valid_b:      \valid(b + (0..n-1));
  requires sep:      \separated(a + (0..n-1), b + (0..n-1));

  assigns b[0..(n-1)];

  ensures first:  EqualRanges{Here,Here}(a,   m,   b+(n-m));
  ensures last:   EqualRanges{Here,Here}(a+m, n-m, b);
*/
void rotate_copy(const value_type* a, size_type m, size_type n,
                 value_type* b);
```

Listing 6.16: Formal specification of rotate_copy

The separated-clause tells WP that a and b must not overlap.

6.7.2. Implementation of rotate_copy

Listing 6.17 shows an implementation of the rotate_copy function. The implementation simply calls the function copy twice.

```
void rotate_copy(const value_type* a, size_type m, size_type n,
                 value_type* b)
{
  copy(a,   m, b+(n-m));
  copy(a + m, n-m, b);
}
```

Listing 6.17: Implementation of rotate_copy

6.8. The replace_copy algorithm

The replace_copy algorithm of the C++ Standard Library substitutes specific elements from general sequences. Here, the general implementation[42] has been altered to process value_type ranges. The new signature reads:

```
size_type replace_copy(const value_type* a, size_type n,
                       value_type* b,
                       value_type oldv, value_type newv);
```

The replace_copy algorithm copies the elements from the range a[0..n] to range b[0..n], substituting every occurrence of oldv by newv. The return value is the length of the range. As the length of the range is already a parameter of the function this return value does not contain new information. However, the length returned is analogous to the implementation of the C++ Standard Library.

6.8.1. Formal specification of replace_copy

The ACSL specification of replace_copy is shown in Listing 6.18.

```
/*@
  requires valid_a: \valid_read(a + (0..n-1));
  requires valid_b:      \valid(b + (0..n-1));
  requires sep:       \separated(a + (0..n-1), b + (0..n-1));

  assigns b[0..n-1];

  ensures change: \forall integer i; 0 <= i < n
            ==> (\old(a[i]) == oldv ==> b[i] == newv);

  ensures keep:   \forall integer i; 0 <= i < n
            ==> (\old(a[i]) != oldv ==> b[i] == \old(a[i]));

  ensures result: \result == n;
*/
size_type replace_copy(const value_type* a, size_type n,
                       value_type* b,
                       value_type  oldv, value_type newv);
```

Listing 6.18: Formal specification of the replace_copy

In particular, the specification requires that the arrays a and b are non-overlapping. The core functionality of replace_copy is specified as follows: For every element a[j] of a, we have two possibilities. Either it equals oldv or it is different from oldv. In the former case, we specify that the corresponding element b[j] has to be substituted with newv. In the latter case, we specify that b[j] equals a[j].

[42]See http://www.sgi.com/tech/stl/replace_copy.html.

6.8.2. Implementation of replace_copy

An implementation (including loop annotations) of replace_copy is shown in Listing 6.19. Note how the structure of the loop annotations resembles the specification of Listing 6.18.

```
size_type replace_copy(const value_type* a, size_type n,
                       value_type* b,
                       value_type  oldv, value_type newv)
{
  /*@
    loop invariant bounds: 0 <= i <= n;

    loop invariant change: \forall integer k; 0 <= k < i
         ==> \at(a[k], Pre) == oldv ==> b[k] == newv;

    loop invariant keep:   \forall integer k; 0 <= k < i
         ==> \at(a[k], Pre) != oldv ==> b[k] == \at(a[k], Pre);

    loop assigns i, b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i)
  {
    b[i] = (a[i] == oldv ? newv : a[i]);
  }

  return n;
}
```

Listing 6.19: Implementation of the replace_copy algorithm

6.9. The remove_copy algorithm

The remove_copy algorithm of the C++ Standard Library copies all elements of a sequence other than a given value. Here, the general implementation has been altered to process value_type ranges.[43] The new signature reads:

```
size_type remove_copy(const value_type* a, size_type n,
                      value_type* b, value_type v);
```

The most important facts of this algorithms are

1. The return value is the length of the resulting range.
2. The remove_copy algorithm copies elements that are not equal to v from range a[0..n-1] to the range b[0..\result-1].
3. The algorithm is stable, that is, the relative order of the elements in b is the same as in a.

6.9.1. Formal specification of remove_copy

In order to achieve a concise specification we start with introducing two auxiliary predicates.

We use the predicate PreserveCount in Listing lst:preservecount in order to express that the number of elements that are different from v is the same in the source and target ranges.

```
/*@
  predicate
    PreserveCount(value_type* a, size_type m,
                  value_type* b, size_type n, value_type v) =
      \forall value_type x; x != v ==>
         Count(a, m, x) == Count(b, n, x);
*/
```

Listing 6.20: The predicate PreserveCount

The predicate Unchanged from Listing 6.7 is used to express that remove_copy does not change the elements b[\result..n-1].

[43] See http://www.sgi.com/tech/stl/remove_copy.html.

Listing 6.21 shows our first attempt to specify remove_copy.

```
/*@
  requires valid_a: \valid_read(a + (0..n-1));
  requires valid_b:      \valid(b + (0..n-1));
  requires sep:       \separated(a + (0..n-1), b+(0..n-1));

  assigns  b[0..(n-1)];

  ensures  bound:       0 <= \result <= n;
  ensures  result:      \result == n - Count(a, n, v);
  ensures  removed:     !HasValue(b, \result, v);
  ensures  kept:        PreserveCount(a, n, b, \result, v);
  ensures  unchanged:   Unchanged{Here,Old}(b, \result, n);
*/
size_type remove_copy(const value_type* a, size_type  n,
                      value_type* b, value_type v);
```

Listing 6.21: Formal specification of remove_copy

Note the re-use of predicate HasValue (Listing 3.11) to express that the target range does not contain the value v.

We use the logic function Count from Section 3.8 to express that only the elements that differ from v are copied:

- The return value of remove_copy is the number of copied elements. This value must obviously be equal to n diminished by the number of occurrences of v in a[0..n-1].
- We use Count to express that any value that differs from v appears as often in the input range a[0..n-1] as in the output range b[0..n-1].

While this formal specification is a good representation of the informal requirements it does not capture that remove_copy is *stable*: Given a range a = {1,0,5,2,0,5} and a value v = 0 the expected result of remove_copy is b = {1,5,2,5}. However, since Count is invariant under permutations the specification in Listing 6.21 would also allow the result b = {5,5,1,2}. In Section 6.10 we will discuss how the stability of remove_copy can be captured in an ACSL specification.

6.9.2. Implementation of remove_copy

An implementation of remove_copy is shown in Listing 6.22. Not surprisingly, the logical function Count and the predicates PreserveCount and Unchanged also appear in the loop invariants of remove_copy.

```
size_type remove_copy(const value_type* a, size_type n,
                      value_type* b, value_type v)
{
  size_type j = 0;

  /*@
    loop invariant bound:      0 <= j <= i <= n;
    loop invariant result:     j == i - Count(a, i, v);
    loop invariant removed:    !HasValue(b, j, v);
    loop invariant kept:       PreserveCount(a, i, b, j, v);
    loop invariant unchanged:  Unchanged{Here,Pre}(b, j, n);

    loop assigns i, j, b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i)
  {
    //@ assert EqualRanges{Here,Pre}(a, n);
    if (a[i] != v)
    {
      b[j++] = a[i];
    }
  }

  return j;
}
```

Listing 6.22: Implementation of remove_copy

6.10. Capturing the stability of remove_copy

The most important facts of this algorithms are

1. The remove_copy algorithm copies elements that are different from v from the range a[0..n-1] to a range beginning at b[0].
2. The return value is the number of copied elements.
3. The algorithm is stable, that is, the relative order of the elements in b is the same as in a.

A particular challenge in the specification of `remove_copy` is how to express the stability of the removal. Figure 6.23 shows how `remove_copy` is supposed to copy elements that differ from `v` from one range to the other.

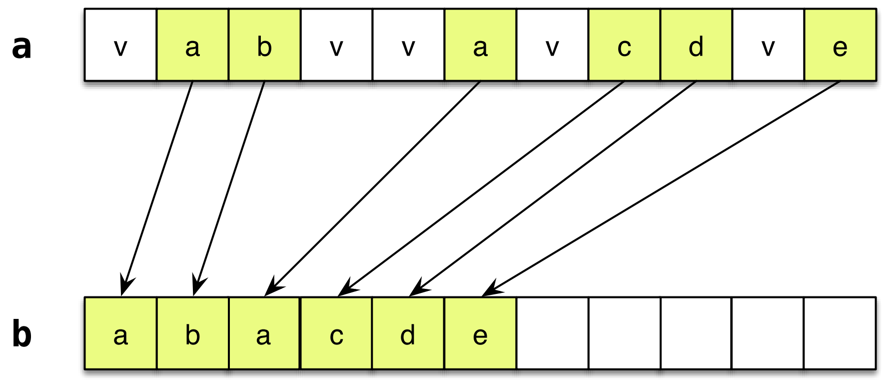


Figure 6.23.: Stability of `remove_copy`

Figure 6.24 shows, with respect to array indices, how the elements different from `v` "slide" to positions with smaller indices. The main observation here is that *an element slides as many positions down as there are elements in front of it that equal* `v`.

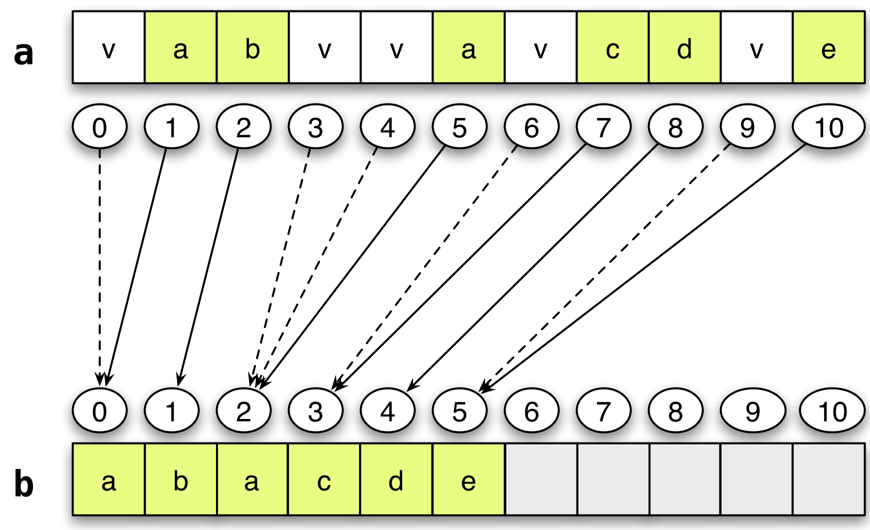


Figure 6.24.: Stability of `remove_copy` with respect to indices

As it turns out, it is quite easy[44] to express this property using the previously introduced logic function `Count` (see Listing 3.24 on Page 44). We simply define in Listing 6.25 a logic function `RemoveCount` which subtracts from every position `i` the number of occurrences of `v` that come

[44] To tell the truth, it took us quite some time to really understand how easy it is!

before i. Note that RemoveCount(a, v, i) equals the number of elements of a[0..i-1] that are copied to the destination range b[0..n-1] by remove_copy.

```
/*@
  logic
    integer RemoveCount{L}(value_type* a, integer i, value_type v) =
     i - Count{L}(a, i, v);

  lemma RemoveCountEmpty:
     \forall value_type *a, v, integer i;
       i <= 0 ==> RemoveCount(a, i, v) == i;

  lemma RemoveCountHit:
     \forall value_type *a, v, integer i; a[i] == v ==>
      RemoveCount(a, i+1, v) == RemoveCount(a, i, v);

  lemma RemoveCountMiss:
     \forall value_type *a, v, integer i; a[i] != v ==>
       RemoveCount(a, i+1, v) == RemoveCount(a, i, v) + 1;

  lemma RemoveCountRead{L1,L2}:
     \forall value_type *a, v, integer i; EqualRanges{L1,L2}(a, i) ==>
       RemoveCount{L1}(a, i, v) == RemoveCount{L2}(a, i, v);
*/
```

Listing 6.25: The logic function RemoveCount

Also, note that RemoveCount is defined for all integers, including those indices i where a[i] equals v (see the dashed lines in Figure 6.24). In the specification of remove_copy we will, however, only use RemoveCount for indices where a[i] is different from v.

6.10.1. Formal specification of remove_copy

The predicate StableRemove (Listing 6.26) uses RemoveCount to formally capture the stability with respect to corresponding elements of the source and target ranges.

```
/*@
  predicate
    StableRemove(value_type* a, integer n,
                 value_type* b, value_type v) =
          \forall integer i; 0 <= i < n ==>
             a[i] != v ==> b[RemoveCount(a, i, v)] == a[i];
*/
```

Listing 6.26: The predicate StableRemove

Listing 6.27 shows improved specification of remove_copy that also captures the required stability.

```
/*@
  requires valid_a: \valid_read(a + (0..n-1));
  requires valid_b:      \valid(b + (0..n-1));
  requires sep:      \separated(a + (0..n-1), b+(0..n-1));

  assigns  b[0..(n-1)];

  ensures  bound:      0 <= \result <= n;
  ensures  result:     \result == RemoveCount(a, n, v);
  ensures  removed:    !HasValue(b, \result, v);
  ensures  kept:       PreserveCount(a, n, b, \result, v);
  ensures  unchanged:  Unchanged{Here,Old}(b, \result, n);
  ensures  stable:     StableRemove(a, n, b, v);
*/
size_type remove_copy(const value_type* a, size_type  n,
                      value_type* b, value_type v);
```

Listing 6.27: Improved formal specification of remove_copy

There are essentially two changes compared to the specification in Listing 6.21 on Page 87.

1. We now use RemoveCount in order to specify the expected return value in postcondition result.
2. We use StableRemove in the new postcondition stable. Here we exactly specify to which element in the output range b[0..n-1] an element of the input range a[0..n-1], that is different from v, is copied.

6.10.2. Implementation of remove_copy

Listing 6.28 shows the additional loop annotations that are necessary to verify the stronger specification in Listing 6.27.

```
size_type remove_copy(const value_type* a, size_type n,
                      value_type* b, value_type v)
{
  size_type j = 0;

  /*@
    loop invariant bound:      0 <= j <= i <= n;
    loop invariant result:     j == RemoveCount(a, i, v);
    loop invariant removed:    !HasValue(b, j, v);
    loop invariant kept:       PreserveCount(a, i, b, j, v);
    loop invariant unchanged:  Unchanged{Here,Pre}(b, j, n);
    loop invariant stable:     StableRemove(a, i, b, v);

    loop assigns i, j, b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i)
  {
    //@ assert EqualRanges{Here,Pre}(a, n);
    if (a[i] != v)
    {
      b[j++] = a[i];
    }
  }

  return j;
}
```

Listing 6.28: Implementation of remove_copy with additional loop invariants

In order to prove the additional loop invariant `stable` we rely on the following monotonicity properties of `RemoveCount`. The proof of the lemmas in Listing 6.29 relies on the properties of `Count` that have been formulated in Listings 3.24 and 3.25.

```
/*@
  lemma RemoveCountMonotonic :
     \forall value_type *a, v, integer m, n; 0 <= m <= n ==>
        RemoveCount(a, m, v) <= RemoveCount(a, n, v);

  lemma RemoveCountStrictlyMonotonic :
     \forall value_type *a, v, integer n;
       \forall integer i; 0 <= i < n ==> a[i] != v  ==>
          RemoveCount(a, i, v) < RemoveCount(a, n, v);
*/
```

Listing 6.29: Additional lemmas for `RemoveCount`

7. Numeric algorithms

The algorithms that we considered so far only *compared*, *read* or *copied* values in sequences. In this chapter, we consider the so-called *numeric* algorithms of the C++ standard library that use algebraic operations on `value_type` in order to combine the elements of sequences.

We consider the following algorithms.

`iota` writes sequentially increasing values into a range (Section 7.1 on Page 96)

`accumulate` computes the sum of the elements in a range (Section 7.2 on Page 97)

`inner_product` computes the inner product of two ranges (Section 7.3 on Page 100)

`partial_sum` computes the sequence of partial sums of a range (Section 7.4 on Page 103)

`adjacent_difference` computes the differences of adjacent elements in a range (Section 7.5 on Page 106)

The formal specification of these algorithms raises new questions. In particular, we now have to deal with arithmetic overflows in `value_type`.

7.1. The iota algorithm

The iota algorithm in the C++ standard library assigns sequentially increasing values to a range, where the initial value is user defined. Our version of the original signature[45] reads:

```
void iota(value_type* a, size_type n, value_type val);
```

Starting at val, the function assigns consecutive integers to the range a. When specifying iota we must be careful to deal with possible overflows.

7.1.1. Formal specification of iota

The ACSL specification of iota is shown in Listing 7.1.

The specification of iota refers to INT_MAX which is defined in limits.h.

```
/*@
  requires valid:  \valid(a + (0..n-1));
  requires limit1: n <= INT_MAX;
  requires limit2: val + n <= INT_MAX;

  assigns a[0..n-1];

  ensures increment: \forall integer k; 0 <= k < n
                     ==> a[k] == val + k;
*/
void iota(value_type* a, size_type n, value_type val);
```

Listing 7.1: Formal specification of iota

In order to avoid integer overflows both the length n of the array and the sum val+n must not be greater than the constant INT_MAX. Note that the expression "val+n" is of type integer (cf. Sect. 3.1), and therefore cannot overflow. In contrast, inserting an initial statement "assert(val +n <= INT_MAX)" into the code would be flawed since the left-hand side overflows just in those cases to be caught by the assert.

Upon termination, each element of a contains the sum of its index within a and the argument val.

7.1.2. Implementation of iota

Listing 7.2 shows an implementation of the iota function.

The loop invariant previous describes that in each iteration of the loop the current value val is equal to the sum of the value val in state of function entry and the loop index i (note the use of the \at operator here). This invariant is essential to prove the last invariant which is needed for the postcondition from our specification Listing 7.1.

[45] See http://www.sgi.com/tech/stl/iota.html.

```
void iota(value_type* a, size_type n, value_type val)
{
  /*@
    loop invariant bound:      0 <= i <= n;
    loop invariant increment: val == \at(val, Pre) + i;
    loop invariant previous:  \forall integer k; 0 <= k < i
                     ==> a[k] == \at(val, Pre) + k;

    loop assigns i, val, a[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i)
  {
    a[i] = val++;
  }
}
```

Listing 7.2: Implementation of `iota`

7.2. The `accumulate` algorithm

The `accumulate` algorithm in the C++ standard library computes the sum of an given initial value and the elements in a range. Our version of the original signature[46] reads:

```
value_type
accumulate(const value_type* a, size_type n, value_type init);
```

The result of `accumulate` equals the value

$$\mathtt{init} + \sum_{i=0}^{\mathtt{n}-1} \mathtt{a}[i]$$

thus, `accumulate` will return `init` for an empty range.

7.2.1. Axiomatic definition of accumulating over an array

As in the case of `count` (see Section 3.8) we specify `accumulate` by first defining a *logic function* `Accumulate` that formally defines the summation of elements in an array.

With this definition the following equation holds

$$\mathtt{Accumulate}(\mathtt{a}, \mathtt{n}+1, \mathtt{init}) = \mathtt{init} + \sum_{i=0}^{\mathtt{n}} \mathtt{a}[i] \tag{7.1}$$

We also provide an overloaded version of `Accumulate` that uses 0 for the omitted `init` argument. Thus, we have

$$\mathtt{Accumulate}(\mathtt{a}, \mathtt{n}+1) = \sum_{i=0}^{\mathtt{n}} \mathtt{a}[i] \tag{7.2}$$

[46]See `http://www.sgi.com/tech/stl/accumulate.html`.

```
/*@
   axiomatic AccumulateAxiomatic
   {
      logic integer Accumulate{L}(value_type* a, integer n,
                                    value_type init) reads a[0..n-1];

      axiom AccumulateEmpty: \forall value_type *a, init, integer n;
           n <= 0 ==> Accumulate(a, n, init) == init;

      axiom AccumulateNext: \forall value_type *a, init, integer n;
           n >= 0 ==> Accumulate(a, n + 1, init) ==
                        Accumulate(a, n, init) + a[n];

      axiom AccumulateRead{L1,L2}:
        \forall value_type *a, init, integer n;
           EqualRanges{L1,L2}(a, n) ==>
           Accumulate{L1}(a, n, init) == Accumulate{L2}(a, n, init);
   }

   // overloaded version
   logic integer Accumulate{L}(value_type* a, integer n) =
                 Accumulate{L}(a, n, (value_type) 0);
*/
```

Listing 7.3: The logic function `Accumulate`

In Listing 7.3, the `reads` clause, as well as Axiom `AccumulateRead`, guarantees that the result of `Accumulate` only depends on the values of `a[0..n-1]`.[47]

7.2.2. Formal specification of `accumulate`

Using the logic function `Accumulate`, the ACSL specification of `accumulate` is then as simple as in Listing 7.4.

Note that the property `bounds` formulates as a precondition that the *partial accumulations*

$$\mathtt{init} + \sum_{k=0}^{\mathtt{i}} \mathtt{a}[k] \tag{7.3}$$

do not overflow for $0 \leq i < n$. Otherwise, one cannot not guarantee that the result of `accumulate` equals the mathematical description of `Accumulate`.

[47]That is, the axiom is redundant. We stated it nevertheless since it sometime provides additional help to provers.

```
/*@
    requires valid:  \valid_read(a + (0..n-1));
    requires bounds: \forall integer i; 0 <= i <= n ==>
                       INT_MIN <= Accumulate(a, i, init) <= INT_MAX;

    assigns \nothing;

    ensures result:  \result == Accumulate(a, n, init);
*/
value_type
accumulate(const value_type* a, size_type n, value_type init);
```

Listing 7.4: Formal specification of `accumulate`

7.2.3. Implementation of `accumulate`

Listing 7.5 shows an implementation of the `accumulate` function with corresponding loop annotations. Note that loop invariant `partial` claims that in the *i*-th iterations step `result` equals the accumulated value of Equation (7.3).

```
value_type
accumulate(const value_type* a, size_type n, value_type init)
{
  value_type result = init;

  /*@
     loop invariant index:    0 <= i <= n;
     loop invariant partial:  result == Accumulate(a, i, init);
     loop assigns i, result;
     loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i)
  {
    result = result + a[i];
  }

  return result;
}
```

Listing 7.5: Implementation of `accumulate`

7.3. The `inner_product` algorithm

The `inner_product` algorithm in the C++ standard library computes the sum of a given initial value and the elements in a range. Our version of the original signature[48] reads:

```
value_type
inner_product(const value_type* a, const value_type* b,
              size_type n, value_type init);
```

The result of `inner_product` equals the value

$$\texttt{init} + \sum_{i=0}^{\texttt{n}-1} \texttt{a}[i] \cdot \texttt{b}[i]$$

thus, `inner_product` will return `init` for an empty range.

7.3.1. The logic function `InnerProduct`

As in the case of `accumulate` (see Section 7.2) we specify `inner_product` by first defining a *logic function* `InnerProduct` that formally defines the summation of the element-wise product of two arrays.

Axiom `InnerProductRead`, as well as the `reads` clause, guarantees that the result of the logic function `InnerProduct` only depends on the values of `a[0..n-1]` and `b[0..n-1]`.

7.3.2. Formal specification of `inner_product`

Using the logic function `InnerProduct`, we specify `inner_product` as shown in Listing 7.7.

Note that the properties labeled with `bounds` formulate as preconditions that neither the expressions

$$\texttt{a}[i] \cdot \texttt{b}[i] \tag{7.4}$$

nor

$$\texttt{init} + \sum_{k=0}^{\texttt{i}} \texttt{a}[k] \cdot \texttt{b}[k] \tag{7.5}$$

overflow for $0 \le i < n$. Otherwise, one cannot not guarantee that the result of `inner_product` equals the mathematical description of `InnerProduct`.

[48] See `http://www.sgi.com/tech/stl/inner_product.html`.

```
/*@
   axiomatic InnerProductAxiomatic
   {
     logic integer
     InnerProduct{L}(value_type* a, value_type* b, integer n,
                     value_type init) reads a[0..n-1], b[0..n-1];

     axiom InnerProductEmpty:
       \forall value_type *a, *b, init, integer n;
         n <= 0 ==> InnerProduct(a, b, n, init) == init;

     axiom InnerProductNext:
       \forall value_type *a, *b, init, integer n;
         n >= 0 ==> InnerProduct(a, b, n + 1, init) ==
                    InnerProduct(a, b, n, init) + (a[n] * b[n]);

     axiom InnerProductRead{L1,L2}:
       \forall value_type *a, *b, init, integer n;
         EqualRanges{L1,L2}(a, n) && EqualRanges{L1,L2}(b, n) ==>
           InnerProduct{L1}(a, b, n, init) ==
           InnerProduct{L2}(a, b, n, init);
   }

*/
```

Listing 7.6: The logic function InnerProduct

```
/*@
    requires valid_a: \valid_read(a + (0..n-1));
    requires valid_b: \valid_read(b + (0..n-1));
    requires bounds:  \forall integer i; 0 <= i < n ==>
                      INT_MIN <= a[i] * b[i] <= INT_MAX;
    requires bounds:  \forall integer i; 0 <= i <= n ==>
                      INT_MIN <= InnerProduct(a, b, i, init) <= INT_MAX;

    assigns \nothing;

    ensures result:    \result == InnerProduct(a, b, n, init);
    ensures unchanged: EqualRanges{Here,Pre}(a, n);
    ensures unchanged: EqualRanges{Here,Pre}(b, n);
*/
value_type
inner_product(const value_type* a, const value_type* b, size_type n,
              value_type init);
```

Listing 7.7: Formal specification of inner_product

7.3.3. Implementation of `inner_product`

Listing 7.8 shows an implementation of `inner_product` with corresponding loop annotations. Note that the loop invariant `partial` claims that in the *i*-th iterations step `result` equals the accumulated value of Equation (7.5).

```
value_type
inner_product(const value_type* a, const value_type* b, size_type n,
              value_type init)
{
  value_type result = init;

  /*@
     loop invariant index:   0 <= i <= n;
     loop invariant pre:     result == InnerProduct(a, b, i, init);
     loop assigns i, result;
     loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i)
  {
    result = result + a[i] * b[i];
  }

  return result;
}
```

Listing 7.8: Implementation of `inner_product`

7.4. The `partial_sum` algorithm

The `partial_sum` algorithm in the C++ standard library computes the sum of a given initial value and the elements in a range. Our version of the original signature[49] reads:

```
size_type
partial_sum(const value_type* a, size_type n, value_type* b);
```

After executing the function `partial_sum` the array `b[0..n-1]` holds the following values

$$
\begin{aligned}
\mathsf{b}[0] &= \mathsf{a}[0] \\
\mathsf{b}[1] &= \mathsf{a}[0] + \mathsf{a}[1] \\
&\ \vdots \\
\mathsf{b}[n-1] &= \mathsf{a}[0] + \mathsf{a}[1] + \ldots + \mathsf{a}[n-1]
\end{aligned}
$$

We can write this more concisely as

$$\mathsf{b}[i] = \sum_{k=0}^{\mathtt{i}} \mathsf{a}[k] \tag{7.6}$$

for $0 \leq i < n$.

7.4.1. The predicate `PartialSum`

Using Equations (7.2) and (7.6) we define the ACSL predicate `PartialSum` as shown in Listing 7.9.

```
/*@
  predicate
    PartialSum(value_type* a, integer n, value_type* b) =
      \forall integer i; 0 <= i < n ==>
        b[i] == Accumulate(a, i+1);
*/
```

Listing 7.9: The predicate `PartialSum`

The definition of `PartialSum` is another example of reusing specification artifacts.

[49] See `http://www.sgi.com/tech/stl/partial_sum.html`.

7.4.2. Formal specification of `partial_sum`

Using the predicate `PartialSum`, we specify `partial_sum` as shown in Listing 7.10.

```
/*@
  requires valid_in:   \valid_read(a + (0..n-1));
  requires valid_out:  \valid(b + (0..n-1));
  requires separated:  \separated(a + (0..n-1), b + (0..n-1));
  requires bounds:     \forall integer i; 0 <= i < n ==>
                       INT_MIN <= Accumulate(a, i) + a[i] <= INT_MAX;

  assigns b[0..n-1];

  ensures result:      \result == n;
  ensures partialsum:  PartialSum(a, n, b);
  ensures unchanged:   EqualRanges{Here,Pre}(a, n, a);
*/
size_type
partial_sum(const value_type* a, size_type n, value_type* b);
```

Listing 7.10: Formal specification of `partial_sum`

Note that we require that the arrays `a[0..n-1]` and `b[0..n-1]` are separated, that is, they do not overlap. This is not required in the informal specification of `partial_sum` in the C++ standard library.

The property labeled with `bounds` formulates as preconditions that the expression

$$\sum_{k=0}^{\mathtt{i}} \mathsf{a}[k]$$

which is here written as

$$\sum_{k=0}^{\mathtt{i-1}} \mathsf{a}[k] + \mathsf{a}[i] \tag{7.7}$$

does not overflow for $0 \le i < n$.

7.4.3. Implementation of `partial_sum`

Listing 7.11 shows an implementation of `partial_sum` with corresponding loop annotations.

```
size_type
partial_sum(const value_type* a, size_type n, value_type* b)
{
  if (n > 0)
  {
    b[0] = a[0];

    /*@
       loop invariant index:      1 <= i <= n;
       loop invariant unchanged: EqualRanges{Here,Pre}(a, n, a);
       loop invariant previous:  PartialSum(a, i, b);
       loop assigns i, b[1..n-1];
       loop variant n - i;
    */
    for (size_type i = 1; i < n; ++i)
    {
      b[i] = b[i - 1] + a[i];
    }
  }

  return n;
}
```

Listing 7.11: Implementation of `partial_sum`

The somewhat awkward formulation of the precondition `bounds` in Listing 7.10 (see also Equation (7.7)) allows us to automatically verify that there are no overflows in the expression

```
b[i] = b[i - 1] + a[i];
```

in the loop body of Listing 7.11.

7.5. The `adjacent_difference` algorithm

The `adjacent_difference` algorithm in the C++ standard library computes the sum of a given initial value and the elements in a range. Our version of the original signature[50] reads:

```
size_type
adjacent_difference(const value_type* a, size_type n, value_type* b);
```

After executing the function `adjacent_difference` the array `b[0..n-1]` holds the following values

$$\begin{aligned} \mathsf{b}[0] &= \mathsf{a}[0] \\ \mathsf{b}[1] &= \mathsf{a}[1] - \mathsf{a}[0] \\ &\vdots \\ \mathsf{b}[n-1] &= \mathsf{a}[n-1] - \mathsf{a}[n-2] \end{aligned}$$

If we form the partial sums of the sequence `b` we find that

$$\begin{aligned} \mathsf{b}[0] &= \mathsf{a}[0] \\ \mathsf{b}[1] + \mathsf{b}[0] &= \mathsf{a}[1] \\ &\vdots \\ \mathsf{b}[0] + \mathsf{b}[1] + \ldots + \mathsf{b}[n-1] &= \mathsf{a}[n-1] \end{aligned}$$

Thus, we have for $0 \leq i < n$

$$\sum_{k=0}^{\mathtt{i}} \mathsf{b}[k] = \mathsf{a}[i] \tag{7.8}$$

which is nicely related to Equation (7.6).

7.5.1. Formal specification of `adjacent_difference`

Equation 7.8 justifies that we can re-use predicate `PartialSum` (Listing 7.9) to formally specify `adjacent_difference`.

Note that the only difference between the specifications 7.12 and 7.10 are

- the formulation of the respective preconditions `bounds`
- and the order of arguments in the respective postconditions `partialsum`.

As in the case of the specification of `partial_sum` we require that the arrays `a[0..n-1]` and `b[0..n-1]` are separated.

[50] See `http://www.sgi.com/tech/stl/adjacent_difference.html`.

```
/*@
   requires valid_in:   \valid_read(a + (0..n-1));
   requires valid_out:  \valid(b + (0..n-1));
   requires separated:  \separated(a + (0..n-1), b + (0..n-1));
   requires bounds:     \forall integer i; 1 <= i < n ==>
                        INT_MIN <= a[i] - a[i-1] <= INT_MAX;

   assigns b[0..n-1];

   ensures result:      \result == n;
   ensures partialsum:  PartialSum(b, n, a);
   ensures unchanged:   EqualRanges{Here,Pre}(a, n);
*/
size_type
adjacent_difference(const value_type* a, size_type n, value_type* b);
```

Listing 7.12: Formal specification of `adjacent_difference`

7.5.2. Implementation of `adjacent_difference`

Listing 7.13 shows an implementation of `adjacent_difference` with corresponding loop annotations.

```
size_type
adjacent_difference(const value_type* a, size_type n, value_type* b)
{
  if (n > 0)
  {
    b[0] = a[0];

    /*@
       loop invariant index:     1 <= i <= n;
       loop invariant unchanged: EqualRanges{Here,Pre}(a, n);
       loop invariant previous:  PartialSum(b, i, a);
       loop assigns i, b[1..n-1];
       loop variant n - i;
    */
    for (size_type i = 1u; i < n; ++i)
    {
      b[i] = a[i] - a[i - 1u];
    }
  }

  return n;
}
```

Listing 7.13: Implementation of `adjacent_difference`

As in the case of the specification of `adjacent_difference` we would like to emphasize the strong similarities to the loop annotations of `partial_sum` (Listing 7.11).

8. The `Stack` data type

Originally, ACSL is tailored to the task of specifying and verifying one single C function at a time. However, in practice we are also faced with the task to implement a family of functions, usually around some sophisticated data structure, which have to obey certain rules of interdependence. In this kind of task, we are not interested in the properties of a single function (usually called "*implementation details*"), but in properties describing how several function play together (usually called "*abstract interface description*", or "*abstract data type properties*").

This chapter introduces a methodology to formally denote and verify the latter property sets using ACSL. For a more detailed discussion of our approach to the formal verification of `Stack` we refer to this thesis [14].

A *stack* is a data type that can hold objects and has the property that, if an object *a* is *pushed* on a stack *before* object *b*, then *a* can only be removed (*popped*) after *b*. A stack is, in other words, a *first-in, last-out* data type (see Figure 8.1). The *top* function of a stack returns the last element that has been pushed on a stack.

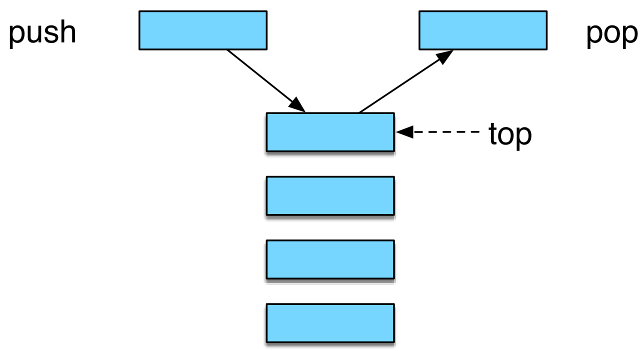


Figure 8.1.: Push and pop on a stack

We consider only stacks that have a finite *capacity*, that is, that can only hold a maximum number *c* of elements that is constant throughout their lifetime. This restriction allows us to define a stack without relying on dynamic memory allocation. When a stack is *created* or *initialized*, it contains no elements, i.e., its *size* is 0. The function *push* and *pop* increases and decreases the size of a stack by at most one, respectively.

8.1. Methodology overview

Figure 8.2 gives an overview of our methodology to specify and verify abstract data types (verification of one axiom shown only).

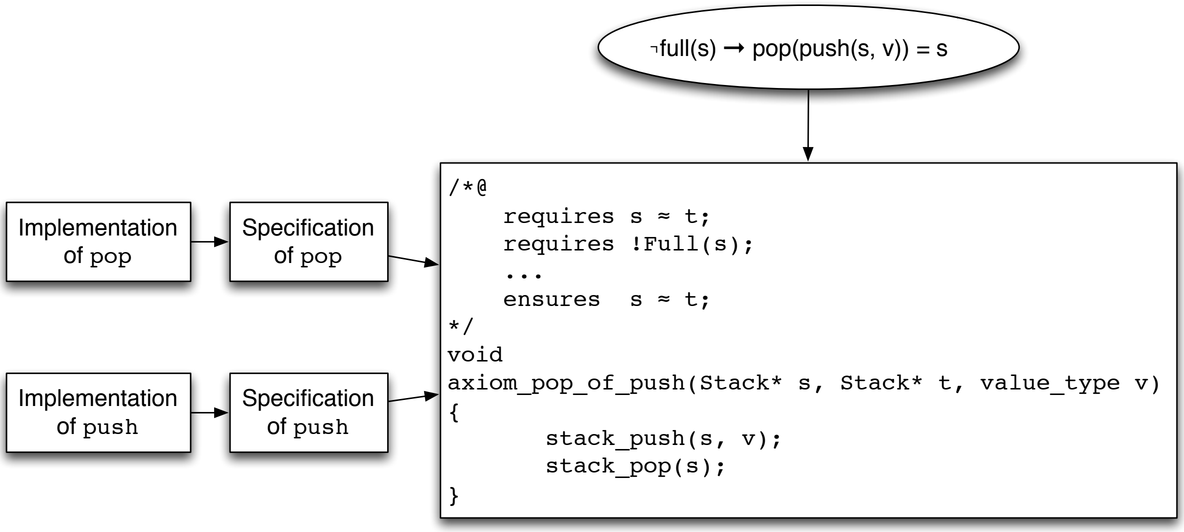


Figure 8.2.: Methodology Overview

What we will basically do is:

1. specify axioms about how the stack functions should interact with each other (Section 8.2),
2. define a basic implementation of C data structures (only one in our example, viz. `struct Stack`; see Section 8.3) and some invariants the instances of them have to obey (Section 8.4),
3. provide for each stack function an ACSL contract and a C implementation (Section 8.7),
4. verify each function against its contract (Section 8.7),
5. transform the axioms into ACSL-annotated C code (Section 8.8), and
6. verify that code, using access function contracts and data-type invariants as necessary (Section 8.8).

Section 8.5 provides an ACSL-predicate deciding whether two instances of a `struct Stack` are considered to be equal (indication by "≈" in Figure 8.2), while Section 8.6 gives a corresponding C implementation. The issue of an appropriate definition of equality of data instances is familiar to any C programmer who had to replace a faulty comparison `if(s1 == s2)` by the correct `if(strcmp(s1,s2)== 0)` to compare two strings `char *s1,*s2` for equality.

8.2. Stack axioms

To specify the interplay of the stack access functions, we use a set of axioms[51], all but one of them having the form of a conditional equation.

Let V denote an arbitrary type. We denote by S_c the type of stacks with capacity $c > 0$ of elements of type V. The aforementioned functions then have the following signatures.

$$\begin{aligned}
\text{init} &: S_c \to S_c, \\
\text{push} &: S_c \times V \to S_c, \\
\text{pop} &: S_c \to S_c, \\
\text{top} &: S_c \to V, \\
\text{size} &: S_c \to \mathbb{N}.
\end{aligned}$$

With $\mathbb{B}$ denoting the *boolean* type we will also define two auxiliary functions

$$\begin{aligned}
\text{empty} &: S_c \to \mathbb{B}, \\
\text{full} &: S_c \to \mathbb{B}.
\end{aligned}$$

To qualify as a stack these functions must satisfy the following rules which are also referred to as *stack axioms*.

8.2.1. Stack initialization

After a stack has been initialized its size is 0.

$$\text{size}(\text{init}(s)) = 0. \tag{8.1}$$

The auxiliary functions empty and full are defined as follows

$$\text{empty}(s), \qquad \text{iff} \qquad \text{size(s)} = 0, \tag{8.2}$$
$$\text{full}(s), \qquad \text{iff} \qquad \text{size(s)} = c. \tag{8.3}$$

We expect that for every stack s the following condition holds

$$0 \leq \text{size}(s) \leq c. \tag{8.4}$$

[51] There is an analogy in geometry: Euclid (e.g. [15]) invented the use of axioms there, but still kept definitions of *point*, *line*, *plane*, etc. Hilbert [16] recognized that the latter are not only unformalizable, but also unnecessary, and dropped them, keeping only the formal descriptions of relations between them.

8.2.2. Adding an element to a stack

To push an element v on a stack the stack must not be full. If an element has been pushed on an eligible stack, its size increases by 1

$$\text{size}(\text{push}(s, v)) = \text{size}(s) + 1, \qquad \text{if} \quad \neg\text{full}(s). \tag{8.5}$$

Moreover, the element pushed on a stack is the top element of the resulting stack

$$\text{top}(\text{push}(s, v)) = v, \qquad \text{if} \quad \neg\text{full}(s). \tag{8.6}$$

8.2.3. Removing an element from a stack

An element can only be removed from a non-empty stack. If an element has been removed from an eligible stack the stack size decreases by 1

$$\text{size}(\text{pop}(s)) = \text{size}(s) - 1, \qquad \text{if} \quad \neg\text{empty}(s). \tag{8.7}$$

If an element is pushed on a stack and immediately afterwards an element is removed from the resulting stack then the final stack is equal to the original stack

$$\text{pop}(\text{push}(s, v)) = s, \qquad \text{if} \quad \neg\text{full}(s). \tag{8.8}$$

Conversely, if an element is removed from a non-empty stack and if afterwards the top element of the original stack is pushed on the new stack then the resulting stack is equal to the original stack.

$$\text{push}(\text{pop}(s), \text{top}(s)) = s, \qquad \text{if} \quad \neg\text{empty}(s). \tag{8.9}$$

8.2.4. A note on exception handling

We don't impose a requirement on `push(s, v)` if `s` is a full stack, nor on `pop(s)` or `top(s)` if `s` is an empty stack. Specifying the behavior in such *exceptional* situations is a problem by its own; a variety of approaches is discussed in the literature. We won't elaborate further on this issue, but only give an example to warn about "innocent-looking" exception specifications that may lead to undesired results.

If we'd introduce an additional error value `err` in the element type V and require `top(s)= err` if `s` is empty, we'd be faced with the problem of specifying the behavior of `push(s, err)`. At first glance, it would seem a good idea to have `err` just been ignored by `push`, i.e. to require

$$\text{push}(s, \text{err}) = s. \tag{8.10}$$

However, we then could derive for any non-full and non-empty stack `s`, that

$$\begin{aligned} \text{size}(s) &= \text{size}(\text{pop}(\text{push}(s, \text{err}))) && \text{by 8.8} \\ &= \text{size}(\text{pop}(s)) && \text{as assumed in 8.10} \\ &= \text{size}(s) - 1 && \text{by 8.7} \end{aligned}$$

i.e. no such stacks could exist, or all `int` values would be equal.

8.3. The structure `Stack` and its associated functions

We now introduce one possible C implementation of the above axioms. It is centred around the C structure `Stack` shown in Listing 8.3.

```
struct Stack
{
  value_type* obj;

  size_type   capacity;

  size_type   size;
};

typedef struct Stack Stack;
```

Listing 8.3: Definition of type `Stack`

This struct holds an array `obj` of positive length called `capacity`. The capacity of a stack is the maximum number of elements this stack can hold. The field `size` indicates the number elements that are currently in the stack. See also Figure 8.4 which attempts to interpret this definition according to Figure 8.1.

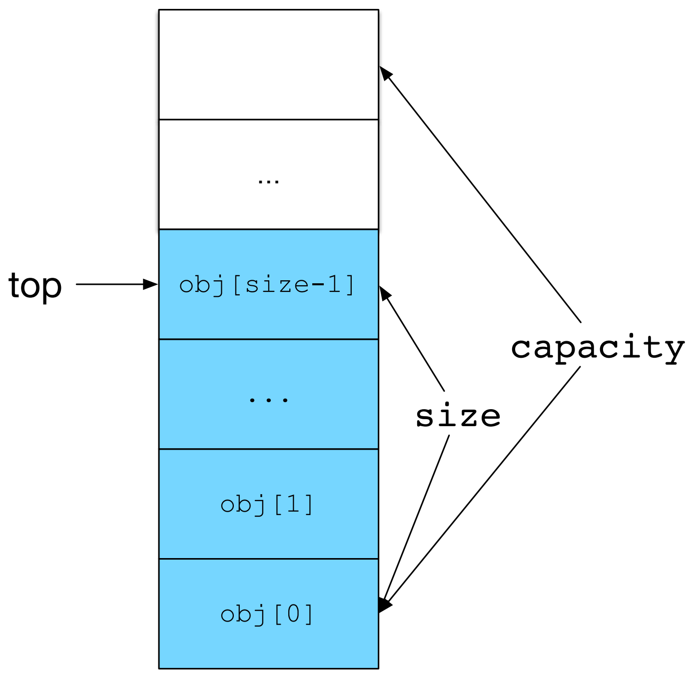


Figure 8.4.: Interpreting the data structure `Stack`

Based on the stack functions from Section 8.2, we declare in Listing 8.5 the following functions as part of our Stack data type.

```
void         stack_init(Stack* s, value_type* a, size_type n);

bool         stack_equal(const Stack* s, const Stack* t);

size_type    stack_size(const Stack* s);

bool         stack_empty(const Stack* s);

bool         stack_full(const Stack* s);

value_type   stack_top(const Stack* s);

void         stack_push(Stack* s, value_type v);

void         stack_pop(Stack* s);
```

Listing 8.5: Declaration of functions of type Stack

Most of these functions directly correspond to methods of the C++ std::stack template class.[52] The function stack_equal corresponds to the comparison operator ==, whereas one use of stack_init is to bring a stack into a well-defined initial state. The function stack_full has no counterpart in std::stack. This reflects the fact that we avoid dynamic memory allocation, while std::stack does not.

8.4. Stack invariants

Not every possible instance of type Stack is considered a valid one, e.g., with our definition of Stack in Listing 8.3, Stack s = {{0,0,0,0},4,5} is not. Below, we will define an ACSL-predicate Valid that discriminates valid and invalid instances.

Before, we introduce in Listing 8.6 the auxiliary logical function Capacity and Size which we can use in specifications to refer to the fields capacity and size of Stack, respectively. This listing also contains the logical function Top which defines the array element with index size-1 as the top place of a stack. The reader can consider this as an attempt to hide implementation details from the specification.

[52]See http://www.sgi.com/tech/stl/stack.html

```
//@  logic size_type Capacity{L}(Stack* s) = s->capacity;

//@  logic size_type Size{L}(Stack* s) = s->size;

//@  logic value_type* Storage{L}(Stack* s) = s->obj;

//@  logic value_type Top{L}(Stack* s) = s->obj[s->size-1];
```

Listing 8.6: The logical functions `Capacity`, `Size` and `Top`

We also introduce in Listing 8.7 two predicates that express the concepts of empty and full stacks by referring to a stack's size and capacity (see Equations (8.2) and (8.3)).

```
//@  predicate Empty{L}(Stack* s) =  Size(s) == 0;

//@  predicate Full{L}(Stack* s)  =  Size(s) == Capacity(s);
```

Listing 8.7: Predicates for empty an full stacks

There are some obvious invariants that must be fulfilled by every valid object of type `Stack`:

- The stack capacity shall be strictly greater than zero (an empty stack is ok but a stack that cannot hold anything is not useful).
- The pointer `obj` shall refer to an array of length `capacity`.
- The number of elements `size` of a stack the must be non-negative and not greater than its capacity.

These invariants are formalized in the predicate `Valid` of Listing 8.8.

```
/*@
  predicate Valid{L}(Stack* s) =
    \valid(s) &&
    0 < Capacity(s) &&
    0 <= Size(s) <= Capacity(s) &&
    \valid(Storage(s) + (0..Capacity(s)-1)) &&
    \separated(s, Storage(s) + (0..Capacity(s)-1));
*/
```

Listing 8.8: The predicate `Valid`

Note how the use of the previously defined logical functions and predicates allows us to define the stack invariant without directly referring to the fields of `Stack`. As we usually have to deal with a pointer `s` of type `Stack` we add the necessary `\valid(s)` to the predicate `Valid`.

8.5. Equality of stacks

Defining equality of instances of non-trivial data types, in particular in object-oriented languages, is not an easy task. The book *Programming in Scala*[17, Chapter 28] devotes to this topic a whole chapter of more than twenty pages. In the following two sections we give a few hints how ACSL and Frama-C can help to correctly define equality for a simple data type.

We consider two stacks as equal if they have the same size and if they contain the same objects. To be more precise, let `s` and `t` two pointers of type `Stack`, then we define the predicate `Equal` as in Listing 8.9.

```
/*@
   predicate Equal{S,T}(Stack* s, Stack* t) =
        Size{S}(s) == Size{T}(t) &&
        EqualRanges{S,T}(Storage{S}(s), Size{S}(s), Storage{T}(t));
*/
```

Listing 8.9: Equality of stacks

Our use of labels in Listing 8.9 makes the specification somewhat hard to read (in particular in the last line where we reuse the predicate `EqualRanges` from Page 28). However, this definition of `Equal` will allow us later to compare the same stack object at different points of a program. The logical expression `Equal{A,B}(s,t)` reads informally as: The stack object `*s` at program point `A` equals the stack object `*t` at program point `B`.

The reader might wonder why we exclude the capacity of a stack into the definition of stack equality. This approach can be motivated with the behavior of the method `capacity` of the class `std::vector<T>`. There, equal instances of type `std::vector<T>` may very well have different capacities.[53]

If equal stacks can have different capacities then, according to our definition of the predicate `Full` in Listing 8.7, we can have to equal stacks where one is full and the other one is not.

A finer, but very important point in our specification of equality of stacks is that the elements of the arrays `s->obj` and `t->obj` are compared only up to `s->size` and *not* up to `s->capacity`. Thus the two stacks `s` and `t` in Figure 8.10 are considered equal although there is are obvious differences in their internal arrays.

Figure 8.10.: Example of two equal stacks

[53] See `http://www.cplusplus.com/reference/vector/vector/capacity`

If we define an equality relation (=) of objects for a data type such as `Stack`, we have to make sure that the following rules hold.

$$\begin{aligned} &\text{reflexivity} && \forall s \in S : s = s, && (8.11a)\\ &\text{symmetry} && \forall s, t \in S : s = t \implies t = s, && (8.11b)\\ &\text{transitivity} && \forall s, t, u \in S : s = t \wedge t = u \implies s = u. && (8.11c) \end{aligned}$$

Any relation that satisfies the conditions (8.11) is referred to as an *equivalence relation*. The mathematical set of all instances that are considered equal to some given instance `s` is called the equivalence class of `s` with respect to that relation.

Listing 8.11 shows a formalization of these three rules for the relation `Equal`; it can be automatically verified that they are a consequence of the definition of `Equal` in Listing 8.9.

```
/*@
  lemma StackEqualReflexive{S} :
    \forall Stack* s; Equal{S,S}(s, s);

  lemma StackEqualSymmetric{S,T} :
    \forall Stack *s, *t;
      Equal{S,T}(s, t) ==> Equal{T,S}(t, s);

  lemma StackEqualTransitive{S,T,U}:
    \forall Stack *s, *t, *u;
      Equal{S,T}(s, t) && Equal{T,U}(t, u) ==> Equal{S,U}(s, u);
*/
```

Listing 8.11: Equality of stacks is an equivalence relation

The two stacks in Figure 8.10 show that an equivalence class of `Equal` can contain more than one element.[54] The stacks `s` and `t` in Figure 8.10 are also referred to as two *representatives* of the same equivalence class. In such a situation, the question arises whether a function that is defined on a set with an equivalence relation can be defined in such a way that its definition is *independent of the chosen representatives*.[55] We ask, in other words, whether the function is *well-defined* on the set of all equivalence classes of the relation `Equal`.[56] The question of well-definition will play an important role when verifying the functions of the `Stack` (see Section 8.7).

[54]This is a common situation in mathematics. For example, the equivalence class of the rational number $\frac{1}{2}$ contains infinitely many elements, viz. $\frac{1}{2}, \frac{2}{4}, \frac{7}{14}, \ldots$.

[55]This is why mathematicians have to *prove* that $\frac{1}{2} + \frac{3}{5}$ equals $\frac{7}{14} + \frac{3}{5}$.

[56]See `http://en.wikipedia.org/wiki/Well-definition`.

8.6. Runtime equality of stacks

The function `stack_equal` is the C equivalent for the `Equal` predicate. The specification of `stack_equal` is shown in Listing 8.12. Note that this specifications explicitly refers to valid stacks.

```
/*@
    requires Valid(s);
    requires Valid(t);

    assigns \nothing;

    ensures \result == 1  <==>  Equal{Here,Here}(s, t);
    ensures \result == 0  <==> !Equal{Here,Here}(s, t);
*/
bool stack_equal(const Stack* s, const Stack* t);
```

Listing 8.12: Specification of `stack_equal`

The implementation of `stack_equal` in Listing 8.13 compares two stacks according to the same rules of predicate `Equal`.

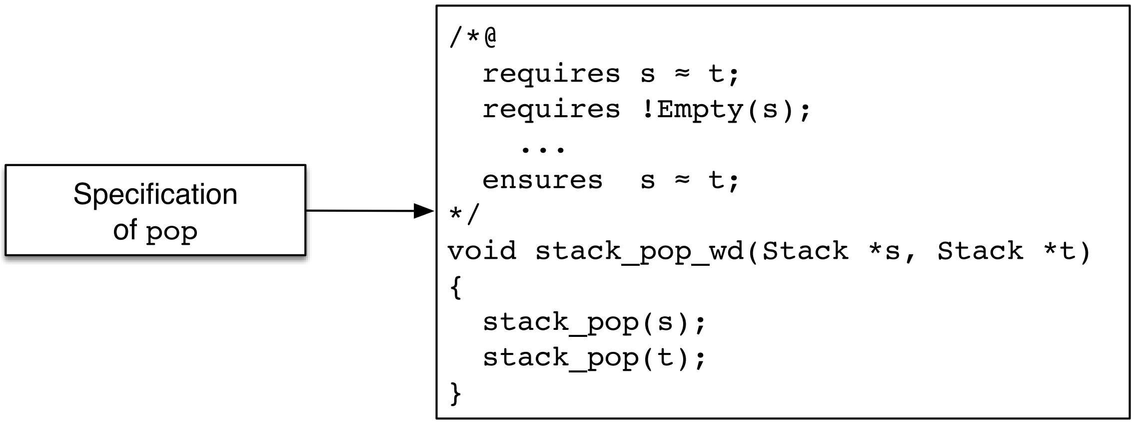
```
bool stack_equal(const Stack* s, const Stack* t)
{
  return (s->size == t->size) && equal(s->obj, s->size, t->obj);
}
```

Listing 8.13: Implementation of `stack_equal`

8.7. Verification of stack functions

In this section we verify the functions `stack_init` (Section 8.7.1), `stack_size` (Section 8.7.2), `stack_empty` (Section 8.7.3), `stack_full` (Section 8.7.4), `stack_top` (Section 8.7.5), and `stack_push` (Section 8.7.6) `stack_pop` (Section 8.7.7), of the data type `Stack`. To be more precise, we provide for each of function `stack_foo`:

- an ACSL specification of `stack_foo`
- a C implementation of `stack_foo`
- a C function `stack_foo_wd`[57] accompanied by a an ACSL contract that expresses that the implementation of `stack_foo` is well-defined. Figure 8.14 shows our methodology for the verification of well-definition in the `pop` example, (≈) again indicating the user-defined `Stack` equality.




```
/*@
  requires s ≈ t;
  requires !Empty(s);
    ...
  ensures  s ≈ t;
*/
void stack_pop_wd(Stack *s, Stack *t)
{
  stack_pop(s);
  stack_pop(t);
}
```

Figure 8.14.: Methodology for the verification of well-definition

Note that the specifications of the various functions will explicitly refer to the *internal state* of `Stack`. In Section 8.8 we will show that the *interplay* of these functions satisfy the stack axioms from Section 8.2.

[57]The suffix `_wd` stands for *well definition*.

8.7.1. The function `stack_init`

Listing 8.15 shows the ACSL specification of `stack_init`. Note that our specification of the post-conditions contains a redundancy because a stack is empty if and only if its size is zero.

```
/*@
  requires \valid(s);
  requires 0 < capacity;
  requires \valid(storage + (0..capacity-1));
  requires \separated(s, storage + (0..capacity-1));

  assigns s->obj;
  assigns s->capacity;
  assigns s->size;

  ensures Valid(s);
  ensures Capacity(s) == capacity;
  ensures Size(s) == 0;
  ensures Empty(s);
  ensures Storage(s) == storage;
*/
void stack_init(Stack* s, value_type* storage, size_type capacity);
```

Listing 8.15: Specification of `stack_init`

Listing 8.15 shows the implementation of `stack_init`. It simply initializes `obj` and `capacity` with the respective value of the array and sets the field `size` to zero.

```
void stack_init(Stack* s, value_type* storage, size_type capacity)
{
  s->obj      = storage;
  s->capacity = capacity;
  s->size     = 0;
}
```

Listing 8.16: Implementation of `stack_init`

8.7.2. The function stack_size

The function stack_size is the runtime version of the logical function Size from Listing 8.6 on Page 115. The specification of stack_size in Listing 8.17 simply states that stack_size produces the same result as Size.

```
/*@
    requires Valid(s);

    assigns \nothing;

    ensures \result == Size(s);
*/
size_type stack_size(const Stack* s);
```

Listing 8.17: Specification of stack_size

As in the definition of the logical function Size the implementation of stack_size in Figure 8.18 simply returns the field size.

```
size_type stack_size(const Stack* s)
{
  return s->size;
}
```

Listing 8.18: Implementation of stack_size

Listing 8.19 shows our check whether stack_size is well-defined. Since stack_size neither modifies the state of its Stack argument nor that of any global variable we only check whether it produces the same result for equal stacks. Note that we simply may use operator == to compare integers since we didn't introduce a nontrivial equivalence relation on that data type.

```
/*@
  requires Valid(s) && Valid(t);
  requires Equal{Here,Here}(s, t);

  assigns \nothing;

  ensures \result;
*/
bool stack_size_wd(const Stack* s, const Stack* t)
{
  return stack_size(s) == stack_size(t);
}
```

Listing 8.19: Well-definition of stack_size

8.7.3. The function stack_empty

The function stack_empty is the runtime version of the predicate Empty from Listing 8.7 on Page 115.

```
/*@
    requires Valid(s);

    assigns \nothing;

    ensures \result == 1 <==>  Empty(s);
    ensures \result == 0 <==> !Empty(s);
*/
bool stack_empty(const Stack* s);
```

Listing 8.20: Specification of stack_empty

As in the definition of the predicate Empty the implementation of stack_empty in Figure 8.21 simply checks whether the size of the stack is zero.

```
bool stack_empty(const Stack* s)
{
  return stack_size(s) == 0;
}
```

Listing 8.21: Implementation of stack_empty

Listing 8.22 shows our check whether stack_empty is well-defined.

```
/*@
  requires Valid(s);
  requires Valid(t);
  requires Equal{Here,Here}(s, t);

  assigns \nothing;

  ensures \result;
*/
bool stack_empty_wd(const Stack* s, const Stack* t)
{
  return stack_empty(s) == stack_empty(t);
}
```

Listing 8.22: Well-definition of stack_empty

8.7.4. The function stack_full

The function stack_full is the runtime version of the predicate Full from Listing 8.7 on Page 115.

```
/*@
    requires Valid(s);

    assigns \nothing;

    ensures \result == 1 <==>  Full(s);
    ensures \result == 0 <==> !Full(s);
*/
bool stack_full(const Stack* s);
```

Listing 8.23: Specification of stack_full

As in the definition of the predicate Full the implementation of stack_full in Figure 8.24 simply checks whether the size of the stack equals its capacity.

```
bool stack_full(const Stack* s)
{
  return stack_size(s) == s->capacity;
}
```

Listing 8.24: Implementation of stack_full

Note that with our definition of stack equality (Section 8.5) there can be equal stack with different capacities. Accordingly, there can exist equal stacks where one is full while the other is not.

8.7.5. The function stack_top

The function stack_top is the runtime version of the logical function Top from Listing 8.6 on Page 115. The specification of stack_top in Listing 8.25 simply states that for non-empty stacks stack_top produces the same result as Top which in turn just returns the element obj[size-1] of Stack.

```
/*@
    requires Valid(s);

    assigns \nothing;

    ensures !Empty(s) ==> \result == Top(s);
*/
value_type stack_top(const Stack* s);
```

Listing 8.25: Specification of stack_top

For a non-empty stack the implementation of stack_top in Figure 8.26 simply returns the element obj[size-1]. Note that our implementation of stack_top does not crash when it is applied to an empty stack. In this case we return the first element of the internal, non-empty array obj. This is consistent with our specification of stack_top which only refers to non-empty stacks.

```
value_type stack_top(const Stack* s)
{
  if (!stack_empty(s))
  {
    return s->obj[s->size - 1];
  }
  else
  {
    return s->obj[0];
  }
}
```

Listing 8.26: Implementation of stack_top

Listing 8.27 shows our check whether `stack_top` well-defined for non-empty stacks.

```
/*@
  requires Valid(s) && !Empty(s);
  requires Valid(t) && !Empty(t);
  requires Equal{Here,Here}(s, t);

  assigns \nothing;

  ensures \result;
*/
bool stack_top_wd(const Stack* s, const Stack* t)
{
  return stack_top(s) == stack_top(t);
}
```

Listing 8.27: Well-definition of `stack_top`

Since our axioms in Section 8.2 did not impose any behavior on the behavior of `stack_top` for empty stacks, we prove the well-definition of `stack_top` only for nonempty stacks.

8.7.6. The function stack_push

Listing 8.28 shows the ACSL specification of the function stack_push. In accordance with Axiom (8.5), stack_push is supposed to increase the number of elements of a non-full stack by one. The specification also demands that the value that is pushed on a non-full stack becomes the top element of the resulting stack (see Axiom (8.6)).

```
/*@
  requires Valid(s);

  assigns s->size;
  assigns s->obj[s->size];

  behavior not_full:
    assumes !Full(s);

    assigns s->size;
    assigns s->obj[s->size];

    ensures Valid(s);
    ensures Size(s) == Size{Old}(s) + 1;
    ensures Top(s) == v;
    ensures !Empty(s);
    ensures Unchanged{Pre,Here}(Storage(s), 0, Size{Pre}(s));
    ensures Storage(s) == Storage{Old}(s);
    ensures Capacity(s) == Capacity{Old}(s);

  behavior full:
    assumes Full(s);

    assigns \nothing;

    ensures Valid(s);
    ensures Full(s);
    ensures Unchanged{Pre,Here}(Storage(s), 0, Size(s));
    ensures Size(s) == Size{Old}(s);
    ensures Storage(s) == Storage{Old}(s);
    ensures Capacity(s) == Capacity{Old}(s);

  complete behaviors;
  disjoint behaviors;
*/
void stack_push(Stack* s, value_type v);
```

Listing 8.28: Specification of stack_push

The implementation of `stack_push` is shown in Listing 8.29. It checks whether its argument is a non-full stack in which case it increases the field `size` by one but only after it has assigned the function argument to the element `obj[size]`.

```
void stack_push(Stack* s, value_type v)
{
  if (!stack_full(s))
  {
    s->obj[s->size++] = v;
  }
}
```

Listing 8.29: Implementation of `stack_push`

`stack_push` does not return a value but rather modifies its argument. For the well-definition of `stack_push` we therefore check whether it turns equal stacks into equal stacks. However, equality of the stack arguments is not sufficient for a proof that `stack_push` is well-defined. We must also ensure that there is no *aliasing* between the two stacks. Otherwise modifying one stack could modify the other stack in unexpected ways. In order to express that there is no aliasing between two stacks, we define in Listing 8.30 the predicate `Separated`.

```
/*@
   predicate Separated(Stack* s, Stack* t) =
      \separated(s, s->obj + (0..s->capacity-1),
                 t, t->obj + (0..t->capacity-1));
*/
```

Listing 8.30: The predicate `Separated`

Listing 8.31 shows our formalization of well-definition for `stack_push`.

```
/*@
  requires Valid(s) && Valid(t);
  requires Equal{Here,Here}(s, t);
  requires !Full(s) && !Full(t);
  requires Separated(s, t);

  ensures Valid(s) && Valid(t);
  ensures Equal{Here,Here}(s, t);
*/
void stack_push_wd(Stack* s, Stack* t, value_type v)
{
  stack_push(s, v);
  stack_push(t, v);
}
```

Listing 8.31: Well-definition of `stack_push`

8.7.7. The function `stack_pop`

Listing 8.32 shows the ACSL specification of the function `stack_pop`. In accordance with Axiom (8.7) `stack_pop` is supposed to reduce the number of elements in a non-empty stack by one. In addition to the requirements imposed by the axioms, our specification demands that `stack_pop` changes no memory location if it is applied to an empty stack.

```
/*@
  requires Valid(s);

  assigns s->size;

  ensures  Valid(s);

  behavior not_empty:
    assumes !Empty(s);

    assigns s->size;

    ensures Size(s) == Size{Old}(s) - 1;
    ensures !Full(s);
    ensures Unchanged{Pre,Here}(Storage(s), 0, Size(s));
    ensures Storage(s) == Storage{Old}(s);
    ensures Capacity(s) == Capacity{Old}(s);

  behavior empty:
    assumes Empty(s);

    assigns \nothing;

    ensures Empty(s);
    ensures Unchanged{Pre,Here}(Storage(s), 0, Size(s));
    ensures Size(s) == Size{Old}(s);
    ensures Storage(s) == Storage{Old}(s);
    ensures Capacity(s) == Capacity{Old}(s);

  complete behaviors;
  disjoint behaviors;
*/
void stack_pop(Stack* s);
```

Listing 8.32: Specification of `stack_pop`

The implementation of `stack_pop` is shown in Listing 8.33. It checks whether its argument is a non-empty stack in which case it decreases the field `size` by one.

```
void stack_pop(Stack* s)
{
  if (!stack_empty(s))
  {
    --s->size;
  }
}
```

Listing 8.33: Implementation of `stack_pop`

Listing 8.34 shows our check whether `stack_pop` is well-defined. As in the case of `stack_push` we use the predicate `Separated` (Listing 8.30) in order to express that there is no aliasing between the two stack arguments.

```
/*@
  requires Valid(s);
  requires Valid(t);
  requires Equal{Here,Here}(s, t);
  requires Separated(s, t);

  assigns s->size;
  assigns t->size;

  ensures Valid(s);
  ensures Valid(t);
  ensures Equal{Here,Here}(s, t);
*/
void stack_pop_wd(Stack* s, Stack* t)
{
  stack_pop(s);
  stack_pop(t);
}
```

Listing 8.34: Well-definition of `stack_pop`

8.8. Verification of stack axioms

In this section we show that the stack functions defined in Section 8.7 satisfy the stack Axioms of Section 8.2.

The annotated code has been obtained from the axioms in a fully systematical way. In order to transform a condition equation $p \to s = t$:

- Generate a clause `requires p`.
- Generate a clause `requires x1 == ... == xn` for each variable `x` with n occurrences in s and t.
- Change the i-th occurrence of `x` to `xi` in s and t.
- Translate both terms s and t to reversed polish notation.
- Generate a clause `ensures y1 == y2`, where `y1` and `y2` denote the value corresponding to the translated s and t, respectively.

This makes it easy to implement a tool that does the translation automatically, but yields a slightly longer contract in our example.

8.8.1. Resetting a stack

Our formulation in ACSL/C of the Axiom in Equation (8.1) on Page 111 is shown in Listing 8.35.

```
/*@
  requires \valid(s);
  requires 0 < n;
  requires \valid(a + (0..n-1));
  requires \separated(s, a + (0..n-1));

  assigns s->obj, s->capacity, s->size;

  ensures Valid(s);
  ensures \result == 0;
*/
size_type axiom_size_of_init(Stack* s, value_type* a, size_type n)
{
  stack_init(s, a, n);
  return stack_size(s);
}
```

Listing 8.35: Specification of Axiom (8.1)

8.8.2. Adding an element to a stack

Axioms (8.5) and (8.6) describe the behavior of a stack when an element is added.

```
/*@
  requires Valid(s);
  requires !Full(s);

  assigns s->size;
  assigns s->obj[s->size];

  ensures Valid(s);
  ensures \result == Size{Old}(s) + 1;
*/
size_type axiom_size_of_push(Stack* s, value_type v)
{
  stack_push(s, v);
  return stack_size(s);
}
```

Listing 8.36: Specification of Axiom (8.5)

Except for the `assigns` clauses, the ACSL-specification refers only to encapsulating logical functions and predicates defined in Section 8.4. If ACSL would provide a means to define encapsulating logical functions returning also sets of memory locations, the expressions in `assigns` clauses would not need to refer to the details of our `Stack` implementation.[58] As an alternative, `assigns` clauses could be omitted, as long as the proofs are only used to convince a human reader.

```
/*@
  requires Valid(s);
  requires !Full(s);

  assigns s->size;
  assigns s->obj[s->size];

  ensures \result == v;
*/
value_type axiom_top_of_push(Stack* s, value_type v)
{
  stack_push(s, v);
  return stack_top(s);
}
```

Listing 8.37: Specification of Axiom (8.6)

[58] In [9, § 2.3.4], a powerful sublanguage to build memory location set expressions is defined, lacking, however, just function definitions.

8.8.3. Removing an element from a stack

This section shows the Listings for Axioms 8.7, 8.8 and 8.9 which describe the behavior of a stack when an element is removed.

```
/*@
  requires  Valid(s) && !Empty(s);
  assigns   s->size;
  ensures   \result == Size{Old}(s) - 1;
*/
size_type axiom_size_of_pop(Stack* s)
{
  stack_pop(s);
  return stack_size(s);
}
```

Listing 8.38: Specification of Axiom (8.7)

```
/*@
  requires  Valid(s) && !Full(s);
  assigns   s->size, s->obj[s->size];
  ensures   Equal{Pre,Here}(s, s);
*/
void axiom_pop_of_push(Stack* s, value_type v)
{
  stack_push(s, v);
  stack_pop(s);
}
```

Listing 8.39: Specification of Axiom (8.8)

```
/*@
  requires  Valid(s) && !Empty(s);
  assigns   s->size, s->obj[s->size-1];
  ensures   Equal{Here,Old}(s, s);
*/
void axiom_push_of_pop_top(Stack* s)
{
  const value_type val = stack_top(s);
  stack_pop(s);
  stack_push(s, val);
}
```

Listing 8.40: Specification of Axiom (8.9)

9. Formal verification

In this chapter we introduce the formal verification tools used in this tutorial. We will afterwards present to what extent the examples from Chapters 3–8 could be deductively verified.

Within Frama-C, the WP plug-in [2] enables deductive verification of C programs that have been annotated with the ANSI/ISO-C Specification Language (ACSL)[1]. The WP plug-in uses weakest precondition computations to generate proof obligations. To formally prove the ACSL properties, these proof obligations can be submitted to external automatic theorem provers or interactive proof assistants.

For our experiments we used the WP plugin-in of Sodium release of Frama-C[59] together with the automatic theorem provers Alt-Ergo (version 0.99.1)[60] and CVC4 (version 1.4)[61] and the interactive theorem prover Coq (version 8.4.6)[62].

Here are the options of Frama-C that we used and that influence the number of generated proof obligations.

```
-wp
-wp-model Typed+ref
-wp-rte
-wp-split
```

For each algorithm we list in the following tables the number of generated verification conditions (VC), as well as the percentage of proven verification conditions. The tables show that all verification conditions could be verified. Moreover, with the exception of the more precise specification of `remove_copy` (Section 6.10) all algorithms are completely verified by the automatic theorem provers (Qed[63], Alt-Ergo and CVC4). We discharged the remaining few proof obligations of `remove_copy` with Coq (see Table 9.4).

Please note that the number of proven verification conditions do *not* reflect on the quality/strength of the individual provers. The reason for that is that we "pipe" each verification condition sequentially through Qed, Alt-Ergo, CVC4 and Coq. If one prover succeeds, then the remaining provers are not called.

[59] See `http://frama-c.com/install-sodium-20150201.html`

[60] See `http://alt-ergo.lri.fr`

[61] For the use of CVC4 (see `http://cvc4.cs.nyu.edu/web`) we relied on version 0.86 of the Why3 platform for deductive program verification (see `http://why3.lri.fr`).

[62] See `https://coq.inria.fr`

[63] Qed is the simplification engine of WP

Algorithm	Section	VCs			Individual Provers			
		All	Proven	(%)	Qed	Alt-Ergo	CVC4	Coq
`equal`	3.1	22	22	100	12	10	0	0
`equal` (`IsEqual`)	3.1	18	18	100	9	9	0	0
`equal` (`mismatch`)	3.2	7	7	100	6	1	0	0
`mismatch`	3.2	28	28	100	16	12	0	0
`find`	3.3	27	27	100	15	12	0	0
`find` (2)	3.4	27	27	100	17	10	0	0
`find_first_of`	3.5	34	34	100	23	11	0	0
`adjacent_find`	3.6	36	36	100	20	16	0	0
`search`	3.7	59	59	100	31	28	0	0
`count`	3.8	31	31	100	18	13	0	0

Table 9.1.: Results for non-mutating algorithms

Algorithm	Section	VCs			Individual Provers			
		All	Proven	(%)	Qed	Alt-Ergo	CVC4	Coq
properties of operator <	4.1	6	6	100	4	2	0	0
`max_element`	4.2	45	45	100	29	16	0	0
`max_element` (2)	4.3	45	45	100	29	16	0	0
`max_seq`	4.4	8	8	100	5	3	0	0
`min_element`	4.5	45	45	100	29	16	0	0

Table 9.2.: Results for maximum and minimum algorithms

Algorithm	Section	VCs			Individual Provers			
		All	Proven	(%)	Qed	Alt-Ergo	CVC4	Coq
`lower_bound`	5.1	36	36	100	20	16	0	0
`upper_bound`	5.2	36	36	100	18	18	0	0
`equal_range`	5.3	22	22	100	17	5	0	0
`binary_search`	5.4	12	12	100	8	4	0	0
`binary_search` (2)	5.4	15	15	100	8	7	0	0

Table 9.3.: Results for binary search algorithms

Algorithm	Section	VCs			Individual Provers			
		All	Proven	(%)	Qed	Alt-Ergo	CVC4	Coq
`fill`	6.2	17	17	100	7	9	1	0
`swap`	6.1	8	8	100	8	0	0	0
`swap_ranges`	6.3	38	38	100	9	25	4	0
`copy`	6.4	18	18	100	7	10	1	0
`reverse_copy`	6.5	21	21	100	7	13	1	0
`reverse`	6.6	38	38	100	11	24	3	0
`rotate_copy`	6.7	20	20	100	4	15	1	0
`replace_copy`	6.8	33	33	100	14	15	4	0
`remove_copy`	6.9	48	48	100	23	21	3	1
`remove_copy` (2)	6.10	62	62	100	23	29	3	7

Table 9.4.: Results for mutating algorithms

Algorithm	Section	VCs			Individual Provers			
		All	Proven	(%)	Qed	Alt-Ergo	CVC4	Coq
`iota`	7.1	22	22	100	12	9	1	0
`accumulate`	7.2	18	18	100	10	8	0	0
`inner_product`	7.3	23	23	100	10	13	0	0
`partial_sum`	7.4	32	32	100	13	15	3	1
`adjacent_difference`	7.5	32	32	100	14	16	1	1

Table 9.5.: Results for numeric algorithms

Algorithm	Section	VCs			Individual Provers			
		All	Proven	(%)	Qed	Alt-Ergo	CVC4	Coq
stack_equal	8.6	22	22	100	7	15	0	0
stack_init	8.7.1	14	14	100	3	11	0	0
stack_size	8.7.2	6	6	100	1	5	0	0
stack_empty	8.7.3	10	10	100	5	5	0	0
stack_full	8.7.4	11	11	100	5	6	0	0
stack_top	8.7.5	17	17	100	7	10	0	0
stack_push	8.7.6	56	56	100	41	12	3	0
stack_pop	8.7.7	43	43	100	31	12	0	0

Table 9.6.: Results for `Stack` functions

Algorithm	Section	VCs			Individual Provers			
		All	Proven	(%)	Qed	Alt-Ergo	CVC4	Coq
stack_size_wd	8.7.2	12	12	100	8	4	0	0
stack_empty_wd	8.7.3	12	12	100	8	4	0	0
stack_top_wd	8.7.5	12	12	100	8	4	0	0
stack_push_wd	8.7.6	8	8	100	1	4	3	0
stack_pop_wd	8.7.7	12	12	100	6	3	3	0

Table 9.7.: Results for the well-definition of the `Stack` functions

Algorithm	Section	VCs			Individual Provers			
		All	Proven	(%)	Qed	Alt-Ergo	CVC4	Coq
axiom_size_of_init	8.8.1	19	19	100	16	3	0	0
axiom_size_of_push	8.8.2	14	14	100	9	5	0	0
axiom_top_of_push	8.8.2	13	13	100	8	5	0	0
axiom_pop_of_push	8.8.3	12	12	100	7	5	0	0
axiom_size_of_pop	8.8.3	11	11	100	7	4	0	0
axiom_push_of_pop_top	8.8.3	17	17	100	11	6	0	0

Table 9.8.: Results for `Stack` axioms

A. History

This chapter describes the changes in previous versions of this document. For the most recent changes see Section .

The version numbers of this document are related to the versioning of Frama-C [3]. The versions of Frama-C are named consecutively after the elements of the periodic table. Therefore, our version numbering (X.Y.Z) are constructed as follows:

X the major number of our tutorial is the atomic number[64] of the chemical element after which Frama-C is named.

Y the Frama-C subrelease number

Z the subrelease number of this tutorial

A.1. New in Version 11.1.0 (March 2015)

- Use built-in predicates `\valid` and `\valid_read` instead of `IsValidRange`.
- Simplify loop invariants of `find_first_of`.
- Replace two loop invariants of `remove_copy` by ACSL lemmas.
- Rename several predicates
  - `IsEqual` ↦ `EqualRanges`.
  - `IsMaximum` ↦ `MaxElement`.
  - `IsMinimum` ↦ `MinElement`.
  - `Reverse` ↦ `Reversed`.
  - `IsSorted` ↦ `Sorted`.
- Several changes for `Stack`:
  - Rename `Stack` functions from *`foo`*`_stack` to `stack_`*`foo`*.
  - Equality of stacks now ignores the `capacity` field. This is similar to how equality for objects of type `std::vector<T>` is defined. As a consequence `stack_full` is not well-defined any more. Other stack functions are not effected.
  - Remove all assertions from stack functions (including in axioms).
  - Describe predicate `Separated` in text.

[64] See `http://en.wikipedia.org/wiki/Atomic_number`

A.2. New in Version 10.1.1 (January 2015)

- use option `-wp-split` to create simpler (but more) proof obligations
- simplify definition of predicate `Count`
- add new predicates for lower and upper bounds of ranges and use it in
  - `max_element`
  - `min_element`
  - `lower_bound`
  - `upper_bound`
  - `equal_range`
  - `fill`
- use a new auxiliary assertion in `equal_range` to enable the complete *automatic* verification of this algorithm
- add predicate `Unchanged` and use it to simplify the specification of several algorithms
  - `swap_ranges`
  - `reverse`
  - `remove_copy`
  - `stack_push` and `stack_push_wd`
  - `stack_pop` and `stack_pop_wd`
- add predicate `Reverse` and use it for more concise specifications of
  - `reverse_copy`
  - `reverse`
- several changes in the two versions of `remove_copy`
  - use predicate `HasValue` instead of logic function `Count`
  - add predicate `PreserveCount`
  - reformulate logic function `RemoveCount`
  - add predicate `StableRemove`
  - add predicate `RemoveCountMonotonic`
  - add predicate `RemoveCountJump`
- use overloading in ACSL to create shorter logic names for `Stack`
- remove unnecessary labels in several `Stack` functions

A.3. New in Version 10.1.0 (September 2014)

- remove additional labels in the `assumes` clauses of some stack function that were necessary due to an error in Oxygen
- provide a second version of `remove_copy` in order to explain the specification of the *stability* of the algorithms
- coarsen loop assigns of mutating algorithms
- temporarily remove the `unique_copy` algorithm

A.4. New in Version 9.3.1 (not published)

- specify bounds of the return value of `count` and fix reads clause of `Count` predicate
- use an auxiliary function `make_pair` in the implementation of `equal_range`
- provide more precise loop assigns clauses for the mutating algorithms
  - simplify implementation of `fill`
  - removed the `ensures \valid(p)` clause in specification of `swap`
  - simplify implementation of `swap_ranges`
  - simplify implementation of `copy`
  - fix implementation of `reverse_copy` after discovering an undefined behavior
  - new implementation of `reverse` that uses a simple **`for`**-loop
  - simplify implementation of `replace_copy`
  - refactor specification and simplify implementation of `remove_copy`
- remove work-around with `Pre`-label in `assumes` clauses of `stack_push` and `stack_pop`

A.5. New in Version 9.3.0 (December 2013)

- adjustments for *Fluorine* release of Frama-C
- `swap` now ensures that its pointer arguments are valid after the function has been called
- change definition of `size_type` to **`unsigned int`**
- change implementation of the `iota` algorithm . The content of the field `a` is calculated by increasing the value `val` instead of sum `val+i`.
- change implementation of `fill`.
- The specification/implementation of `Stack` has been revised by Kim Völlinger [14] and now has a much better verification rate.

A.6. New in Version 8.1.0 (not published)

- simplified specification and loop annotations of `replace_copy`
- add binary search variant `equal_range`
- greatly simplified specification of `remove_copy` by using the logic function `Count`
- remove chapter on heap operations

A.7. New in Version 7.1.1 (August 2012)

- improvements with respect to several suggestions and comments of Yannick Moy, e.g., specification refinements of `remove_copy`, `reverse_copy` and `iota`
- restricted verification of algorithms to WP with Alt-Ergo
- replaced deprecated `\valid_range` by `\valid` in definition of `IsValidRange`
- fixed inconsistencies in the description of the `Stack` data type
- binary search algorithms can now be proven without additional axioms for integer division
- changed axioms into lemmas to document that provability is expected, even if not currently granted
- adopted new Fraunhofer logo and contact email

A.8. New in Version 7.1.0 (December 2011)

- changed to Frama-C Nitrogen
- changed to Why 2.30
- discussed both plug-ins WP and Jessie
- removed `swap_values` algorithm

A.9. New in Version 6.1.0 (not published)

- changed definition of `Stack`
- renamed `reset_stack` to `init_stack`

A.10. New in Version 5.1.1 (February 2011)

- prepared algorithms for checking by the new WP plug-in of Frama-C

- changed to Alt-Ergo Version 0.92, Z3 Version 2.11 and Why 2.27
- added List of user-defined predicates and logic functions
- added remarks on the relation of logical values in C and ACSL
- rewrote section on `equal` and `mismatch`
- used a simpler logical function to count elements in an array
- added `search` algorithm
- added chapter to unite the maximum/minimum algorithms
- added chapter for the new `lower_bound`, `upper_bound` and `binary_search` algorithms
- added `swap_values` algorithm
- used `IsEqual` predicate for `swap_ranges` and `copy`
- added `reverse_copy` and `reverse` algorithms
- added `rotate_copy` algorithm
- added `unique_copy` algorithm
- added chapter on specification of the data type `Stack`

A.11. New in Version 5.1.0 (May 2010)

- adaption to Frama-C Boron and Why 2.26 releases
- changed from the `-jessie-no-regions` command-line option to using the pragma `SeparationPolicy(`*`value`*`)`

A.12. New in Version 4.2.2 (May 2010)

- changed to latest version of CVC3 2.2
- added additional remarks to our implementation of `find_first_of`
- changed `size_type` (**`int`**) to `integer` in all specifications
- removed casts in `fill` and `iota`
- renamed `is_valid_range` as `IsValidRange`
- renamed `has_value` as `HasValue`
- renamed predicate `all_equal` as `IsEqual`
- extended timeout to 30 sec.

A.13. New in Version 4.2.1 (April 2010)

- added alternative specification of `remove_copy` algorithm that uses `ghost` variables
- added Chapter on heap operations
- added `mismatch` algorithm
- moved algorithms `adjacent_find` and `min_element` from the appendix to chapter on non-mutating algorithms
- added typedefs `size_type` and `value_type` and used them in all algorithms
- renamed `is_valid_int_range` as `is_valid_range`

A.14. New in Version 4.2.0 (January 2010)

- complete rewrite of previous release
- adaption to Frama-C Beryllium 2 release

Bibliography

[1] ANSI/ISO C Specification Language. http://frama-c.com/acsl.html.

[2] WP Plug-in. http://frama-c.com/wp.html.

[3] Frama-C Software Analyzers. http://frama-c.com.

[4] CEA LIST, Laboratory of Applied Research on Software-Intensive Technologies. http://www-list.cea.fr/gb/index_gb.htm.

[5] INRIA-Saclay, French National Institute for Research in Computer Science and Control . http://www.inria.fr/saclay/.

[6] LRI, Laboratory for Computer Science at Université Paris-Sud. http://www.lri.fr/.

[7] Fraunhofer-Institut für Offene Kommunikationssysteme (FOKUS). http://www.fokus.fraunhofer.de.

[8] Virgile Prevosto. ACSL Mini-Tutorial. http://frama-c.com/download/acsl-tutorial.pdf.

[9] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ANSI/ISO C Specification Language, Version 1.8 Frama-C Nitrogen implementation. http://frama-c.com/download/acsl-implementation-Neon-20140301.pdf, March 2014.

[10] Programming languages – C, Committee Draft. http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1362.pdf, 2009.

[11] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[12] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–32, Providence, RI, 1967. American Mathematical Society.

[13] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J Comput*, 6(2):323–350, Jun 1977.

[14] Kim Völlinger. *Einsatz des Beweisassistenten Coq zur deduktiven Programmverifikation*. Diplomarbeit, Humboldt Universität zu Berlin, Germany, August 2013.

[15] Richard Fitzpatrick J.L. Heiberg. *Euclid's Elements of Geometry*. http://farside.ph.utexas.edu/euclid.html, Austin/TX, 2008.

[16] David Hilbert. *Grundlagen der Geometrie*. B.G.Teubner, Stuttgart, 1968.

[17] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.