ACSL By Example

Towards a Verified C Standard Library

Version 13.1.1
for
Frama-C (Aluminium)
November 2016

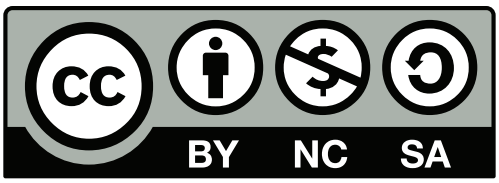
Jochen Burghardt
Robert Clausecker
Jens Gerlach
Hans Pohl

Former Authors

Andreas Carben, Liangliang Gu, Kerstin Hartig,
Timon Lapawczyk, Juan Soto, Kim Völlinger

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: 2012–2016
[3] project duration: 2009–2012

Changes

This release is intended for Frama-C *Aluminium*, issued in August 2016.[4] For changes in previous versions we refer to the Appendix A on Page 177.

List of changes (Version 13.1.1, November 2016)

- improve layout of tables of verification results
- use two additional automatic theorem provers (CVC3 and E)
- non-mutating algorithms
  - add algorithm `find_end`
  - add definition of predicate `HasSubRange` on subranges
  - add definition of predicate `EqualRanges` on subranges
  - rename lemma `HasSubRange_fit_size` to `HasSubRangeSize`
  - rename lemma `HasConstantSubRange_fit_size` to `HasSubRangeSize`
  - rename logic function `CountSection` to `Count` (using overloading in ACSL)
  - add lemma `HasValueCountInversion`
  - add lemma `HasValueShiftInversion`
  - add lemma `CountShift`
- mutating algorithms
  - add algorithm `copy_backward`
  - relax precondition on separation of `copy`, `replace_copy` and `remove_copy`
  - provide a more sophisticated implementation of `remove`
  - re-introduce a second version of `remove_copy` that also specifies the *stability* of the algorithm
  - add algorithm `random_shuffle`

[4] See `http://frama-c.com/download/frama-c-Aluminium-20160501.tar.gz`

Contents

List of logic definitions

List of Figures

List of Tables

1. Introduction

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).

We have chosen our examples from the C++ standard library whose initial version is still known as the *Standard Template Library* (STL).[5] The STL contains a broad collection of *generic* algorithms that work not only on C arrays but also on more elaborate container 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`.[6]

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, as ACSL is 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[7] and Pascal Cuoq from TrustInSoft[8].

We also like to express our gratitude to Claude Marché (LRI/INRIA)[9] and Yannick Moy (AdaCore)[10] for their helpful comments and detailed suggestions for improvement.

[5] See http://www.sgi.com/tech/stl/
[6] We are not directly using `int` in the source code but rather `value_type` which is a `typedef` for `int`.
[7] http://www-list.cea.fr/en
[8] http://trust-in-soft.com
[9] https://www.lri.fr/index_en.php?lang=EN
[10] http://www.adacore.com

1.1. Frama-C

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].

1.2. 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)
- heap algorithms (Chapter 8)

The order of these chapters reflects their increasing complexity.

Using the example of a stack, we tackle in Chapter 9 the problem of how a data type and its associated C functions can be specified with ACSL and automatically verified with Frama-C.

Finally, Chapter 10 lists for each example the results of verification with Frama-C.

1.3. Types, arrays, ranges and valid indices

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.

1.3.1. 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.

1.3.2. 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, §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.

ACSL syntax	Name	Reading
`!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

Here we show three example source-code fragments and annotations.

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

If `x` has an odd value before execution of the code `++x` then `x` has an even value thereafter.

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

If the value of `x` is in the range $\{0, \ldots, y\}$ before execution of the same code, then `x`’s value is in the range $\{0, \ldots, y + 1\}$ after execution.

`//@ assert true;`
`while (--x != 0)`
`  sum += a[x];`
`//@ assert x == 0;`
	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.[11] 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 first two examples 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 the second example, 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 the loop example.
- We can use the predicate **`true`** to denote the absence of a properly restricting precondition, as we did before the **`while`** loop.
- It is not possible to express by pre- and postconditions that a given piece of code will always terminate. The loop example 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 the loop example 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 2.2 to 2.12 in order to reason about programs. We will comment on them in the following sections.

[11] 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 2.2, where

$$\mathtt{P}\{\mathtt{x} \mapsto \mathtt{e}\}$$

denotes the result of substituting each occurrence of the variable `x` in the predicate `P` by the expression `e`.

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

Listing 2.2: The assignment rule

For example, if `P` is the predicate

```
x > 0 && a[2*x] == 0
```

then $\mathtt{P}\{\mathtt{x} \mapsto \mathtt{y} + 1\}$ is the predicate

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

Hence, we get Listing 2.3 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 2.3: An assignment rule example instance

Note that after a substitution several different predicates `P` may result in the same predicate $\mathtt{P}\{\mathtt{x} \mapsto \mathtt{e}\}$. For example, after applying the substitution $\mathtt{P}\{\mathtt{x} \mapsto \mathtt{y} + 1\}$ each of the following four predicates

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

turns into

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

For this reason, the same precondition and statement may result in several different postconditions (All four above expressions are valid postconditions in Listing 2.3, 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 2.4.

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

Listing 2.4: Simpler, but *faulty* assignment rule

Listings 2.5–2.7 show some example instances of this faulty rule.

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

Listing 2.5: An example instance of the faulty rule from Listing 2.4

While Listing 2.5 happens to be ok, Listing 2.6 and 2.7 lead to postconditions that are obviously nonsensical formulas.

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

Listing 2.6: An example instance of the faulty rule from Listing 2.4

The reason is that in the assignment in Listing 2.6 the left-hand side variable `x` also appears in the right-hand side expression `e`, while the assignment in Listing 2.7 just destroys the property from its precondition.

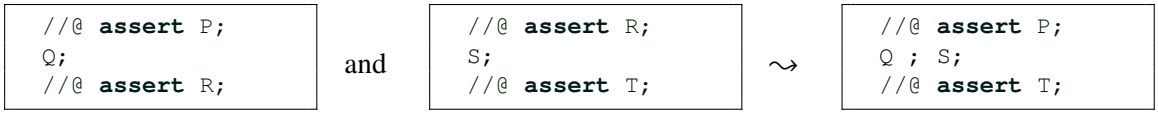
```
//@ assert x < 0;
x = 5;
//@ assert x < 0 && x == 5;
```

Listing 2.7: An example instance of the faulty rule from Listing 2.4

Note that the correct example Listing 2.5 can as well be obtained as an instance of the correct rule from Listing 2.2, 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 2.8, 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.



Listing 2.8: The sequence rule

2.3. The implication rule

The implication rule, shown in Listing 2.9, allows us at any time to sharpen a precondition `P` and to weaken a postcondition `R`. More precisely, if we know that `P' ==> P` and `R ==> R'` then the we can replace the left contract in of Listing 2.9 by the right one.

Listing 2.9: The implication rule

2.4. The choice rule

The choice rule, depicted in Listing 2.10, is needed to verify conditional statements of the form

```
if (C) X;
else   Y;
```

Both the then and else branch 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 `C`, 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 && !C`.

```
//@ assert P && C;
X;
//@ assert S;
```

and

```
//@ assert P && !C;
Y;
//@ assert S;
```

$\rightsquigarrow$

```
//@ assert P;
if (C) X;
else   Y;
//@ assert S;
```

Listing 2.10: The choice rule

Listing 2.11 shows an example application of the choice rule.

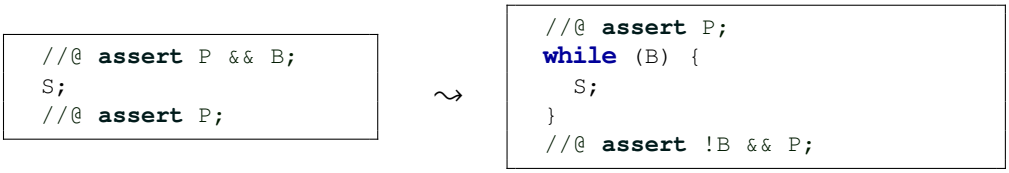
```
//@ assert 0 <= i < n;           // given precondition
if (i < n-1) {
  //@ assert 0 <= i < n - 1;     // using that i < n-1 holds in this branch
  //@ assert 1 <= i+1 < n;       // by the implication rule
  i = i+1;
  //@ assert 1 <= i < n;         // by the assignment rule
  //@ assert 0 <= i < n;         // weakened by the implication rule
} else {
  //@ assert 0 <= i == n-1 < n; // using that !(i<n-1) holds in else part
  //@ assert 0 == 0 && 0 < n;   // 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 both branches
```

Listing 2.11: 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 of the implication rule to establish preconditions for the assignment rule as needed, and to unify the postconditions of the then and else branches, as required by the choice rule.

2.5. The loop rule

The loop rule, shown in Listing 2.12, 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.



Listing 2.12: 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. Listing 2.13 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 2.11 as a sub-part.

```
//@ assert  0 < n;                  // given precondition

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

  i = (i < n-1) ? i+1 : 0;
  //@ assert 0 <= i < n;            // follows by the choice rule

  process(a, i, &done);
  //@ assert 0 <= i < n;            // required property of process
}
//@ assert 0 <= i < n;              // by the loop rule
```

Listing 2.13: An abstract ring buffer loop

To reuse the proof from Listing 2.11, we had to drop the conjunct `!done`, since we didn't consider it in Listing 2.11. 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 2.11, 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 2.11. 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 6.4 on Page 79.

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`[12] 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 2.11 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.

[12] 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 do not 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, if the expression `E` doesn't have side-effects, then

```
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;
}
```

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, a loop statement of the form

```
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 change neither 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. Each algorithm works on one-dimensional arrays.

- `find` (Section 3.1 on Page 26) provides *sequential* or *linear search* and returns the smallest index at which a given value occurs in a given range. In Section 3.2, on Page 28, a predicate is introduced in order to simplify the specification.
- `find_first_of` (Section 3.3, on Page 30) provides similar to `find` a *sequential search*, but unlike `find` it does not search for a particular value, but for an arbitrary member of a set.
- `adjacent_find` (Section 3.4 on Page 32) can be used to find equal neighbors in an array.
- `equal` and `mismatch` (Section 3.5 on Page 34) are useful for comparing two ranges element-by-element and identifying where they differ.
- `search` and `search_n` (Sections 3.6 and 3.7) find a subsequence that is identical to a given sequence when compared element-by-element and returns the position of the first occurrence.
- `count` (Section 3.9, on Page 46) 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 `find` algorithm

The `find` algorithm in the C++ standard library implements *sequential search* for general sequences.[13] 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.

As an example, we consider in Figure 3.1 an array. The arrows indicate which indices will be returned by `find` for a given value. Note that the index 9 points *one past end* of the array. Values that are not contained in the array are colored in gray.

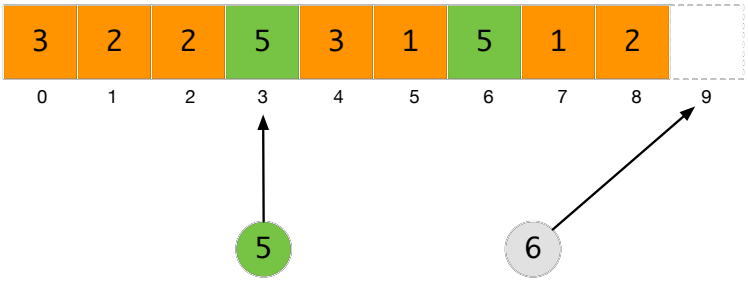


Figure 3.1.: A simple example for `find`

3.1.1. Formal specification of `find`

The formal specification of `find` in ACSL is shown in Listing 3.2.

```
/*@
  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 3.2: Formal specification of `find`

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

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 1.3). 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 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 3.2). 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*.

3.1.2. Implementation of `find`

Listing 3.3 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 3.3: Implementation of `find`

The first loop *invariant* is needed to prove that accesses to `a` only occur with valid indices. The second loop *invariant* is needed for the proof of the postconditions of the behavior `some` (see Listing 3.2). 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.

3.2. The `find` algorithm reconsidered

In this section we specify the `find` algorithm in a slightly different way when compared to Section 3.1. Our approach is motivated by a considerable number of closely related formulas. We have in Listings 3.2 and 3.3 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.

3.2.1. The predicate `HasValue`

In order to be more explicit about the commonalities of these formulas we define a predicate, called `HasValue` (see Listing 3.4), which describes the situation that there is a valid index `i` where `a[i]` equals `val`.

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

  predicate
    HasValue{A}(value_type* a, integer n, value_type v) =
      HasValue(a, 0, n, v);
*/
```

Listing 3.4: The predicate `HasValue`

Note that we needed to provide a label, viz. `A`, to the predicate, since the evaluation the predicate depends on a memory state, viz. the contents of `a[0..n-1]`. In general, we have to write

```
\exists integer i; 0 <= i < n && \at(a[i],A) == v;
```

in order to express that we refer to the value `a[i]` in the program state `A`. However, ACSL allows to abbreviate `\at(a[i],A)` by `a[i]` if, as in `HasValue`, the label `A` is the only available label.

With this predicate we can encapsulate all uses of the universal and existential quantifiers in both the function contract of `find` and in its loop annotations. The result is shown in Listings 3.5 and 3.6.

3.2.2. Formal specification of `find`

Th revised contract for `find` in Listing 3.5 is more concise than the previous one in Listing 3.2. 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` shown in Listing 3.5.

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

  assigns  \nothing;

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

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

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

Listing 3.5: Formal specification of `find` using the `HasValue` predicate

We also enriched the specification of `find` by user-defined names (sometimes called *labels*, too, the distinction to program state identifiers being obvious) to refer to the `requires` and `ensures` clauses. We highly recommend this practice in particular for more complex annotations. For example, Frama-C can be instructed to verify only clauses with a given name.

3.2.3. Implementation of `find`

The predicate `HasValue` is also used in the loop annotation inside the implementation of `find`. Note that, as in the case of the specification, we use labels to name individual annotations.

```
size_type find(const value_type* a, size_type n, value_type val)
{
  /*@
    loop invariant bound:     0 <= i <= n;
    loop invariant not_found: !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 3.6: Implementation of `find` with loop annotations based on `HasValue`

3.3. The `find_first_of` algorithm

The `find_first_of` algorithm[14] is closely related to `find` (see Sections 3.1 and 3.2).

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

Like `find`, it performs a sequential search. However, while `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]`.

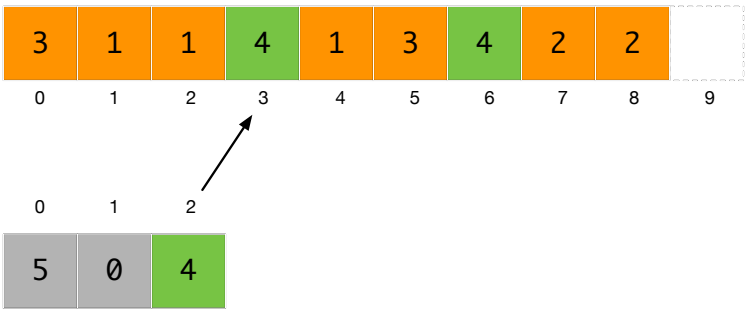


Figure 3.7.: A simple example for `find_first_of`

As an example, we consider in Figure 3.7 two arrays. The arrow indicates the smallest index where one of the elements of the three-element array occurs.

3.3.1. The predicate `HasValueOf`

Similar to our approach in Section 3.2, we define a predicate `HasValueOf` that formalizes the fact that there are valid indices `i` and `j` of the respective arrays `a` and `b` such that `a[i] == b[j]` holds. We have chosen to reuse the predicate `HasValue` (Listing 3.4) to define `HasValueOf` (Listing 3.8).

```
/*@
  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 3.8: The predicate `HasValueOf`

3.3.2. Formal specification of `find_first_of`

The formal specification of `find_first_of` is shown Listing 3.9. The function contract uses the predicates `HasValueOf` and `HasValue` thereby making it very similar the specification `find` (Listing 3.5).

[14] See `http://www.sgi.com/tech/stl/find_first_of.html`

```
/*@
  requires valid: \valid_read(a + (0..m-1));
  requires valid: \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 3.9: Formal specification of `find_first_of`

3.3.3. Implementation of `find_first_of`

Our implementation of `find_first_of` is shown in Listing 3.10.

```
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 3.10: Implementation of `find_first_of`

Note the call of the `find` function. In the original implementation[15], `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 fewer loop annotations.

[15] See `http://www.sgi.com/tech/stl/stl_algo.h`

3.4. The `adjacent_find` algorithm

The `adjacent_find` algorithm[16]

```
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.

The arrow in Figure 3.11 indicates the smallest index where two adjacent elements are equal.

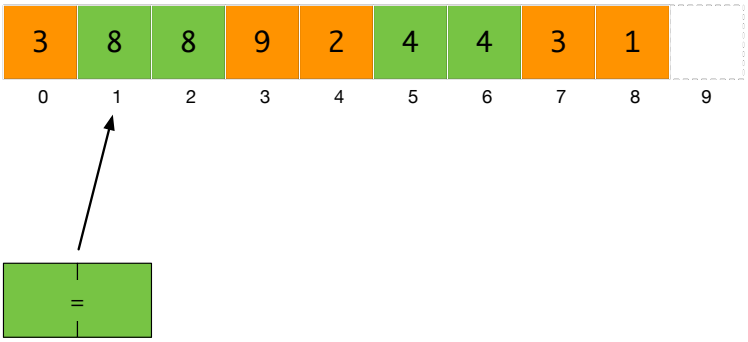


Figure 3.11.: A simple example for `adjacent_find`

3.4.1. The predicate `HasEqualNeighbors`

As in the case of other search algorithms, we first define a predicate `HasEqualNeighbors` (see Listing 3.12) that captures the essence of finding two adjacent indices at which the array holds equal values.

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

Listing 3.12: The predicate `HasEqualNeighbors`

3.4.2. Formal specification of `adjacent_find`

We use the predicate `HasEqualNeighbors` to define the formal specification of `adjacent_find` (see Listing 3.13).

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

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

  assigns \nothing;

  ensures bound:  0 <= \result <= n;

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

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

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

Listing 3.13: Formal specification of `adjacent_find`

3.4.3. Implementation of `adjacent_find`

The implementation of `adjacent_find`, including loop annotations is shown in Listing 3.14. At the beginning we check whether the array contains at least two elements. Otherwise, there is no point in looking for adjacent neighbors ...

```
size_type
adjacent_find(const value_type* a, size_type n)
{
  if (n <= 1) {
    return n;
  }

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

  return n;
}
```

Listing 3.14: Implementation of `adjacent_find`

Note the use of the predicate `HasEqualNeighbors` in the loop invariant to match the similar postcondition of behavior `some`.

3.5. The `equal` and `mismatch` algorithms

The `equal`[17] and `mismatch`[18] algorithms in the C++ Standard Library compare two generic sequences. For our purposes we have modified the generic implementation to that of an array of type `value_type`. The signatures read

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

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

The `mismatch` algorithm is slightly more general than the negation of `equal`: it 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 (common) length `n` of the two ranges.

3.5.1. 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 property.

The motto *don't repeat yourself* is not just good programming practice.[19] 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 property.

In Listing 3.15 we introduce two versions of the predicate `EqualRanges`. The first and more general definition formalizes the fact that the arrays `a[m..n-1]` and `b[m..n-1]` are equal when compared element by element. The second definition of `EqualRanges` is a shortcut that allows to express the equality of two complete ranges.

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

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

Listing 3.15: Two versions of predicate `EqualRanges`

The letters `A` and `B` are *labels*[20] that represent *program points*. 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, §2.6.9].

[17] See `http://www.sgi.com/tech/stl/equal.html`
[18] See `http://www.sgi.com/tech/stl/mismatch.html`
[19] Compare `http://en.wikipedia.org/wiki/Don't_repeat_yourself`
[20] Labels are used in C to name the target of the *goto* jump statement.

To evaluate the expression `\equalranges{A,B}(a,n,b)`, arbitrarily choose a memory state observed at program point `A` and `B`, and consider the contents of the array `a` and `b` in that state, respectively. If, and only if, both contents agree element by element, independent of the state choice, the expression yields **`true`**. In the latter case, in particular, if `B` is reachable at all, `a` must have the same contents every time `A` is passed through, and vice versa.

3.5.2. Formal specification of `equal` and `mismatch`

Using predicate `equal_range` we can formulate the specification of `equal` in Listing 3.16, using the predefined label `Here`. When used in an `ensures` clause, the label `Here` refers to the post-state of a function. Note that the equivalence is needed in the ensures clause. Putting an equality instead is not legal in ACSL, because `EqualRanges` is a predicate, not a function.

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

  assigns \nothing;

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

Listing 3.16: Formal specification of `equal`

The formal specification of `mismatch` in Listing 3.17 is more complex than that of `equal` because the return value of mismatch provides more information than just reporting whether the two arrays are equal. On the other, the specification is conceptually quite similar to that of `find` (Listing 3.5). While `find` returns the smallest index `i` where `a[i] == val` holds, `mismatch` finds the smallest index `a[i] != b[i]`.

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

  assigns \nothing;

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

  behavior some_not_equal:
    assumes !EqualRanges{Here,Here}(a, n, b);
    ensures bound:   0 <= \result < n;
    ensures result:  a[\result] != b[\result];
    ensures first:   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 3.17: Formal specification of `mismatch`

Note in particular the use of `EqualRanges` in the specification of `mismatch`. As in the specification of `find` the completeness and disjointness of `mismatch`'s behaviors is quite obvious, because the `assumes` clauses of `all_equal` and `some_not_equal` are negations of each other.

3.5.3. Implementation of `equal` and `mismatch`

Listing 3.18 shows an implementation of the `equal` algorithm by a simple call of `mismatch`.[21]

```
bool equal(const value_type* a, size_type n, const value_type* b)
{
  return mismatch(a, n, b) == n;
}
```

Listing 3.18: Implementation of `equal` with `mismatch`

Listing 3.19 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 bound:  0 <= i <= n;
    loop invariant equal:  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 3.19: Implementation of `mismatch`

We use the predicate `EqualRanges` as shown in Listing 3.19 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.

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

3.6. 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`.[22] 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 `s` of the array `a` where the condition `a[s+k]==b[k]` holds for each index `k` with `0 <= k < n` (see Figure 3.20). If no such index exists, then `search` returns the length `m` of the array `a`.

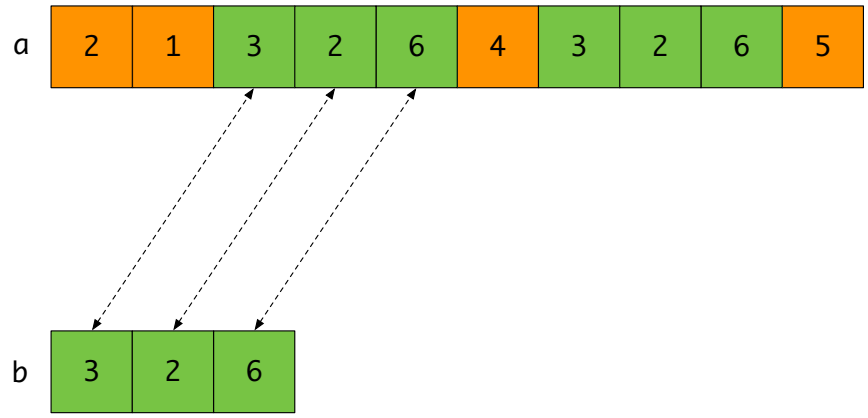


Figure 3.20.: Searching the first occurrence of `b[0..n-1]` in `a[0..m-1]`

3.6.1. The predicate `HasSubRange`

Our specification of `search` starts with introducing the predicate `HasSubRange` in Listing 3.21. This predicate formalizes, using the predicate `EqualRanges` defined in Listing 3.15, that the sequence `a` contains a subsequence which equal the sequence `b`. Of course, in order to contain a subsequence of length `n`, `a` must be at least that large; this is expressed by lemma `HasSubRangeSizes`.

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

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

  lemma HasSubRangeSizes:
    \forall value_type *a, *b, integer f, t, n;
      HasSubRange(a, f, t, b, n) ==> n <= t - f;
*/
```

Listing 3.21: The predicate `HasSubRange`

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

3.6.2. Formal specification of `search`

The ACSL specification of `search` is shown in Listing 3.22. Conceptually, the specification of `search` is very similar to that of `find` (Section 3.1). We therefore use again two behaviors to capture the essential aspects of `search`. The behavior `has_match` applies if the sequence `a` contains a subsequence identical to `b`. We express this condition with `assumes` using the predicate `HasSubRange`.

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

  assigns \nothing;

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

  behavior no_match:
    assumes !HasSubRange(a, 0, m, b, n);
    ensures result: \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 first ensures clause `bound` indicates that the return value must be in the range `0..m-n`. The clause `result` of behavior `has_match` expresses that `search` returns the smallest index where `b` can be found in `a`. Clause `first` indicates that the sequence `a` contains a subsequence (starting from the position `\result`) identical to `b`.

The behavior `no_match` covers the case that there is no subsequence `a` that equals `b`. In this case, `search` must return the length `m` of the range `a`.

If the ranges `a` or `b` are empty then the return value will be `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*.

3.6.3. 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.[23]

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

  /*@
    loop invariant bound:     i <= m-n+1;
    loop invariant not_found: !HasSubRange(a, 0, 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)) {
      return i;
    }
  }

  return m;
}
```

Listing 3.23: Implementation of `search`

The loop invariant `not_found`is needed for the proof of the postconditions of the behavior `has_match` (see Listing 3.22). It expresses that the subsequence `b` has not been found up to the current iteration step.

The trivial case `n>m` is caught separately in order to prevent an overflow in computation of `m-n` in the loop. Neither `n==0` nor `m==0` need to be handled separately, not even for efficiency reasons: in the former case, `equal(a+i,n,b)` will succeed in the first iteration, while in the latter, `n>m` will apply.

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

3.7. The `search_n` algorithm

The `search_n` algorithm in the C++ standard library finds the first place where a given value start to occur a given number of times in a given sequence. For our purposes we have modified the generic implementation to that of an array of type `value_type`.[24] The signature now reads:

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

Note the similarity to the signature of `search` (Sect. 3.6). The only difference is that `b` now is a single value rather than an array.[25]

The function `search_n` returns the first index `s` of the array `a` where the condition `a[s+k] == b` holds for each index `k` with `0 <= k < n` (see Figure 3.24). If no such index exists, then `search_n` returns the length `m` of the array `a`.

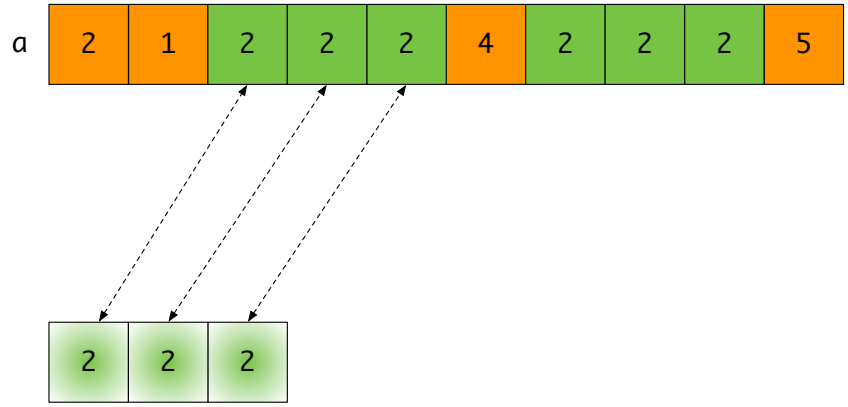


Figure 3.24.: Searching the first occurrence a given constant sequence in `a[0..m-1]`

3.7.1. The predicates `ConstantRange` and `HasConstantSubRange`

Our specification of `search_n` starts with introducing the predicate `ConstantRange` in Listing 3.25, which expresses that each member `a[first..last-1]` has the value `val`.

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

Listing 3.25: The predicate `ConstantRange`

Based on that, the predicate `HasConstantSubRange` in Listing 3.26 formalizes that the sequence `a` of length `m` contains a subsequence of `n` times the value `b`. Similar to `HasSubRange`, in order to contain `n` repetitions, `a` must be at least that large; this is what lemma `HasConstantSubRange_fit_size` says.

[24] See `http://www.sgi.com/tech/stl/search_n.html`
[25] For some reason, the C++ standard library has swapped the `b` and `n` parameter; we followed that order.

```
/*@
  predicate
    HasConstantSubRange{A}(value_type* a, integer m, integer n, value_type b) =
      \exists integer i; 0 <= i <= m-n && ConstantRange(a, i, i+n, b);


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

Listing 3.26: The predicate `HasConstantSubRange`

3.7.2. Formal specification of `search_n`

The ACSL specification of `search_n` is shown in Listing 3.27. Like for `search`, the specification of `search_n` is very similar to that of `find`. We again use two behaviors to capture the essential aspects of `search_n`. The behavior `has_match` applies if the sequence `a` contains an `n`-fold repetition of `b`. We express this condition with `assumes` by using the predicate `HasConstantSubRange`.

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

  assigns \nothing;

  behavior has_match:
    assumes HasConstantSubRange(a, m, n, b);
    ensures result:  0 <= \result <= m-n;
    ensures match:   ConstantRange(a, \result, \result+n, b);
    ensures first:   !HasConstantSubRange(a, \result+n-1, n, b);

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

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

Listing 3.27: Formal specification of `search_n`

The `result` ensures clause of behavior `has_match` indicates that the return value must be in the range `[0..m-n]`. The `match` ensures clause expresses that the return value of `search_n` actually points to an index where `b` can be found `n` or more times in `a`. The `first` ensures clause expresses that the minimal index with this property is returned.

The behavior `no_match` covers the case that there is no matching subsequence in sequence `a`. In this case, `search_n` must return the length `m` of the range `a`.

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*.

3.7.3. Implementation of `search_n`

Our implementation of `search_n` is shown in Listing 3.28. It deviates from the C++ standard library implementation in using only one loop and thus being easier to understand.

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

  if (m < n) {
    return m;
  }

  size_type start = 0;

  /*@
    loop invariant constant:  ConstantRange(a, start, i, b);
    loop invariant start:     start == 0 || !ConstantRange(a, start-1, i, b);
    loop invariant not_found: !HasConstantSubRange(a, i, n, b);
    loop assigns i, start;
    loop variant m - i;
  */
  for (size_type i = 0; i < m; ++i) {
    if (a[i] != b) {
      start = i + 1;
    } else if (i + 1 - start >= n) {
      return start;
    }
  }

  return m;
}
```

Listing 3.28: Implementation of `search_n`

It maintains in the variable `start` the start of the most recent consecutive range of `b`s; that is, `a[start..i-1]` is always a sequence of `b`s only. This is expressed by the loop invariant `constant`. Loop invariant `start` states that the variable `start` actually holds the minimal starting index of such a range, i.e. that `a[start-1..i-1]` does not consist of `b`s only (unless `start` is zero). The loop invariant `not_found` states that we didn't find an `n`-fold repetition of `b` up to now; if we find one, we terminate the loop, returning `start`. Note that no loop invariant `0<=i<=m` is necessary.

3.8. The `find_end` algorithm

The `find_end` algorithm in the C++ standard library searches for the last 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`.[26] The signature now reads:

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

The function `find_end` returns the greatest index `s` of the array `a` where the condition `a[s+k]==b[k]` holds for each index `k` with `0 <= k < n` (see Figure 3.29). If no such index exists, then `find_end` returns the length `m` of the array `a`. One has to remark the special case `n == 0`. In this case the last position of the empty string is found (the length `m`) and returned.

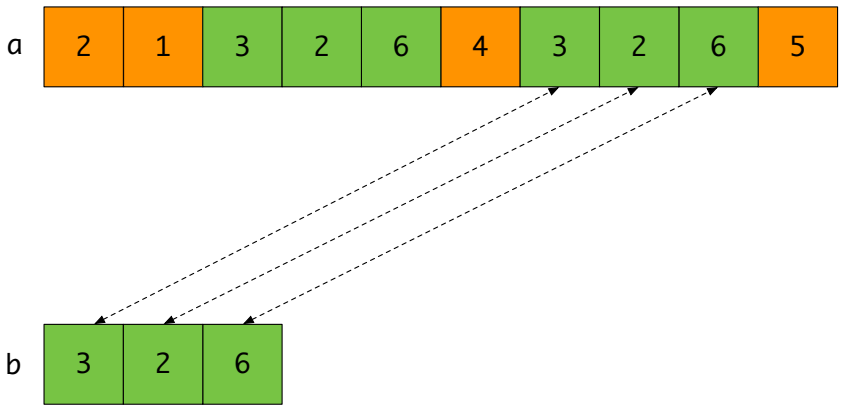


Figure 3.29.: Finding the last occurrence `b[0..n-1]` in `a[0..m-1]`

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

3.8.1. Formal specification of `find_end`

The ACSL specification of `find_end` is shown in Listing 3.30. Conceptually, the specification of `find_end` is very similar to that of `find` in Section 3.2. We therefore use again behaviors to capture the essential aspects of `find_end`.

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

  assigns \nothing;

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

  behavior no_match:
    assumes !HasSubRange(a, 0, m, b, n);
    ensures result: \result == m;

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

Listing 3.30: Formal specification of `find_end`

The behavior `has_match` applies if the sequence `a` contains a subsequence identical to `b`. We express this condition with `assumes` using the predicate `HasSubRange`. The `ensures` clause `bound` indicates that the return value must be in the range `0..m-n`. The clause `result` of behavior `has_match` expresses that `find_end` returns an index where `b` can be found in `a`. Finally, the clause `last` indicates that the sequence `a` does not contain `b` beginning at a position larger than `\result`.

The behavior `no_match` covers the case that there is no subsequence of `a` that equals `b`. In this case, `find_end` must return the length `m` of the range `a`.

It is quite clear that these behaviors are *complete* and *disjoint*.

3.8.2. Implementation of `find_end`

Our implementation of `find_end` is shown in Listing 3.31. Similar to our `search` implementation (Section 3.6), it follows the C++ standard library implementation in being easy to understand, but needing an order of magnitude of `m*n` rather than only `m+n` operations.

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

  if (n > m) {
    return m;
  }

  size_type ret = m;

  /*@
    loop invariant bound:   ret <= m - n || ret == m;
    loop invariant result1: ret == m ==> !HasSubRange(a, n+i-1, b, n);
    loop invariant result2: ret < m  ==> EqualRanges{Here,Here}(a+ret, n, b);
    loop invariant last:    ret < m  ==> !HasSubRange(a, ret+1, i+n-1, b, n);
    loop assigns i, ret;
    loop variant m - i;
  */
  for (size_type i = 0; i <= m - n; ++i) {
    if (equal(a + i, n, b)) {
      ret = i;
    }
  }

  return ret;
}
```

Listing 3.31: Implementation of `find_end`

We maintain in the variable `ret` the prospective value to be returned, according to the current knowledge. Initially, it is set to `m`, meaning "no occurrence of `b` found yet". Whenever an occurrence is found, `ret` is updated to its starting position.

Invariant `bound` states that `ret` either still has the value `m` or has a value up to `m-n`. For the former case, invariant `result1` indicates that no occurrence of `b` has been found. For the latter case, invariant `result2` indicates that an occurrence at `ret` has been found, and invariant `last` states that none was found so far after `ret`.

3.9. 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[27] 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`.

3.9.1. An axiomatic definition of counting on array sections

When trying to specify `count` we are faced with the situation that ACSL does not provide a definition of counting a value in an array.[28] We therefore start with an axiomatic definition of of *logic function* `Count` that captures the basic intuitive features of of counting on an array section. The expression `CountSection(a,m,n,v)` returns the number of occurrences of `v` in `a[m],...,a[n-1]`.

The specification of `count` will then be fairly short because it employs our *logic function* `Count` whose (considerably) longer definition is given in Listing 3.32.[29]

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

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

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

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

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

Listing 3.32: The logic function `Count`

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

[28] This statement is not quite true because the ACSL documentation lists `numof` as one of several *higher order logic constructions* [9, §2.6.7]. However, these *extended quantifiers* are mentioned only as experimental features.

[29] This definition of `Count` is a generalization of the *logic function* `nb_occ` of the ACSL specification [9, p. 55].

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

  Axiom `CountSectionRead` makes this claim explicit by ensuring 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`. We use predicate `Unchanged` (Listing 6.1 in Section 6.1) to express the premise of Axiom `CountSectionRead`. Axiom `CountSectionRead` is helpful if one has to verify *mutating* algorithms that rely on `Count`, e.g., `remove_copy` in Section 6.13. It is an inductive consequence of axioms `CountSectionEmpty`, `CountSectionOneHit`, and `CountSectionOneMiss`, but we don't prove it here.

The following properties of `Count` can be verified with the help of the axioms given in Listing 3.32.

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


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

Listing 3.33: Some lemmas for `Count`

3.9.2. Counting on a whole array

We also provide in Listing 3.34 an overloaded version of the logic function `Count` that only takes the starting address and the size of an array. This is just a convenience function for the use in specifications that do not need to consider array sections. Note how the accompanying lemmas in Listing 3.34 correspond to the axioms in Listing 3.32.

```
/*@
  logic integer
    Count{L}(value_type* a, integer n, value_type v) = Count{L}(a, 0, n, v);


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

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

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

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

Listing 3.34: The logic function `Count`

The lemmas for `Count` in Listing 3.35 are just reformulated versions of those in Listing 3.33.

```
/*@
  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 m, n;
       0 <= m <= n  ==>
         Count(a, n, v) == Count(a, 0, m, v) + Count(a, m, n, v);

*/
```

Listing 3.35: Some lemmas for `Count`

3.9.3. Formal specification of `count`

Listing 3.36 shows how we use the logic function `Count` from Listing 3.34 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 size of the array.

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

  assigns \nothing;

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

Listing 3.36: Formal specification of `count`

3.9.4. Implementation of `count`

Listing 3.37 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 bound: 0 <= i <= n;
    loop invariant bound: 0 <= counted <= i;
    loop invariant count: 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 3.37: 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 54) 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 55).
- `max_seq` (Section 4.4, on Page 57) 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.[30] To be precises, the operator < must be a *partial order*,[31] 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`[32] 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)`.

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

[30] See `http://www.sgi.com/tech/stl/LessThanComparable.html`.
[31] See `http://en.wikipedia.org/wiki/Partially_ordered_set`
[32] See Section 1.3

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


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


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


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

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[33] 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:  \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 upper:   \forall integer i; 0 <= i < n       ==> a[i] <= a[\result];
    ensures strict:  \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 (`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.

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.

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

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 bound:  0 <= i <= n;
    loop invariant max:    0 <= max <  n;
    loop invariant upper:  \forall integer k; 0 <= k < i   ==> a[k] <= a[max];
    loop invariant strict: \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 max:      MaxElement(a, n, \result);
    ensures strict:   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.

```
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 max:     0 <= max < n;
    loop invariant upper:   UpperBound(a, 0, i, a[max]);
    loop invariant strict:  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[34] 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 min:     MinElement(a, n, \result);
    ensures strict:  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`

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 $[0..n-1]$.

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

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
- two variants for the implementation of `equal_range` in Section 5.3
- two variants for the formal specification of `binary_search` in Section 5.4

All binary search algorithms require that their input array is sorted in ascending order. There are two versions of predicate `Sorted` in Listing 5.1. The first one defines when a section of an array is sorted in ascending order. The second version uses the first one to express that the whole array is sorted.

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

  predicate
    Sorted{L}(value_type* a, integer n) = Sorted{L}(a, 0, n);
*/
```

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 3.1, 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[35] 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. The index `lb`, that `lower_bound` returns satisfies the inequality

$$0 \leq \texttt{lb} \leq n \tag{5.1}$$

and has the following properties for a valid index `k` of the array under consideration

$$0 \leq k < \texttt{lb} \implies a[k] < \texttt{val} \tag{5.2}$$
$$\texttt{lb} \leq k < n \implies \texttt{val} \leq a[k] \tag{5.3}$$

Conditions (5.2) and (5.3) imply that `val` can only occur in the array section `a[lb..n-1]`. In this sense `lower_bound` returns a *lower bound* for the potential indices.

As an example, we consider in Figure 5.2 a sorted array. The arrows indicate which indices will be returned by `lower_bound` for a given value. Note that the index 9 points *one past end* of the array. Values that are not contained in the array are colored in gray.

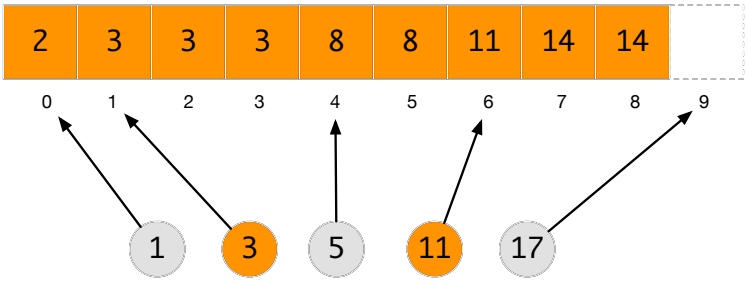


Figure 5.2.: Some examples for `lower_bound`

Figure 5.2 also clarifies that care must be taken when interpreting the return value of `lower_bound`. An important difference to the algorithms in Chapter 3 is that a return value of `lower_bound` that is less than n does not necessarily implies `a[lb] == val`. We can only be sure that `val <= a[lb]` holds.

5.1.1. Formal specification of `lower_bound`

The ACSL specification of `lower_bound` is shown in Listing 5.3. The preconditions `sorted` expresses that the values in the (valid) array need to be sorted in ascending order. The postconditions reflect the conditions listed above and can be expressed using predicates from Listing 4.3, namely,

- Condition (5.1) becomes postcondition `result`

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

- Condition (5.2) becomes postcondition `left`
- Condition (5.3) postcondition property `right`

```
/*@
  requires valid:  \valid_read(a + (0..n-1));
  requires sorted: 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 5.3: Formal specification of `lower_bound`

5.1.2. Implementation of `lower_bound`

Our implementation of `lower_bound` is shown in Listing 5.4. 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 that are less than `val` and all indices not less than the temporary right bound `right` contain values that are greater or equal than `val`. The expression to compute of `middle` is slightly more complex than the naïve `(left+right)/2`, but it avoids potential overflows.

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

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

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

  return left;
}
```

Listing 5.4: Implementation of `lower_bound`

5.2. The `upper_bound` algorithm

The `upper_bound`[36] algorithm is a variant of binary search and 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)
```

As with the other binary search algorithms, `upper_bound` requires that its input array is sorted in ascending order. The index `ub` returned by `upper_bound` satisfies the inequality

$$0 \leq \texttt{ub} \leq n \tag{5.4}$$

and is involved in the following implications for a valid index `k` of the array under consideration

$$0 \leq k < \texttt{ub} \implies a[k] \leq \texttt{val} \tag{5.5}$$
$$\texttt{ub} \leq k < n \implies \texttt{val} < a[k] \tag{5.6}$$

Conditions (5.5) and (5.6) imply that `val` can only occur in the array section `a[0..ub-1]`. In this sense `upper_bound` returns a *upper bound* for the potential indices where `val` can occur. It also means that the searched-for value `val` can *never* be located at the index `ub`.

Figure 5.5 is a variant of Figure 5.2 for the case of `upper_bound` and the same example array. The arrows indicate which indices will be returned by `upper_bound` for a given value. Note how, compared to Figure 5.2, only the arrows from values that *are present* in the array change there target index.

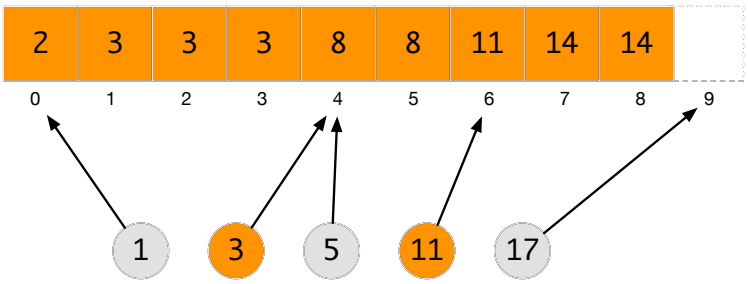


Figure 5.5.: Some examples for `upper_bound`

5.2.1. Formal specification of `upper_bound`

The ACSL specification of `upper_bound` is shown in Listing 5.6.

The specification is quite similar to the specification of `lower_bound` (see Listing 5.3). The preconditions `sorted` expresses that the values in the (valid) array need to be sorted in ascending order. The postconditions reflect the conditions listed above and can be expressed using predicates from Listing 4.3, namely,

- Condition (5.4) becomes postcondition `result`
- Condition (5.5) becomes postcondition `left`
- Condition (5.6) postcondition property `right`

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

```
/*@
  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 5.6: Formal specification of `upper_bound`

5.2.2. Implementation of `upper_bound`

Our implementation of `upper_bound` is shown in Listing 5.7.

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;

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

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

  return right;
}
```

Listing 5.7: 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. 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).

For our purposes we have modified `equal_range`[37] to take an array of type `value_type`. Moreover, instead of a pair of iterators, our version returns a pair of indices. To be more precise, the return type of `equal_range` is the struct `size_type_pair` from Listing 5.9. Thus, the signature of `equal_range` now reads:

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

Figure 5.8 combines Figure 5.2 with Figure 5.5 in order visualize the behavior of `equal_range` for select test cases. The two types of arrows → and ⇢ represent the respective fields `first` and `second` of the return value. For values that are not contained in the array, the two arrows point to the same index. More generally, if `equal_range` returns the pair (`lb`, `ub`), then the difference `ub` – `lb` is equal to the number of occurrences of the argument `val` in the array.

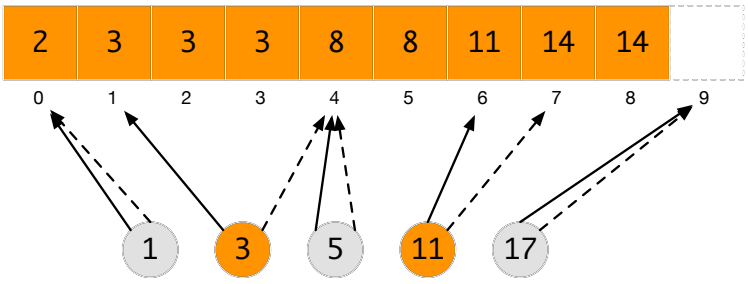


Figure 5.8.: Some examples for `equal_range`

We will provide two implementations of `equal_range` and verify both of them. The first implementation just straightforwardly calls `lower_bound` and `upper_bound` and simply returns their results (see Listing 5.11). The second, more elaborate, implementation follows the original STL code by attempting to minimize duplicate computations (see Listing 5.12).

Let (`lb`, `ub`) be the return value `equal_range`, then the conditions (5.1)–(5.6) can be merged into the inequality

$$0 \le \texttt{lb} \le \texttt{ub} \le n \tag{5.7}$$

and the following three implications for a valid index k of the array under consideration

$$0 \le k < \texttt{lb} \implies a[k] < \texttt{val} \tag{5.8}$$
$$\texttt{lb} \le k < \texttt{ub} \implies a[k] = \texttt{val} \tag{5.9}$$
$$\texttt{ub} \le k < n \implies a[k] > \texttt{val} \tag{5.10}$$

Here are some justifications for these conditions.

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

- Conditions (5.8) and (5.10) are just the Conditions (5.2) and (5.6), respectively.
- The Inequality (5.7) follows from the Inequalities (5.1) and (5.4) and the following considerations: If `ub` were less than `lb`, then according to (5.8) we would have a[`ub`] < `val`. One the other hand, we know from (5.10) that opposite inequality `val` < a[`ub`] holds. Therefore, we have `lb` ≤ `ub`.
- Condition (5.9) follows from the combination of (5.3) and (5.5) and the fact that ≤ is a total order on the integers.

5.3.1. The auxiliary function `make_pair`

The type `size_type_pair` and the `make_pair` function in Listing 5.9 are used both for the first and second implementation of `equal_range`. The specification and implementation of this simple function is shown in Listing 5.9.

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

typedef struct size_type_pair size_type_pair;


/*@
    assigns \nothing;

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

  return pair;
}
```

Listing 5.9: The type `size_pair_type` and the function `make_pair`

5.3.2. Formal specification of `equal_range`

The ACSL specification of `equal_range` is shown in Listing 5.10.

```
/*@
  requires valid: \valid_read(a + (0..n-1));
  requires sorted: 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 5.10: Formal specification of `equal_range`

The ACSL specification of `equal_range` is shown in Listing 5.10. The preconditions `sorted` expresses that the values in the (valid) array need to be sorted in ascending order. The postconditions reflect the conditions listed above and can be expressed using predicates from Listing 4.3, namely,

- Condition (5.7) becomes postcondition `result`
- Condition (5.8) becomes postcondition `left`
- Condition (5.9) becomes postcondition `middle`
- Condition (5.10) postcondition property `right`

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

5.3.3. First implementation of `equal_range`

Our first implementation of `equal_range` is shown in Listing 5.11. We call the two functions `lower_bound` and `upper_bound` and return their respective results as a 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 5.11: First 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 5.6). The benefit of this reformulation is that both the assertion `aux` and the postcondition `first <= second` can now be verified automatically.

5.3.4. Second implementation of `equal_range`

The first implementation of `equal_range` does more work than needed. STL uses a slightly more complicated implementation of `equal_range` that performs as much range reduction as possible before calling `upper_bound` and `lower_bound` on the reduced ranges. In Listing 5.12 we translated the STL implementation to C code and verified it. It is a drop-in replacement for the first implementation and implements the same formal specification, provided in Listing 5.10.

```
size_type_pair
equal_range(const value_type* a, size_type n, value_type val)
{
  size_type first  = 0;
  size_type middle = 0;
  size_type last   = n;

  /*@
    loop invariant bounds: 0 <= first <= last <= n;
    loop invariant left:   StrictUpperBound(a, 0, first, val);
    loop invariant right:  StrictLowerBound(a, last, n, val);
    loop assigns first, last, middle;
    loop variant last - first;
   */
  while (last > first) {
    middle = first + (last - first) / 2;

    if (a[middle] < val) {
      first = middle + 1;
    } else if (val < a[middle]) {
      last = middle;
    } else {
      break;
    }
  }

  if (first < last) {
    //@ assert sorted: Sorted(a, first, middle);
    size_type left = first + lower_bound(a + first, middle - first, val);
    //@ assert constant: LowerBound(a, left, middle, val);
    //@ assert strict: StrictUpperBound(a, first, left, val);

    ++middle;

    //@ assert sorted: Sorted(a, middle, last);
    size_type right = middle + upper_bound(a + middle, last - middle, val);
    //@ assert constant: UpperBound(a, middle, right, val);
    //@ assert strict: StrictLowerBound(a, right, last, val);

    return make_pair(left, right);
  } else {
    return make_pair(first, first);
  }
}
```

Listing 5.12: Second implementation of `equal_range`

Due to the higher complexity of the second implementation, additional assertions had to be added to ensure that Frama C is able to verify the correctness of the code. All of these are related to pointer arithmetic and shifting base pointers. They fall into three groups and are briefly discussed below. In order to enable the automatic verification of these properties we added the ACSL lemmas in Listing 5.13.

```
/*@
  lemma SortedShift{L}:
    \forall value_type *a, integer l, r;
      0 <= l <= r  ==>  Sorted{L}(a, l, r)  ==>  Sorted{L}(a+l, r-l);

  lemma LowerBoundShift{L}:
    \forall value_type *a, val, integer b, c, d;
      LowerBound{L}(a + b, c,     d,     val)  ==>
      LowerBound{L}(a,     c + b, d + b, val);

  lemma StrictLowerBoundShift{L}:
    \forall value_type *a, val, integer b, c, d;
      StrictLowerBound{L}(a + b, c,     d,     val)  ==>
      StrictLowerBound{L}(a,     c + b, d + b, val);

  lemma UpperBoundShift{L}:
    \forall value_type *a, val, integer b, c;
      UpperBound{L}(a+b,  0, c-b, val)  ==>
      UpperBound{L}(a,    b, c,   val);

  lemma StrictUpperBoundShift{L}:
    \forall value_type *a, val, integer b, c;
      StrictUpperBound{L}(a+b, 0, c-b, val)  ==>
      StrictUpperBound{L}(a,   b, c,   val);
*/
```

Listing 5.13: Lemmas to aid the verification of the second `equal_range` implementation

The `sorted` properties

Both `upper_bound` and `lower_bound` require that they operate on sorted data. This is also true for `equal_range`, however, inside our second implementation we need a more specific formulation, namely,

```
Sorted(a + middle, last - middle)
```

A three-argument form of the `Sorted` predicate from Listing 5.1 was added so we can spell out an intermediate step. This enables the provers to verify the preconditions of the call to `lower_bound` automatically. A similar assertion is present before the call to `upper_bound`.

The `strict` properties

Part of the post conditions of `equal_range` is that `val` is both a strict upper and a strict lower bound. However, the calls to `upper_bound` and `lower_bound` only give us

```
StrictUpperBound(a + first, 0, left - first, val)

StrictLowerBound(a + middle, right - middle, last - middle, val)
```

which is not enough to reach the desired post conditions automatically. One intermediate step for each of the assertions was sufficient to guide the prover to the desired result.

The `constant` properties

Conceptually similar to the `strict` properties the `constant` properties guide the prover towards

```
LowerBound(a, left, n, val)

UpperBound(a, 0, right, val)
```

Combining these properties allow the postcondition `middle` to be derived automatically.

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[38] 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.[39]

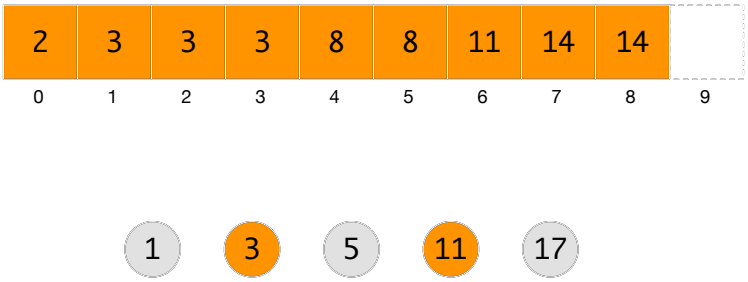


Figure 5.14.: Some examples for `binary_search`

In Figure 5.14 we do not need to use arrows to visualize the effects of `binary_search`. The colors orange and grey of the sought-after values indicate whether the algorithm returns true or false, respectively.

5.4.1. Formal specification of `binary_search`

The ACSL specification of `binary_search` is shown in Listing 5.15.

```
/*@
  requires valid:  \valid_read(a + (0..n-1));
  requires sorted: 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 5.15: Formal specification of `binary_search`

Note that instead of the somewhat lengthy existential quantification in Listing 5.15 we can use our previously introduced predicate `HasValue` (see Listing 3.4) in to achieve the more concise formal specification in Listing 5.16.

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

[39] 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 1.3) this means that `val` equals `a[i]`.

```
/*@
  requires valid:  \valid_read(a + (0..n-1));
  requires sorted: 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 5.16: Formal specification of `binary_search` using the `HasValue` predicate

It is interesting to compare this specification with that of `find` in Listing 3.5. 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.

5.4.2. Implementation of `binary_search`

Our implementation of `binary_search` is shown in Listing 5.17.

```
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 5.17: 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, §2.3.2]. Below are the algorithms we will discuss in this chapter. First, however, we introduce in Sections 6.1 and 6.2 the auxiliary predicates `Unchanged` and `MultisetUnchanged`, respectively.

- `fill` in Section 6.3 initializes each element of an array by a given fixed value.
- `swap` in Section 6.4 exchanges two values.
- `swap_ranges` in Section 6.5 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` in Section 6.6 copies a source array to a destination array.
- `copy_backward` in Section 6.7 also copies a source array to a destination array. This version, however, uses another separation condition than `copy`.
- `reverse_copy` and `reverse` in Sections 6.8 and 6.9, respectively, reverse an array. Whereas `reverse_copy` copies the result to a separate destination array, the `reverse` algorithm works in place.
- `rotate_copy` in Section 6.10 rotates a source array by `m` positions and copies the results to a destination array.
- `replace_copy` and `replace` in Sections 6.11 and 6.12, respectively, substitute each occurrence of a value by a given new value. Whereas `replace_copy` copies the result to a separate array, the `replace` algorithm works in place.
- `remove_copy` in Section 6.13 copies a source array to a destination array, but omits each occurrence of a given value. Whereas `remove_copy` copies the result to a separate array, the `remove` in Section 6.14 algorithm works in place.

  Section 6.15 describes a more precise specification of `remove_copy`. The extended function contract also captures the *stability* of `remove_copy`.

6.1. The predicate `Unchanged`

Many of the algorithms in this section iterate sequentially over one or several sequences. For the verification of such algorithms it is often important to express that a section of an array, or the complete array, have remained *unchanged*; this cannot always be expressed by an `assigns` clause. In Listing 6.1 we therefore introduce the overloaded predicate `Unchanged` together with some simple lemmas. The expression `Unchanged{K,L}(a,f,l)` is true if the range `a[f..l-1]` in state `K` is element-wise equal to that range in state `L`.

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

  predicate
    Unchanged{K,L}(value_type* a, integer n) = Unchanged{K,L}(a, 0, n);
*/
```

Listing 6.1: The predicate `Unchanged`

We also provide a few lemmas for `Unchanged` that we need for the verification of some algorithms.

Lemma `UnchangedSection` in Listing 6.2 states that if the range `a[0..n-1]` does not change when going from state `K` to state `L`, then the subrange `a[0..m-1]` does not change either.

```
/*@
  lemma
    UnchangedSection{K,L}:
      \forall value_type *a, integer m, n, p, q;
         m <= p <= q <= n          ==>
         Unchanged{K,L}(a, m, n)   ==>
         Unchanged{K,L}(a, p, q)   ==>
         \at(a[m],K) == \at(a[m], L);
*/
```

Listing 6.2: The lemma `UnchangedSection`

Lemma `UnchangedStep` in Listing 6.3 expresses the simple fact that "unchangedness" is an inductive property.

```
/*@
  lemma
    UnchangedStep{K,L}:
      \forall value_type *a, integer n;
        Unchanged{K,L}(a, n)         ==>
        \at(a[n],K) == \at(a[n],L)   ==>
        Unchanged{K,L}(a, n+1);
*/
```

Listing 6.3: The lemma `UnchangedStep`

Lemma `UnchangedTransitive` in Listing 6.4 expresses the transitivity of `Unchanged` with respect to program states.

```
/*@
  lemma UnchangedTransitive{K,L,M}:
    \forall value_type *a, integer n;
      Unchanged{K,L}(a, n)  ==>
      Unchanged{L,M}(a, n)  ==>
      Unchanged{K,M}(a, n);
*/
```

Listing 6.4: The lemma `UnchangedTransitive`

6.2. The predicate `MultisetUnchanged`

Various algorithms in this document *rearrange* or *reorder* the elements of a given range such that the number of each element remains unchanged. In other words, reordering leaves the *multiset*[40] of elements in the range unchanged.

We use the predicate `MultisetUnchanged`, defined in Listing 6.5, to formally describe this property. This predicate, which is given in two overloaded versions, relies on the logic function `Count` that is defined in Listing 3.34.

```
/*@
  predicate
    MultisetUnchanged{L1,L2}(value_type* a, integer first, integer last) =
      \forall value_type v;
        Count{L1}(a, first, last, v) == Count{L2}(a, first, last, v);


  predicate
    MultisetUnchanged{L1,L2}(value_type* a, integer n) =
      MultisetUnchanged{L1,L2}(a, 0, n);
*/
```

Listing 6.5: The predicate `MultisetUnchanged`

[40] See `http://en.wikipedia.org/wiki/Multiset`

6.3. 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[41] to that of an array of type `value_type`. The signature now reads:

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

6.3.1. Formal specification of `fill`

Listing 6.6 shows the formal specification of `fill` in ACSL. We can express the postcondition of `fill` simply by using the predicate `ConstantRange` from Listing 3.25.

```
/*@
  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 6.6: Formal specification of `fill`

The `assigns`-clauses formalize that `fill` modifies only the entries of the range `a[0..n-1]`. In general, when more than one *assigns clause* appears in a function's specification, it is permitted to modify any of the referenced memory locations. However, if no *assigns clause* appears at all, the function is free to modify any memory location, see [9, §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.

6.3.2. Implementation of `fill`

Listing 6.7 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 6.7: Implementation of `fill`

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 3.25.

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

6.4. The `swap` algorithm

The `swap` algorithm[42] in the C++ standard library exchanges the contents of two variables. Similarly, the `iter_swap` algorithm[43] 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`.

6.4.1. Formal specification of `swap`

The ACSL specification for the `swap` function is shown in Listing 6.8. The preconditions are formalized by the `requires`-clauses which state that both pointer arguments of the `swap` function must be dereferenceable.

```
/*@
  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 6.8: Formal specification of `swap`

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.

6.4.2. Implementation of `swap`

Listing 6.9 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 6.9: Implementation of `swap`

[42] See `http://www.sgi.com/tech/stl/swap.html`
[43] See `http://www.sgi.com/tech/stl/iter_swap.html`

6.5. The `swap_ranges` algorithm

The `swap_ranges` algorithm[44] 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`.

6.5.1. Formal specification of `swap_ranges`

Listing 6.10 shows an ACSL specification for the `swap_ranges` algorithm.

```
/*@
  requires valid:  \valid(a + (0..n-1));
  requires valid:  \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:  EqualRanges{Here,Old}(a, n, b);
  ensures equal:  EqualRanges{Old,Here}(a, n, b);
*/
void swap_ranges(value_type* a, size_type n, value_type* b);
```

Listing 6.10: 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 3.15) together with the label `Old` and `Here` to express the postcondition of `swap_ranges`. In our specification in Listing 6.10, 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.

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

6.5.2. Implementation of `swap_ranges`

Listing 6.11 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:  EqualRanges{Here,Pre}(a, i, b);
    loop invariant equal:  EqualRanges{Here,Pre}(b, i, a);

    loop invariant unchanged:  Unchanged{Here,Pre}(a, i, n);
    loop invariant unchanged:  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 6.11: Implementation of `swap_ranges`

For the postcondition of the specification in Listing 6.10 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 6.1 is used to express this property.

6.6. 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[45] 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[0..n-1] to the destination range b[0..n-1], as shown in Figure 6.12.

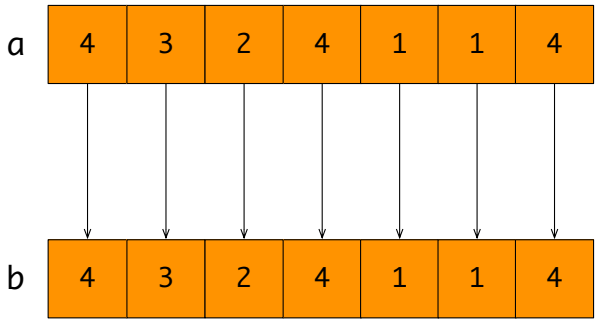


Figure 6.12.: Effects of copy

6.6.1. Formal specification of copy

Figure 6.12 might suggest that the ranges a[0..n-1] and b[0..n-1] must not overlap. However, since the informal specification requires that elements are copied in the order of increasing indices only a weaker condition is necessary. To be more specific, it is required that the pointer b does not refer to elements of a[0..n-1] as shown in the example in Figure 6.13.

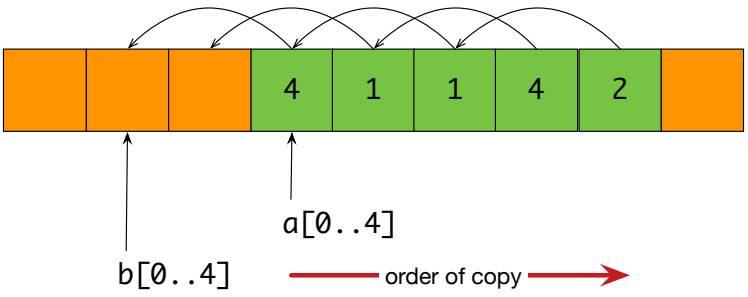


Figure 6.13.: Possible overlap of copy ranges

The ACSL specification of copy is shown in Listing 6.14. The copy algorithm expects that the ranges a and b are valid for reading and writing, respectively. Note the precondition sep that expresses the previously discussed non-overlapping property.

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

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

  assigns b[0..n-1];

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

Listing 6.14: Formal specification of `copy`

Again, we can use the `EqualRanges` predicate from Section 3.5 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.

6.6.2. Implementation of `copy`

Listing 6.15 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{Pre,Here}(a, i, b);
    loop invariant equal: Unchanged{Pre,Here}(a, i, n);
    loop assigns   i, b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i) {
    b[i] = a[i];
  }
}
```

Listing 6.15: Implementation of `copy`

For the postcondition `equal` to be true, we must ensure that for every index `i`, the value `a[i]` must not yet have been changed before it is copied to `b[i]`. We express this by using the `Unchanged` predicate.[46]

The `assigns` clause ensures that nothing but the range `b[0..n-1]` and the loop variable `i` is modified. Keep in mind, however, that parts of the source range `a[0..n-1]` might change due to its potential overlap with the destination range.

[46] Alternatively, this could also be expressed by changing the `loop assigns` clause to `i, b[0..i-1]`; however, Frama-C doesn't yet support `loop assigns` clauses containing the loop variable.

6.7. The `copy_backward` algorithm

The `copy_backward` algorithm in the C++ Standard Library implements another duplication algorithm for general sequences. For our purposes we have modified the generic implementation[47] to that of a range of type `value_type`. The signature now reads:

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

The main reason for the existence of `copy_backward` is to allow copying when the start of the destination range `a[0..n-1]` is contained in the source range `b[0..n-1]`. In this case, `copy` can't be employed since its precondition `sep` is violated, as can be see in Listing 6.14.

The informal specification of `copy_backward` states that copying starts at the end of the source range. For this to work, however, the pointer `b+n` must not be contained in the source range. Note that the order of operation (or procedure) calls cannot be specified in ACSL.[48] A similar remark about order of operations tacitly applied to earlier functions as well, e.g. to `copy`, where the C++ order was prescribed by confining the signature to a `ForwardIterator`.

Figure 6.16 gives an example where `copy_backward`, but *not* `copy`, can be applied.

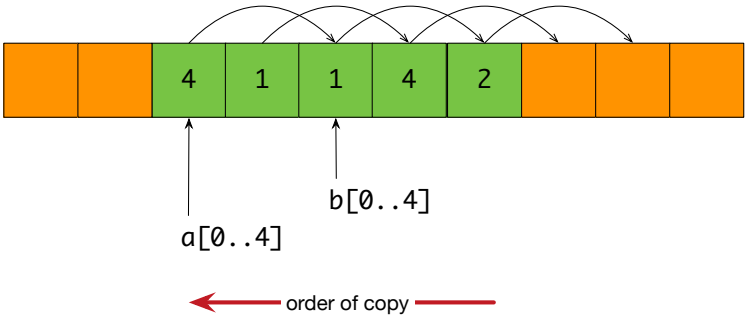


Figure 6.16.: Possible overlap of `copy_backward` ranges

Note that in the original signature the argument `b` refers to one past the end of the destination range. Here, however, it refers to its start. The reason for this change is that in C++ `copy_backward` is defined for *bidirectional iterators* which do not provide random access operations such as adding or subtracting an index. Since our C version works on pointers we do not consider it as necessary to use the one past the end pointer.

[47] See `http://www.sgi.com/tech/stl/copy_backward.html`

[48] The Aoraï specification language and the corresponding Frama-C plugin are provided to specify and verify temporal properties of code; however, they are beyond the scope of this tutorial.

6.7.1. Formal specification of `copy_backward`

The ACSL specification of `copy_backward` is shown in Listing 6.17. The `copy_backward` algorithm expects that the ranges `a[0..n-1]` and `b[0..n-1]` are valid for reading and writing, respectively. Precondition `sep` formalizes the constraints on the overlap of the source and destination ranges as discussed at the beginning of this section.

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

  assigns  b[0..n-1];

  ensures EqualRanges{Old,Here}(a, n, b);
*/
void copy_backward(const value_type* a, size_type n, value_type* b);
```

Listing 6.17: Formal specification of `copy_backward`

The function `copy_backward` 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 3.5 to express that the array `a` equals `b` after `copy_backward` has been called.

6.7.2. Implementation of `copy_backward`

Listing 6.18 shows an implementation of the `copy_backward` function.

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

Listing 6.18: Implementation of `copy_backward`

We have loop invariants similar to `copy`, stating the loop variable's range (`bound`) and the area that has already been copied in each cycle (`equal`).

6.8. The `reverse_copy` algorithm

The `reverse_copy`[49] algorithm of the C++ Standard Library inverts 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 implementation 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);
```

6.8.1. The predicate `Reverse`

Informally, `reverse_copy` copies the elements from the array `a` into array `b` such that the copy is a reverse of the original array. In order to concisely formalize these conditions we define the overloaded predicates `Reverse` that are shown in Listing 6.19.

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

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

  predicate
    Reverse{A,B}(value_type* a, integer n) = Reverse{A,B}(a, n, a);

*/
```

Listing 6.19: The predicate `Reverse`

Figure 6.20 graphically represents the first version of predicate `Reverse` in Listing 6.19.

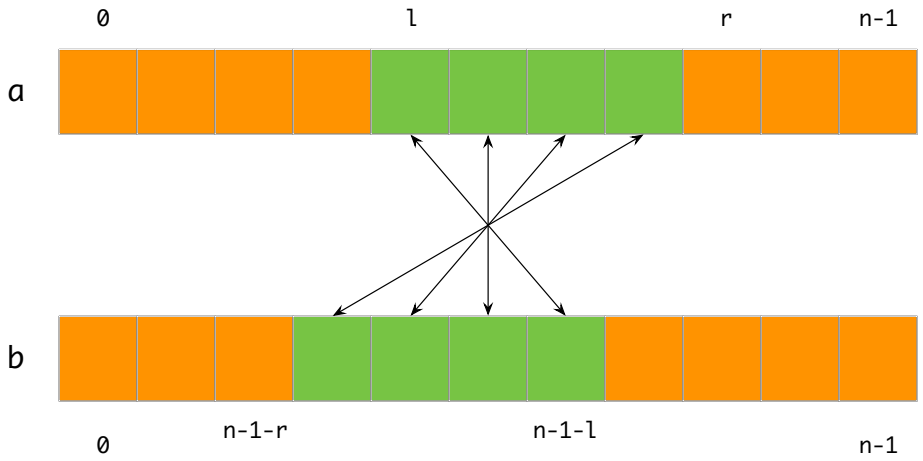


Figure 6.20.: Sketch of predicate `Reverse`

The second version of `Reverse` in Listing 6.19 captures the reverse relation between two complete arrays while the third version is useful when a single array shall be reversed (see Section 6.9).

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

6.8.2. Formal specification of `reverse_copy`

The ACSL specification of `reverse_copy` is shown in Listing 6.21. We use the second version of predicate `Reverse` from Listing 6.19 in order to formulate the postcondition of `reverse_copy`.

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

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

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

Listing 6.21: Formal specification of `reverse_copy`

6.8.3. Implementation of `reverse_copy`

Listing 6.22 shows an implementation of the `reverse_copy` function. 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 invariant `reverse` where we employ the first version of `Reverse` from Listing 6.19.

```
void reverse_copy(const value_type* a, size_type n, value_type* b)
{
  /*@
    loop invariant bound:   0 <= i <= n;
    loop invariant reverse: Reverse{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 6.22: Implementation of `reverse_copy`

6.9. The `reverse` algorithm

The `reverse`[50] algorithm of the C++ Standard Library inverts 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 implementation to that of a range of type `value_type`. The signature now reads:

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

6.9.1. Formal specification of `reverse`

The ACSL specification for the `reverse` function is shown in listing 6.23. In the postcondition we use the third version of predicate `Reverse` from Listing 6.19.

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

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

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

Listing 6.23: Formal specification of `reverse`

6.9.2. Implementation of `reverse`

Listing 6.24 shows an implementation of the `reverse` function. Since the `reverse` algorithm operates *in place* we use the `swap` function from Section 6.4 in order to exchange the elements of the first half of the array with the corresponding elements of the second half.

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

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

    loop invariant left:   Reverse{Here,Pre}(a, n, a, 0,  i);
    loop invariant middle: Unchanged{Here,Pre}(a, i, n-i);
    loop invariant right:  Reverse{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) {
    swap(&a[i], &a[n - 1 - i]);
  }
}
```

Listing 6.24: Implementation of `reverse`

We reuse the predicates `Reverse` (Listing 6.19) and `Unchanged` (Listing 6.1) in order to write concise loop invariants.

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

6.10. The `rotate_copy` algorithm

The `rotate_copy` algorithm in the C++ Standard Library rotates a sequence downwards by `m` positions and copies the results to another same-sized sequence. For our purposes we have modified the generic implementation[51] 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 last `n-m` elements of the array `a[0..n-1]` are moved `m` places downwards and stored as the first `n-m` elements of the array `b[0..n-1]`, whereas the first `m` elements of the array `a[0..n-1]` are wrapped around and stored as the last `m` elements of the array `b[0..n-1]`

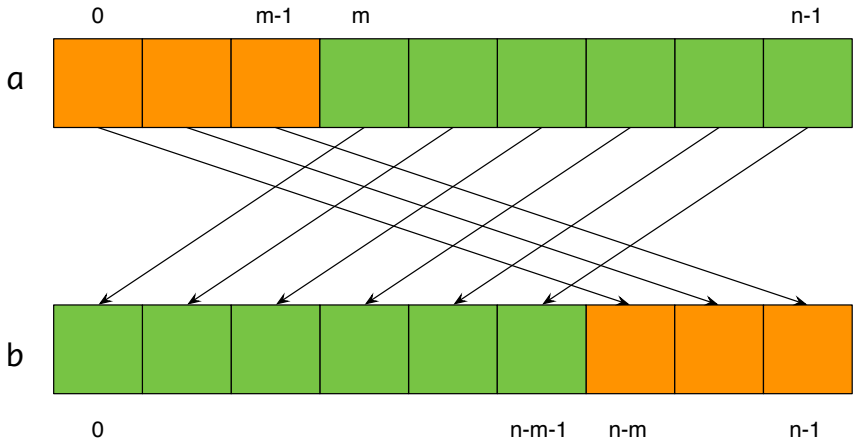


Figure 6.25.: Effects of `rotate_copy`

Figure 6.25 illustrates the effects of `rotate_copy` by highlighting how the initial and final segments of the array `a[0..n-1]` are mapped to corresponding segments of the array `b[0..n-1]`.

6.10.1. Formal specification of `rotate_copy`

The ACSL specification of `rotate_copy` is shown in Listing 6.26.

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

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

  ensures equal_first: EqualRanges{Here,Here}(a,   m,   b+(n-m));
  ensures equal_last:  EqualRanges{Here,Here}(a+m, n-m, b);
  ensures unchanged:   Unchanged{Old,Here}(a, n);
*/
void rotate_copy(const value_type* a, size_type m, size_type n, value_type* b);
```

Listing 6.26: Formal specification of `rotate_copy`

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

6.10.2. Implementation of `rotate_copy`

Listing 6.27 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 6.27: Implementation of `rotate_copy`

6.11. The `replace_copy` algorithm

The `replace_copy` algorithm of the C++ Standard Library substitutes specific elements from general sequences. Here, the general implementation[52] 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 v, value_type w);
```

The `replace_copy` algorithm copies the elements from the range `a[0..n]` to range `b[0..n]`, substituting every occurrence of `v` by `w`. 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.

6.11.1. The predicate `Replace`

We start with defining in Listing 6.28 the predicate `Replace` that describes the intended relationship between the input array `a[0..n-1]` and the output array `b[0..n-1]`. Note the introduction of *local bindings* `\let ai = ...` and `\let bi = ...` in the definition of `Replace` (see [9, §2.2]).

```
/*@
  predicate
    Replace{K,L}(value_type* a, integer n, value_type* b,
                 value_type v, value_type w) =
       \forall integer i; 0 <= i < n ==>
         \let ai = \at(a[i],K); \let bi = \at(b[i],L);
              (ai == v ==> bi == w) && (ai != v ==> bi == ai) ;


  predicate
    Replace{K,L}(value_type* a, integer n, value_type v, value_type w) =
      Replace{K,L}(a, n, a, v, w);
*/
```

Listing 6.28: The predicate `Replace`

Listing 6.28 also contains a second, overloaded version of `Replace` which we will use for the specification of the related in-place algorithm `replace` in Section 6.12.

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

6.11.2. Formal specification of `replace_copy`

Using predicate `Replace` the ACSL specification of `replace_copy` is as simple as in Listing 6.29. Note that we require that the pointer `b` does not refer to elements of the source range `a[0..n-1]` (see also Section 6.6).

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

  assigns b[0..n-1];

  ensures replace:   Replace{Old,Here}(a, n, b, v, w);
  ensures result:    \result == n;
*/
size_type
replace_copy(const value_type* a, size_type n, value_type* b, value_type v,
             value_type w);
```

Listing 6.29: Formal specification of the `replace_copy`

6.11.3. Implementation of `replace_copy`

An implementation (including loop annotations) of `replace_copy` is shown in Listing 6.30. Note how the structure of the loop annotations resembles the specification of Listing 6.29.

```
size_type
replace_copy(const value_type* a, size_type n, value_type* b, value_type v,
             value_type w)
{
  /*@
    loop invariant bounds:    0 <= i <= n;
    loop invariant replace:   Replace{Pre,Here}(a, i, b, v, w);
    loop invariant unchanged: Unchanged{Pre,Here}(a, i, n);
    loop assigns i, b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i) {
    b[i] = (a[i] == v ? w : a[i]);
  }

  return n;
}
```

Listing 6.30: Implementation of the `replace_copy` algorithm

6.12. The `replace` algorithm

The `replace` algorithm of the C++ Standard Library substitutes specific values in a general sequence. Here, the general implementation[53] has been altered to process `value_type` ranges. The new signature reads

```
void replace(value_type* a, size_type n, value_type v, value_type w);
```

The `replace` algorithm substitutes all elements from the range `a[0..n-1]` that equal `v` by `w`.

6.12.1. Formal specification of `replace`

Using the second predicate `Replace` from Listing 6.28 the ACSL specification of `replace` can be expressed as in Listing 6.31.

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

  assigns a[0..n-1];

  ensures replace:  Replace{Old,Here}(a, n, v, w);
*/
void replace(value_type* a, size_type n, value_type v, value_type w);
```

Listing 6.31: Formal specification of the `replace`

6.12.2. Implementation of `replace`

An implementation of `replace` is shown in Listing 6.32. The loop invariant `unchanged` expresses that when entering iteration `i` the elements `a[i..n-1]` have not yet changed.

```
void replace(value_type* a, size_type n, value_type v, value_type w)
{
  /*@
    loop invariant bounds:    0 <= i <= n;
    loop invariant replace:   Replace{Pre,Here}(a, i, v, w);
    loop invariant unchanged: Unchanged{Pre,Here}(a, i, n);
    loop assigns i, a[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i) {
    if (a[i] == v) {
      a[i] = w;
    }
  }
}
```

Listing 6.32: Implementation of the `replace` algorithm

[53] See `http://www.sgi.com/tech/stl/replace.html`

6.13. 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.[54] 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`.

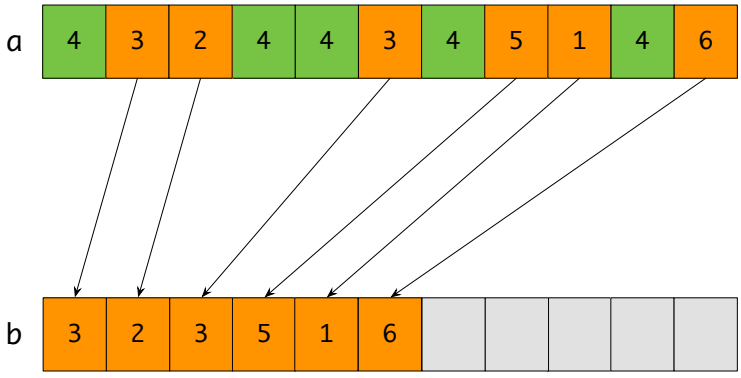


Figure 6.33.: Effects of `remove_copy`

Figure 6.33 shows how `remove_copy` is supposed to copy elements that differ from `v` from range `a` to `b`.

6.13.1. The predicate `MultisetRetainRest`

In order to achieve a concise specification we introduce the auxiliary predicate `MultisetRetainRest` (see Listing 6.34). The expression `MultisetRetainRest{A,B}(a, m, b, n, v)` is true if the range `a[0..n-1]` at time `A` contains the same elements as `b[0..m-1]` at time `B`, except possibly for occurrences of `v`; the elements' order may differ in `a` and `b`.

```
/*@
  predicate
    MultisetRetainRest{A,B}(value_type* a, integer m,
                            value_type* b, integer n,
                            value_type  v) =

        \forall value_type x;
          x != v ==> Count{A}(a, m, x) == Count{B}(b, n, x);
*/
```

Listing 6.34: The predicate `MultisetRetainRest`

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

6.13.2. Formal specification of `remove_copy`

Listing 6.35 now shows our first attempt to specify `remove_copy`.

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

  assigns  b[0..n-1];

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

Listing 6.35: Formal specification of `remove_copy`

- The informal specification of `remove_copy` states that the pointer `b` does not reference the range `a[0..n-1]`. This is expressed by demanding `\separated(a + (0..n-1), b)`.
- We use the predicate `MultisetRetainRest` to express that the number of elements different from `v` is the same in the source and target range.

  Note that this property does not guarantee the stability of `remove_copy` because given e.g. a range $\{1,0,5,2,0,5\}$ and the value `v=0` the expected result of `remove_copy` is the range $\{1,5,2,5\}$. However, since `Count` is invariant under permutations the specification in Listing 6.35 would also allow e.g. the result $\{5,5,1,2\}$. In Section 6.15 we will discuss how the stability of `remove_copy` can be captured in an ACSL specification.
- The predicate `Unchanged` from Listing 6.1 is used to express that `remove_copy` does not change `b[\result..n-1]`.
- Note the re-use of predicate `HasValue` (Listing 3.4) to express that the target range does not contain the value `v`.

6.13.3. Implementation of `remove_copy`

An implementation of `remove_copy` is shown in Listing 6.36.

```
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 size:       j == i - Count{Pre}(a, i, v);
    loop invariant unchanged:  Unchanged{Pre,Here}(a, i, n);
    loop invariant retain:     MultisetRetainRest{Pre,Here}(a, i, b, j, v);
    loop invariant discard:    !HasValue(b, j, v);
    loop invariant unchanged:  Unchanged{Pre,Here}(b, j, n);
    loop assigns i, j, b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i) {
    if (a[i] != v) {
      b[j++] = a[i];
      //@ assert retain_pre:  MultisetRetainRest{Pre,Here}(a, i,   b, j-1, v);
      //@ assert retain_now:  MultisetRetainRest{Pre,Here}(a, i+1, b, j,   v);
    }
  }

  return j;
}
```

Listing 6.36: Implementation of `remove_copy`

Not surprisingly, the logical function `Count` as well as the predicates `HasValue`, `Unchanged`, and `MultisetRetainRest` also appear in the loop invariants of `remove_copy`. In a previous release of this manual, a series of complex assertions was necessary in order to guide the provers towards the loop invariant `retain`. In this version we still need the assertions `retain_pre` and `retain_now` to automatically verify the said loop invariant in a reasonable time.

Since the precondition `sep` does not guarantee that the range `a[0..n-1]` remains unchanged we have to refer to the pre-state of the array `a[0..n-1]` by using the logic label `Pre` instead of `Here` for all references of `a` where possible. This is necessary despite `a[i]` retaining its original value until the `i`-th iteration—a fact that is quickly verified through the loop invariant `unchanged`.

6.14. The `remove` algorithm

Besides the `remove_copy` function, the C++ Standard Library also contains a function `remove` performing the same operation as `remove_copy`, but in place. Its signature is very similar to that of `remove_copy`, except that the `b` is missing since `a` is modified in place:

```
size_type remove(value_type* a, size_type n, value_type v);
```

Figure 6.37 shows how `remove` is supposed to remove all occurrences of a given value from a range.

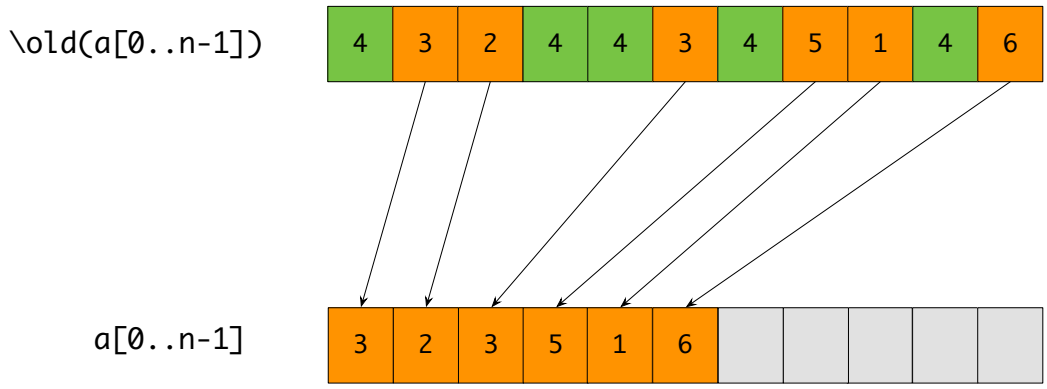


Figure 6.37.: Effects of `remove`

As it turns out, we can reuse the function `remove_copy` to implement `remove`.

6.14.1. Formal specification of `remove`

Listing 6.38 shows the formal specification of `remove`. It is very similar to the specification of `remove_copy` (see Listing 6.35) except all references to `b` have been turned into references to the pre-state of `a` and the unchangedness of `a` being no longer asserted.

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

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

  ensures  bound:      0 <= \result <= n;
  ensures  size:       \result == n - Count{Old}(a, n, v);
  ensures  retain:     MultisetRetainRest{Old,Here}(a, n, a, \result, v);
  ensures  discard:    !HasValue(a, \result, v);
  ensures  unchanged:  Unchanged{Old,Here}(a, \result, n);
*/
size_type remove(value_type* a, size_type n, value_type v);
```

Listing 6.38: Formal specification of `remove`

Informal verification of an implementation of `remove`

Our implementation of `remove` closely follows those of the original STL implementation[55] by calling the algorithms `find` and `remove_copy`. Without additional assertions that simplify the formal verification the code looks likes shown here.

```
size_type remove(value_type* a, size_type n, value_type v)
{
  size_type first = find(a, n, v);

  if (first == n) {
    return n;
  } else {
    return first + remove_copy(a + first + 1, n - first - 1, a + first, v);
  }
}
```

Informally, the implementation starts with a call to the function `find` (Listing 3.5) to check whether the range `a[0..n-1]` contains the value `v`.

1. If `v` does not occur in the range, then nothing needs to be removed and the algorithms terminates.
2. Otherwise `find` returns the `first` occurrence of `v` in `a[0..n-1]`. We then call `remove_copy` (cf. Listing 6.35) on the (possibly empty) subrange `a[first+1..n-1]` and write its output to the subrange that starts at `a[first]`. Note that the starting address `a + first` of the output range of `remove_copy` does not point into the input range `a[first+1..n-1]` and therefore preserves the precondition `sep` of `remove_copy`.

   Considering the return value `m` of `remove_copy`, we know the following facts.

   a) The resulting subrange `a[0..first-1]` remains unchanged and does not contain `v`.

   b) The occurrence of `v` at `\old(a[first])` is overwritten by `remove_copy` with a different value. In fact, `remove_copy` will remove all occurrences of `v` from `\old(a[first+1..n-1)` and put its output into the range `a[first..m-1]`.

   c) Thus, the range `a[0..m-1]`, which is the concatenation of the ranges of `a[0..first-1]` and `a[first..m-1]`, contains the desired result.

[55] See `https://www.sgi.com/tech/stl/stl_algo.h`

6.14.2. Implementation of `remove`

Listing 6.39 shows our implementation of `remove` together with additional assertions whose purpose is to guide the automatic provers to a formal proof of the informal discussion above. Note that we have added more assertions so that we can explain some particular points of the verification more easily.

```
size_type remove(value_type* a, size_type n, value_type v)
{
  size_type first = find(a, n, v);

  if (first == n) {

    //@ assert discard_nothing:  !HasValue(a, n, v);
    //@ assert size_zero:       Count{Pre}(a, n, v) == 0;
    return n;
  } else {

    size_type m = remove_copy(a + first + 1, n - first - 1, a + first, v);

    //@ assert discard_first:  !HasValue(a, 0, first, v);
    //@ assert discard_remove: !HasValue(a + first, 0, m, v);
    //@ assert discard_remove: !HasValue(a, first, first + m, v);
    //@ assert discard_union:  !HasValue(a, first + m, v);

    //@ assert size_first:  1 == Count{Pre}(a, 0, first + 1, v);
    //@ assert size_remove: m == n - first - 1 - Count{Pre}(a, first + 1, n, v);
    //@ assert size_union:  first + m == n - Count{Pre}(a, 0, n, v);

    //@ assert retain:    MultisetRetainRest{Pre,Here}(a, n, a, first + m, v);
    //@ assert unchanged: Unchanged{Pre,Here}(a, first + m, n);
    return first + m;
  }
}
```

Listing 6.39: Implementation of `remove`

We now have a closer look at the assertions `size_zero`, `discard_remove` and `size_remove`.

The assertion `size_zero`

The assertion `discard_nothing` in the then-part of our if-statement is a direct consequence of the call of `find`. In order to verify the postcondition `size`, however, the assertion `count_zero` is more helpful. It is fairly easy for a human being to understand the following fact:

> If the range `a[0..n-1]` does not contain the value `v`, then the number of occurrences of `v` in the range is zero.

However, in order to enable WP that it can derive the property `size_zero` from the property `discard_nothing` we had to

1. formalize the above implication as the ACSL lemma `HasValueCountInversion` (shown in Listing 6.40
2. provide a proof of the lemma through induction on the length of the range.

```
/*@
  lemma HasValueCountInversion{L}:
    \forall value_type *a, v, integer m, n;
      !HasValue(a, m, n, v)  ==>  Count(a, m, n, v) == 0;
*/
```

Listing 6.40: The lemma `HasValueCountInversion`

The assertions `discard_remove`

There are two assertions named `discard_remove` in Listing 6.39. The first one is a direct consequence of the postcondition `discard` of `remove_copy` (cf. Listing 6.35). The second assertion can be derived form the first one using the lemma `HasValueShiftInversion` in Listing 6.41.

```
/*@
   lemma HasValueShiftInversion{L}:
     \forall value_type *a, v, integer m, n;
       !HasValue(a + m, 0, n, v)  ==>  !HasValue(a, m, m + n, v);
*/
```

Listing 6.41: The lemma `HasValueShiftInversion`

The assertion `size_remove`

The postcondition `size` of the call to `remove_copy` in the else-branch of our implementation reads as follows.

```
n - Count{Old}(a + first + 1, n - first - 1, v)
```

For the purpose of our verification it is, however, the assertion `size_remove` is more useful. We added the lemma `CountShift` from Listing 6.42 in order to facilitate the derivation of the assertion `size_remove` .

```
/*@
  lemma CountShift{L}:
     \forall value_type *a, v, integer m, n;
        0 <= m  ==>  0 <= n  ==>
        Count(a + m, 0, n, v) == Count(a, m, m + n, v);
*/
```

Listing 6.42: The lemma `CountShift`

6.15. Capturing the stability of `remove_copy`

In this section, we have a closer look at the *stability* of `remove_copy` and its expression in the ACSL language.

Figure 6.43 shows, with respect to array indices, how the elements different from `v` "slide" to smaller or equal positions. The main observation here is that an element slides as many positions *down* as there are occurrences of `v` *below* it.

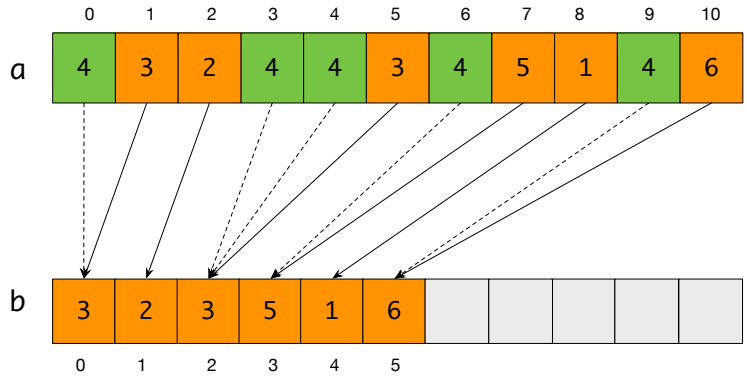


Figure 6.43.: Stability of `remove_copy` with respect to indices

As it turns out, it is relatively easy to express this property using the previously introduced logic function `Count` (see Listing 3.34 in Section 3.9). We simply define in Listing 6.44 a logic function `RemoveCount` which subtracts from every position `i` the number of occurrences of `v` below `i`. In addition, we provide four lemmas that easily follow from corresponding properties of `Count`, but are needed for the verification.

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

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

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

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

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

Listing 6.44: The logic function `RemoveCount`

The value `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`.

Also, note that `RemoveCount` is defined for all integers, including those indices `i` where `a[i]` equals `v` (see the dashed lines in Figure 6.43). In the specification of `remove_copy` we will, however, only use `RemoveCount` for indices where `a[i]` is different from `v`. This can be seen in the definition of predicate `RemoveMapping` (Listing 6.45) that uses `RemoveCount` to formally capture the stability with respect to corresponding elements of the source and target ranges.

```
/*@
  predicate
    RemoveMapping{K,L}(value_type* a, integer n, value_type* b, value_type v) =
      \forall integer i; 0 <= i < n ==>
        \at(a[i],K) != v ==> \at(b[RemoveCount{K}(a, i, v)],L) == \at(a[i],K);
*/
```

Listing 6.45: The predicate `RemoveMapping`

6.15.1. Formal specification of `remove_copy`

Listing 6.46 shows an improved specification of `remove_copy` that also captures the required stability.

```
/*@
  requires valid:  \valid_read(a + (0..n-1));
  requires valid:  \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  size:       \result == RemoveCount(a, n, v);
  ensures  retain:     MultisetRetainRest{Here,Here}(a, n, b, \result, v);
  ensures  discard:    !HasValue(b, \result, v);
  ensures  mapping:    RemoveMapping{Here,Here}(a, n, b, v);
  ensures  unchanged:  Unchanged{Old,Here}(b, \result, n);
  ensures  unchanged:  Unchanged{Old,Here}(a, n);
*/
size_type
remove_copy(const value_type* a, size_type n, value_type* b, value_type v);
```

Listing 6.46: Improved formal specification of `remove_copy`

This specification of `RemoveCount` differs from the one in Listing 6.35 in the following points.

1. We now use `RemoveCount` in order to specify the expected return value in postcondition `result`.
2. We use the predicate `RemoveMapping` in the new postcondition `mapping`. Here we exactly specify to which element in the output range an element of the input range, that is different from `v`, is copied.

6.15.2. The auxiliary function `remove_copy_aux`

When trying to verify the extended contract of Listing 6.46 we had to add more assertions than those of Listing 6.36. In order to better manage the annotations related to the if-statement inside the implementation of `remove_copy` we created an auxiliary function `remove_copy_aux` (Listing 6.47) with a proper function contract.

```
/*@
  requires valid:    \valid_read(a + (0..n-1));
  requires valid:    \valid(b + (0..n-1));
  requires sep:      \separated(a + (0..n-1), b+(0..n-1));
  requires bound:    0 <= i < n;
  requires bound:    0 <= j < n;
  requires size:     j == RemoveCount(a, i, v);
  requires discard:  !HasValue(b, j, v);
  requires retain:   MultisetRetainRest{Here,Here}(a, i, b, j, v);
  requires mapping:  RemoveMapping{Here,Here}(a, i, b, v);

  assigns  b[j];

  ensures bound:      0 <= \result <= n;
  ensures size:       \result == RemoveCount(a, i+1, v);
  ensures discard:    !HasValue(b, \result, v);
  ensures retain:     MultisetRetainRest{Old,Here}(a, i+1, b, \result, v);
  ensures mapping:    RemoveMapping{Old,Here}(a, i+1, b, v);
  ensures unchanged:  Unchanged{Old,Here}(b, \result, n);

  behavior active:
    assumes a[i] != v;
    assigns b[j];
    ensures retain:  b[\old(j)] == a[i];

  behavior passive:
    assumes a[i] == v;
    assigns \nothing;

  complete behaviors;
  disjoint behaviors;
*/
static inline size_type
remove_copy_aux(const value_type* a, size_type n, size_type i,
                value_type* b, size_type j, value_type v)
{
  //@ assert retain:  MultisetRetainRest{Pre,Here}(a, i, b, j, v);
  if (a[i] != v) {
    b[j] = a[i];
    //@ assert size:        j+1 == RemoveCount(a, i+1, v);
    //@ assert mapping:     RemoveMapping{Pre,Here}(a, i+1, b, v);
    //@ assert retain_pre:  MultisetRetainRest{Pre,Here}(a, i,   b, j,   v);
    //@ assert retain_now:  MultisetRetainRest{Pre,Here}(a, i+1, b, j+1, v);
    return j + 1;
  } else {
    //@ assert size:     j == RemoveCount(a, i+1, v);
    //@ assert retain:   MultisetRetainRest{Pre,Here}(a, i+1, b, j, v);
    //@ assert mapping:  RemoveMapping{Pre,Here}(a, i+1, b, v);
    return j;
  }
}
```

Listing 6.47: Auxiliary function `remove_copy_aux`

Despite the elaborate specification of `remove_copy_aux` we had to add assertions to the (modified) if-statement. In particular, we encounter here again the pair of assertions `retain_pre` and `retain_now` that we have already seen in the first implementation of `remove_copy` (cf. Listing 6.36).

Moreover, to ensure the verify the `retain` properties of `remove_copy_aux` we formulated the lemma `MultisetRetainRestMiss` which is shown in Listing 6.48.

```
/*@
  lemma MultisetRetainRestMiss{K,L}:
    \forall value_type *a, *b, v, integer m, n;
       0 <= m  ==>  0 <= n                        ==>
       n == RemoveCount{K}(a, m, v)               ==>
       MultisetRetainRest{K,K}(a, m, b, n, v)  ==>
       \at(a[m],K) != v                           ==>
       \at(a[m],L) == \at(b[n],L)                 ==>
       Unchanged{K,L}(a,m+1)                      ==>
       Unchanged{K,L}(b,n)                        ==>
       n+1  == RemoveCount{L}(a, m+1, v)          ==>
       MultisetRetainRest{K,L}(a, m+1, b, n+1, v);
*/
```

Listing 6.48: The lemma `MultisetRetainRestMiss`

6.15.3. Implementation of `remove_copy`

Listing 6.49 shows an implementation of `remove_copy` that calls the helper function `remove_copy_aux` from Listing 6.47. The use of this helper function simplifies the verification of this more precise specification 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 size:       j == RemoveCount{Pre}(a, i, v);
    loop invariant discard:    !HasValue(b, j, v);
    loop invariant retain:     MultisetRetainRest{Pre,Here}(a, i, b, j, v);
    loop invariant mapping:    RemoveMapping{Here,Here}(a, i, b, v);
    loop invariant unchanged:  Unchanged{Pre,Here}(b, j, n);

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

  //@ assert size:    j == RemoveCount{Pre}(a, n, v);
  //@ assert retain:  MultisetRetainRest{Pre,Here}(a, n, b, j, v);
  //@ assert retain:  MultisetRetainRest{Here,Here}(a, n, b, j, v);
  return j;
}
```

Listing 6.49: Implementation of `remove_copy` with additional loop invariants

In order to prove the loop invariant `mapping` we rely also on monotonicity properties of `Count` and `RemoveCount` that are shown in Listings 6.50 and 6.51, respectively.

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

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

Listing 6.50: Monotonicity properties of `Count`

```
/*@
  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 m, n; 0 <= m < n   ==>
      a[m] != v  ==>  RemoveCount(a, m, v) < RemoveCount(a, n, v);
*/
```

Listing 6.51: Monotonicity properties of `RemoveCount`

Listing 6.52 shows another lemma that we formulated in order to completely verify the `retain` properties in Listing 6.49.

```
/*@
  lemma MultisetRetainRestUnchanged{K,L}:
    \forall value_type *a, *b, v, integer m, n;
       0 <= m  ==>  0 <= n                        ==>
       n == RemoveCount{L}(a, m, v)               ==>
       MultisetRetainRest{K,L}(a, m, b, n, v)  ==>
       Unchanged{K,L}(a,m)                        ==>
       MultisetRetainRest{L,L}(a, m, b, n, v);
*/
```

Listing 6.52: The lemma `MultisetRetainRestUnchanged`

6.16. The `random_shuffle` algorithm

The `random_shuffle` algorithm in the C++ Standard Library randomly rearranges the elements of a given range, that is, it randomly picks one of its possible orderings. For our purposes we have modified the generic implementation[56] to that of a range of type `value_type`. The signature now reads:

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

Figure 6.53 illustrates an example run of `random_shuffle`. We used the ACSL notation `\old(a[0..n-1])` and `a[0..n-1]` to denote the contents of the array pointed to by `a` before and after the run, respectively. Note the difference to `\old(a)` and `a`: while the latter two *pointers* are equal, the former two *ranges* needn't be, as shown in Figure 6.53.

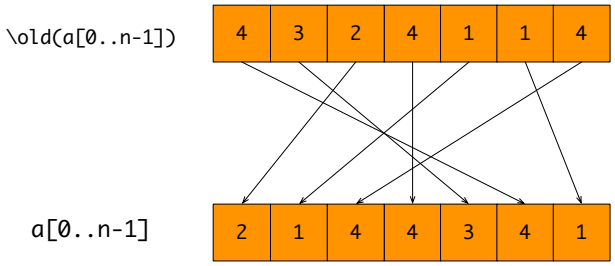


Figure 6.53.: Effects of `random_shuffle`

In Figure 6.53, the values 1, 2, 3, and 4 occurs twice, once, once, and three times, respectively, both before and after the `random_shuffle` run. This expresses that the range has been reordered.

[56] See `http://www.sgi.com/tech/stl/random_shuffle.html`

6.16.1. Formal specification of `random_shuffle`

The ACSL specification of `random_shuffle` is shown in Listing 6.54. The `random_shuffle` algorithm expects that the range `a` is valid for reading and writing. We use the predicate `MultisetUnchanged` defined in Listing 6.5 to express that the contents of `a[0..n-1]` is just permuted, i.e., the number of occurrences of each of its members remains unchanged.

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

  assigns a[0..n-1];

  ensures  MultisetUnchanged{Pre,Here}(a,n);
*/
void random_shuffle(value_type* a, size_type n);
```

Listing 6.54: Formal specification of `random_shuffle`

Note that our specification only states that the resulting range is a reordering of the input range; nothing more and nothing less. Ideally, we would also specify that sequence of reorderings obtained by repeated calls of `random_shuffle` is required to be random. However, ACSL does currently not support the specification of temporal properties related to repeated call results.

More generally speaking, it is not trivial to capture the notion of randomness in a mathematically precise way. As a typical example, one of the papers cited by SGI's description web page of `random_shuffle`, viz. [14, p.6–8], just gives four statistical tests indicating the randomness of the permutations computed with their algorithm. From a theoretical point of view, a sequence of permutations can be called "random" if its Kolmogorov complexity exceeds a certain measure, however, this property is undecidable [15].

6.16.2. Implementation of `random_shuffle`

Listing 6.55 shows an implementation of the `random_shuffle` function. It repeatedly calls the function `swap` from Section 6.4 to *transpose* (randomly) selected elements.

```
void random_shuffle(value_type* a, size_type n)
{
  if (n > 0) {

    /*@
      loop invariant bounds:    1 <= i <= n;
      loop invariant reorder:   MultisetUnchanged{Pre,Here}(a,0,i);
      loop invariant unchanged: Unchanged{Pre,Here}(a,i,n);
      loop assigns   i, a[0..n-1];
      loop variant   n - i;
    */
    for (size_type i = 1; i < n; ++i) {
      const size_type j = random_number(i) + 1;

      //@ ghost Before:
      swap(&a[j], &a[i]);
      //@ assert reorder: MultisetUnchanged{Before,Here}(a, 0, j);
      //@ assert reorder: MultisetUnchanged{Before,Here}(a, j+1, i);
    }
  }
}
```

Listing 6.55: Implementation of `random_shuffle`

The loop invariants `reorder` and `unchanged` of `random_shuffle` are necessary for the verification of the postcondition `reorder`: in the `i`th loop cycle, the subrange `a[0..i-1]` has been reordered, while the remaining subrange `a[i..n-1]` is yet unchanged. We also formulate two auxiliary assertions `reorder`, that use the *ghost label* `Before`, to guide the automatic verification the loop invariant `reorder`.

Our implementation presupposes a random-number generator named `random_number` which is specified in Listing 6.56.

```
/*@
  requires 0 < n;

  assigns \nothing;

  ensures 0 <= \result < n;
*/
size_type random_number(size_type n);
```

Listing 6.56: Formal specification of `random_number`

As in the case of `random_shuffle` itself, we do not formulate specific properties of randomness and only require its result to be in the specified range `0..n-1]`. Moreover, it mustn't modify any memory locations that are visible in this context.[57]

[57] This somewhat vague formulation refers to that fact that a typical (pseudo-) random generator relies on some external state.

7. Numeric algorithms

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

In order to refer to potential arithmetic overflows we introduce the two constants

```
#define VALUE_TYPE_MAX  INT_MAX
#define VALUE_TYPE_MIN  INT_MIN
```

Listing 7.1: Limits of `value_type`

which refer to the numeric limits of `value_type` (see also Section 1.3).

We consider the following algorithms.

- `iota` writes sequentially increasing values into a range (Section 7.1 on Page 110)
- `accumulate` computes the sum of the elements in a range (Section 7.2 on Page 112)
- `inner_product` computes the inner product of two ranges (Section 7.3 on Page 116)
- `partial_sum` computes the sequence of partial sums of a range (Section 7.4 on Page 119)
- `adjacent_difference` computes the differences of adjacent elements in a range (Section 7.5 on Page 123)

The formal specifications of these algorithms raise 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[58] reads:

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

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

7.1.1. Formal specification of `iota`

The specification of `iota` relies on the logic function `Iota` that is defined in Listing 7.2.

```
/*@
  predicate
   Iota(value_type* a, integer n, value_type v) =
     \forall integer i; 0 <= i < n ==> a[i] == v + i;
*/
```

Listing 7.2: Logic function `Iota`

The ACSL specification of `iota` is shown in Listing 7.3. It uses the logic function `Iota` in order to express the postcondition `increment`.

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

  assigns a[0..n-1];

  ensures increment: Iota(a, n, val);
*/
void iota(value_type* a, size_type n, value_type val);
```

Listing 7.3: Formal specification of `iota`

The specification of `iota` refers to `VALUE_TYPE_MAX` which is the maximum value of the underlying integer type (see Listing 7.1). In order to avoid integer overflows the sum `val+n` must not be greater than the constant `VALUE_TYPE_MAX`.

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

7.1.2. Implementation of `iota`

Listing 7.4 shows an implementation of the `iota` function.

```
void iota(value_type* a, size_type n, value_type val)
{
  /*@
    loop invariant bound:     0 <= i <= n;
    loop invariant limit:     val == \at(val, Pre) + i;
    loop invariant increment: Iota(a, i, \at(val, Pre));

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

Listing 7.4: Implementation of `iota`

The loop invariant `increment` 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`. We have to refer here to `\at(val,Pre)` which is the value on entering `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[59] reads:

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

The result of `accumulate` shall equal the value

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

This implies that `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 3.9) we specify `accumulate` by first defining a *logic function* `Accumulate` that formally defines the summation of elements in an array.

```
/*@
   axiomatic AccumulateAxiomatic
   {
      logic value_type 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;
           Unchanged{L1,L2}(a, n) ==>
           Accumulate{L1}(a, n, init) == Accumulate{L2}(a, n, init);
   }
*/
```

Listing 7.5: The logic function `Accumulate`

With this definition the following equation holds

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

Both the `reads` clause and the axiom `AccumulateRead` in Listing 7.5 express that the result of `Accumulate` only depends on the values of `a[0..n-1]`.

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

Listing 7.6 shows an overloaded version of `Accumulate` that uses 0 as default value of `init`. Included in this listing is also a property corresponding to axiom `AccumulateNext` from Listing 7.5, here given as a lemma. We will use this version for the specification of the algorithm `partial_sum` (see Section 7.4).

Thus, for the overloaded version of `Accumulate` we have

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

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

  lemma AccumulateDefault0{L}:
    \forall value_type* a;
      Accumulate(a, 0) == 0;

  lemma AccumulateDefault1{L}:
    \forall value_type* a;
      Accumulate(a, 1) == a[0];

  lemma AccumulateDefaultNext{L}:
    \forall value_type* a, integer n;
      n >= 0 ==> Accumulate(a, n+1) == Accumulate(a, n) + a[n];

  lemma AccumulateDefaultRead{L1,L2}:
    \forall value_type *a, integer n;
      Unchanged{L1,L2}(a, n) ==>
      Accumulate{L1}(a, n) == Accumulate{L2}(a, n);

*/
```

Listing 7.6: An overloaded version of `Accumulate`

7.2.2. Preventing numeric overflows for `accumulate`

Before we present our formal specification of `accumulate` we introduce in Listing 7.7 a predicate `AccumulateBounds` that we will subsequently use in order to compactly express requirements that exclude numeric overflows while accumulating value.

```
/*@
  predicate
    AccumulateBounds{L}(value_type* a, integer n, value_type init) =
      \forall integer i; 0 <= i <= n ==>
        VALUE_TYPE_MIN <= Accumulate(a, i, init) <= VALUE_TYPE_MAX;

  predicate
    AccumulateBounds{L}(value_type* a, integer n) =
      AccumulateBounds{L}(a, n, (value_type) 0);
*/
```

Listing 7.7: The overloaded predicate `AccumulateBounds`

Predicate `AccumulateBounds` expresses that for $0 \leq i < n$ the *partial sums*

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

do not overflow. If one of them did, one couldn't guarantee that the result of `accumulate` equals the mathematical description of `Accumulate`.

Note that we also provide a second (overloaded) version of `AccumulateBounds` which uses a default value 0 for `init`.

7.2.3. Formal specification of `accumulate`

Using the logic function `Accumulate` and the predicate `AccumulateBounds`, the ACSL specification of `accumulate` is then as simple as shown in Listing 7.8.

```
/*@
  requires valid:  \valid_read(a + (0..n-1));
  requires bounds: AccumulateBounds(a, n, init);

  assigns  \nothing;

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

Listing 7.8: Formal specification of `accumulate`

7.2.4. Implementation of `accumulate`

Listing 7.9 shows an implementation of the `accumulate` function with corresponding loop annotations.

```
value_type
accumulate(const value_type* a, size_type n, value_type init)
{
  /*@
    loop invariant index:    0 <= i <= n;
    loop invariant partial:  init == Accumulate(a, i, \at(init,Pre));
    loop assigns i, init;
    loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i) {
    init = init + a[i];
  }

  return init;
}
```

Listing 7.9: Implementation of `accumulate`

Note that loop invariant `partial` claims that in the i-th iteration step `result` equals the accumulated value of Equation (7.3). This depends on the property `bounds` in Listing 7.8 which expresses that there is no numeric overflow when updating the variable `init`.

7.3. The `inner_product` algorithm

The `inner_product` algorithm in the C++ standard library computes the *inner product*[60] of two ranges. Our version of the original signature[61] 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 empty ranges.

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.

```
/*@
   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;
         Unchanged{L1,L2}(a, n) && Unchanged{L1,L2}(b, n) ==>
           InnerProduct{L1}(a, b, n, init) ==
           InnerProduct{L2}(a, b, n, init);
   }
*/
```

Listing 7.10: The logic function `InnerProduct`

Both Axiom `InnerProductRead` and the `reads` clause serve the same purpose in that they express that the result of the `InnerProduct` only depends on the values of `a[0..n-1]` and `b[0..n-1]`.

[60] Also referred to as *dot product*, see `http://en.wikipedia.org/wiki/Dot_product`
[61] See `http://www.sgi.com/tech/stl/inner_product.html`

7.3.2. Preventing numeric overflows for `inner_product`

Before we present our formal specification of `inner_product` we introduce in Listing 7.11 two predicates that we will use subsequently in order to compactly express requirements that exclude numeric overflows while computing the inner product.

```
/*@
  predicate
    ProductBounds(value_type* a, value_type* b, integer n) =
      \forall integer i; 0 <= i < n ==>
        VALUE_TYPE_MIN <= a[i] * b[i] <= VALUE_TYPE_MAX;

   predicate
     InnerProductBounds(value_type* a, value_type* b, integer n,
                        value_type init) =
       \forall integer i; 0 <= i <= n ==>
          VALUE_TYPE_MIN <= InnerProduct(a, b, i, init) <= VALUE_TYPE_MAX;
*/
```

Listing 7.11: The predicates `ProductBounds` and `InnerProductBounds`

Predicate `ProductBounds` expresses that for $0 \leq i < n$ the products

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

do not overflow. Predicate `InnerProductBounds`, on the other hand, states that for $0 \leq i < n$ the *partial sums*

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

do not overflow.

Otherwise, one cannot guarantee that the result of `inner_product` equals the mathematical description of `InnerProduct`.

7.3.3. Formal specification of `inner_product`

Using the logic function `InnerProduct`, we specify `inner_product` as shown in Listing 7.12. Note that we needn't require that `a` and `b` are separated.

```
/*@
  requires valid:   \valid_read(a + (0..n-1));
  requires valid:   \valid_read(b + (0..n-1));
  requires bounds:  ProductBounds(a, b, n);
  requires bounds:  InnerProductBounds(a, b, n, init);

  assigns \nothing;

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

Listing 7.12: Formal specification of `inner_product`

7.3.4. Implementation of `inner_product`

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

```
value_type
inner_product(const value_type* a, const value_type* b, size_type n,
              value_type init)
{
  /*@
     loop invariant index: 0 <= i <= n;
     loop invariant inner: init == InnerProduct(a, b, i, \at(init,Pre));
     loop assigns i, init;
     loop variant n-i;
  */
  for (size_type i = 0; i < n; ++i) {
    init = init + a[i] * b[i];
  }

  return init;
}
```

Listing 7.13: Implementation of `inner_product`

Note that the loop invariant `inner` claims that in the i-th iteration step the current value of `init` equals the accumulated value of Equation (7.5). This depends of course on the properties `bounds` in Listing 7.12, which express that there is no arithmetic overflow when computing the updates of the variable `init`.

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[62] 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

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

More concisely, for $0 \leq i < n$ holds

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

7.4.1. The predicate `PartialSum`

Equations (7.6) and (7.2) suggest that we define the ACSL predicate `PartialSum` in Listing 7.14 by using the logic function `Accumulate` from Listing 7.6. Listing 7.14.

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

Listing 7.14: The predicate `PartialSum`

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

7.4.2. Formal specification of `partial_sum`

Using the predicates `PartialSum` and `AccumulateBounds`, we specify `partial_sum` as shown in Listing 7.15.

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

  assigns b[0..n-1];

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

Listing 7.15: Formal specification of `partial_sum`

Our specification requires that the arrays `a[0..n-1]` and `b[0..n-1]` are separated, that is, they do not overlap. Note that is a stricter requirement than in the case of the original C++ version of `partial_sum`, which allows that `a` equals `b`, thus allowing the computation of partial sums *in place*.[63]

[63] See Note [1] at `http://www.sgi.com/tech/stl/partial_sum.html`

7.4.3. Implementation of `partial_sum`

Listing 7.16 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 bound:      1 <= i <= n;
       loop invariant unchanged:  Unchanged{Here,Pre}(a, n);
       loop invariant accumulate: b[i-1] == Accumulate(a, i);
       loop invariant partialsum: PartialSum(a, i, b);
       loop assigns i, b[1..n-1];
       loop variant n - i;
    */
    for (size_type i = 1u; i < n; ++i) {
      //@ ghost Enter:
      b[i] = b[i - 1u] + a[i];
      //@ assert unchanged:  a[i] == \at(a[i],Enter);
      //@ assert unchanged:  Unchanged{Enter,Here}(a, i);
      //@ assert unchanged:  Unchanged{Enter,Here}(b, i);
    }
  }

  return n;
}
```

Listing 7.16: Implementation of `partial_sum`

In order to facilitate the automatic verification of `partial_sum`, we had to add the assertions `unchanged` and provide the lemmas of Listing 7.17.

7.4.4. Additional lemmas

The lemmas shown in Listing 7.17 are needed for the verification of `partial_sum` and the algorithms in Sections 7.6 and 7.7.

```
/*@
  lemma PartialSumSection{K}:
    \forall value_type *a, *b, integer m, n;
      0 <= m <= n            ==>
      PartialSum{K}(a, n, b) ==>
      PartialSum{K}(a, m, b);

  lemma PartialSumUnchanged{K,L}:
    \forall value_type *a, *b, integer n;
      0 <= n ==>
      PartialSum{K}(a, n, b) ==>
      Unchanged{K, L}(a, n)  ==>
      Unchanged{K, L}(b, n)  ==>
      PartialSum{L}(a, n, b);

  lemma PartialSumStep{L}:
    \forall value_type *a, *b, integer n;
      1 <= n                     ==>
      PartialSum(a, n, b)        ==>
      b[n] == Accumulate(a, n+1) ==>
      PartialSum(a, n+1, b);

  lemma PartialSumStep2{K,L}:
    \forall value_type *a, *b, integer n;
      1 <= n                              ==>
      PartialSum{K}(a, n, b)              ==>
      Unchanged{K,L}(a, n+1)              ==>
      Unchanged{K,L}(b, n)                ==>
      \at(b[n] == Accumulate(a, n+1),L) ==>
      PartialSum{L}(a, n+1, b);
*/
```

Listing 7.17: The lemma `PartialSumStep`

7.5. The `adjacent_difference` algorithm

The `adjacent_difference` algorithm in the C++ standard library computes the differences of adjacent elements in a range. Our version of the original signature[64] 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

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

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

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

which means that applying `partial_sum` on the output of `adjacent_difference` produces the original input of `adjacent_difference`.

Conversely, if `a[0..n-1]` and `b[0..n-1]` are the input and output of `partial_sum`, then we have

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

from which we can conclude

$$\begin{aligned} \mathsf{a}[0] &= \mathsf{b}[0] \\ \mathsf{a}[1] &= \mathsf{b}[1] - \mathsf{b}[0] \\ &\vdots \\ \mathsf{a}[n-1] &= \mathsf{b}[n-1] - \mathsf{b}[n-2] \end{aligned} \tag{7.8}$$

We will verify these claims in Sections 7.6 and 7.7.

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

7.5.1. The predicate `AdjacentDifference`

We define the predicate `AdjacentDifference` in Listing 7.19 by first introducing the logic function `Difference` (Listing 7.18).

```
/*@
   axiomatic DifferenceAxiomatic
   {
      logic value_type
        Difference{L}(value_type* a, integer n) reads a[0..n];

      axiom DifferenceEmptyOrSingle:
        \forall value_type *a, integer n;
           n <= 0 ==> Difference(a, n) == a[0];

      axiom DifferenceNext:
        \forall value_type *a, integer n;
           n >= 1 ==> Difference(a, n) == a[n] - a[n-1];

      axiom DifferenceRead{K,L}:
        \forall value_type *a, integer n;
           Unchanged{K,L}(a, 1+n) ==>
           Difference{K}(a, n) == Difference{L}(a, n);
   }
*/
```

Listing 7.18: The logic function `Difference`

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

Listing 7.19: The predicate `AdjacentDifference`

7.5.2. Formal specification of `adjacent_difference`

We introduce here the predicate `AdjacentDifferenceBounds` (Listing 7.20) that captures conditions that prevent numeric overflows while computing difference of the form `a[i] - a[i-1]`.

```
/*@
  predicate
    AdjacentDifferenceBounds(value_type* a, integer n) =
      \forall integer i; 1 <= i < n ==>
        VALUE_TYPE_MIN <= Difference(a, i) <= VALUE_TYPE_MAX;
*/
```

Listing 7.20: The predicate `AdjacentDifferenceBounds`

Using the predicates `AdjacentDifference` and `AdjacentDifferenceBounds` we can provide a concise formal specification of `adjacent_difference` (Listing 7.21). 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.

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

   assigns b[0..n-1];

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

Listing 7.21: Formal specification of `adjacent_difference`

7.5.3. Implementation of `adjacent_difference`

Listing 7.22 shows an implementation of `adjacent_difference` with corresponding loop annotations.

In order to achieve the verification of the loop invariant `difference` we added

- the assertions `bound` and `difference`
- the lemmas `AdjacentDifferenceStep` and `AdjacentDifferenceSection` from Listing 7.23
- a statement contract with the two postconditions labeled as `step`

```
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:  Unchanged{Here,Pre}(a, n);
       loop invariant difference: AdjacentDifference(a, i, b);
       loop assigns i, b[1..n-1];
       loop variant n - i;
    */
    for (size_type i = 1u; i < n; ++i) {
      //@ assert bound: VALUE_TYPE_MIN <= Difference(a, i) <= VALUE_TYPE_MAX;
      /*@
        assigns b[i];
        ensures step: Unchanged{Old,Here}(b, i);
        ensures step: b[i] == Difference(a, i);
      */
      b[i] = a[i] - a[i - 1u];
      //@ assert difference: AdjacentDifference{Here}(a, i+1, b);
    }
  }

  return n;
}
```

Listing 7.22: Implementation of `adjacent_difference`

7.5.4. Additional Lemmas

The lemmas shown in Listing 7.23 are also needed for the verification of the algorithm in Section 7.7.

```
/*@
  lemma AdjacentDifferenceStep{K,L}:
    \forall value_type *a, *b, integer n;
      AdjacentDifference{K}(a, n, b)      ==>
      Unchanged{K,L}(b, n)                ==>
      Unchanged{K,L}(a, n+1)              ==>
      \at(b[n],L) == Difference{L}(a, n)  ==>
      AdjacentDifference{L}(a, 1+n, b);

  lemma AdjacentDifferenceSection{K}:
    \forall value_type *a, *b, integer m, n;
      0 <= m <= n                      ==>
      AdjacentDifference{K}(a, n, b)   ==>
      AdjacentDifference{K}(a, m, b);
*/
```

Listing 7.23: The lemma `AdjacentDifferenceStep`

7.6. Inverting `partial_sum` with `adjacent_difference`

In Section 7.5 we had informally argued that `partial_sum` and `adjacent_difference` are inverse to each other (see Equations (7.7) and (7.8)). In the current section, we are going to verify the second of these claims with the help of Frama-C, viz. that applying `adjacent_difference` to the output of `partial_sum` produces the original array. In Section 7.7, we will verify the converse first claim.

Listing 7.24 expresses the property from Equation (7.8) as lemma, on the ACSL logical level. This lemma is verified by Frama-C with the help of automatic theorem provers.

```
/*@
  lemma PartialSumInverse{K,L}:
    \forall value_type *a, *b, integer n;
      0 <= n                              ==>
      PartialSum{K}(a, n, b)              ==>
      Unchanged{K,L}(b, n)                ==>
      AdjacentDifference{L}(b, n, a)      ==>
      Unchanged{K,L}(a, n);
*/
```

Listing 7.24: The lemma `PartialSumInverse`

Since the lemma does not deal with arithmetic overflows or potential aliasing of data, we give a corresponding auxiliary C function which takes these issues into account.

Function `partial_sum_inverse`, shown in Listing 7.25, calls first `partial_sum` and then `adjacent_difference`. The contract of this function formulates preconditions that shall guarantee that during the computation neither arithmetic overflows (property `bound`) nor unintended aliasing of arrays (property `separated`) occur. Under these precondition, Frama-C automatically verifies that the final `adjacent_difference` call just restores the original contents of `a` used for the initial `partial_sum` call.

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

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

  ensures unchanged:   Unchanged{Here,Pre}(a, n);
*/
void partial_sum_inverse(value_type* a, size_type n, value_type* b)
{
  partial_sum(a, n, b);
  adjacent_difference(b, n, a);
}
```

Listing 7.25: `partial_sum` and then `adjacent_difference`

7.7. Inverting `adjacent_difference` with `partial_sum`

In this section, we prove the converse property, viz. that applying `adjacent_difference`, and thereafter `partial_sum`, restores the original data array. Listing 7.26 expresses this property as a lemma on the level of ACSL predicates. It had to be proven interactively with Coq, by induction on `n`.

```
/*@
  lemma AdjacentDifferenceInverse{K,L}:
    \forall  value_type *a, *b, integer n;
      0 <= n                              ==>
      AdjacentDifference{K}(a, n, b)   ==>
      Unchanged{K,L}(b, n)             ==>
      PartialSum{L}(b, n, a)           ==>
      Unchanged{K,L}(a, n);
*/
```

Listing 7.26: The lemma `AdjacentDifferenceInverse`

As in the case discussed in Section 7.6, we give a corresponding C function in order to account for possible arithmetic overflows and potential aliasing of data. Function `adjacent_difference_inverse`, shown in Listing 7.27, calls first `adjacent_difference` and then `partial_sum`. The contract of this function formulates preconditions that shall guarantee that during the computation neither arithmetic overflows (property `bound`) nor unintended aliasing of arrays (property `separated`) occur.

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

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

  ensures unchanged:   Unchanged{Here,Pre}(a, n);
*/
void adjacent_difference_inverse(value_type* a, size_type n, value_type* b)
{
  adjacent_difference(a, n, b);
  partial_sum(b, n, a);
}
```

Listing 7.27: `adjacent_difference` and then `partial_sum`

In order to improve the automatic verification rate of the function `adjacent_difference_inverse` we also use lemma `UnchangedTransitive` from Listing 6.4. Both lemmas itself and (with its additional help) the contract of `adjacent_difference_inverse` are proven by Frama-C without further manual intervention. This finishes the formal proof of our inversity claims from Section 7.5.

8. Heap Operations

Heap algorithms were already part of *ACSL by Example* from 2010–2012. In this chapter we re-introduce them and discuss—based on the bachelor thesis of one of the authors—the verification efforts in some detail [16].

The C++ standard[65] introduces the concept of a *heap* as follows:

1. A *heap* is a particular organization of elements in a range between two random access iterators `[a,b)`. Its two key properties are:
   a) There is no element greater than `*a` in the range and
   b) `*a` may be removed by `pop_heap()`, or a new element added by `push_heap()`, in $O(\log(N))$ time.
2. These properties make heaps useful as priority queues.
3. `make_heap()` converts a range into a heap and `sort_heap()` turns a heap into a sorted sequence.

Figure 8.1 gives an overview on the five heap algorithms by means of an example. Algorithms that are in a caller-callee relation have the same color.

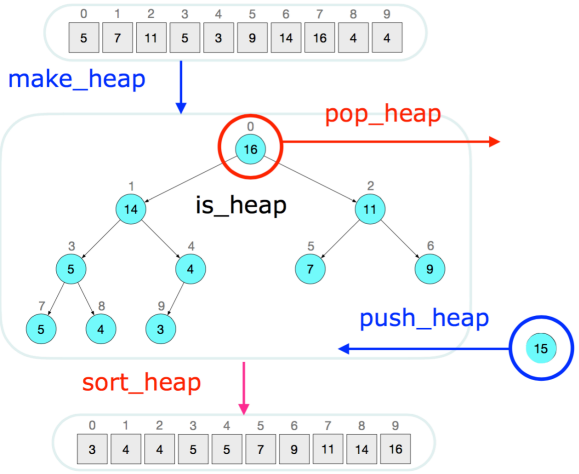


Figure 8.1.: Overview on heap algorithms

- `is_heap` from Section 8.3 allows to test at run time whether a given array is arranged as a heap
- `push_heap` from Section 8.4 *adds* an element to a given heap in such a way that resulting array is again a heap
- `make_heap` from Section 8.5 turns a given array into a heap.
- `sort_heap` from Section 8.6 transforms a given heap into a sorted range.

[65] See `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`

8.1. Introduction

This description of heaps at the beginning of this chapter is of course fairly vague. It outlines only the most important properties of various operations but does not clearly state what specific and verifiable properties a range must satisfy such that it may be called a heap.

A more detailed description can be found in the Apache C++ Standard Library User's Guide:[66]

> A heap is a binary tree in which every node is larger than the values associated with either child. A heap and a binary tree, for that matter, can be very efficiently stored in a vector, by placing the children of node i at positions $2i + 1$ and $2i + 2$.

We have, in other words, the following basic relations between indices of a heap:

$$\text{left child for index } i \qquad \text{child}_\text{l} : i \mapsto 2i + 1 \tag{8.1}$$

$$\text{right child for index } i \qquad \text{child}_\text{r} : i \mapsto 2i + 2 \tag{8.2}$$

and

$$\text{parent index for index } i \qquad \text{parent} : i \mapsto \frac{i - 1}{2} \tag{8.3}$$

These function are related through the following two equations that hold for all integers i. Note that in ACSL integer division rounds towards zero (cf. [9, §2.2.4]).

$$\text{parent}(\text{child}_\text{l}(i)) = i \tag{8.4}$$

$$\text{parent}(\text{child}_\text{r}(i)) = i \tag{8.5}$$

At various places in this chapter we will consider the following multiset of integers.

$$X = \{2, 3, 3, 3, 6, 7, 8, 8, 9, 11, 13, 14\} \tag{8.6}$$

Figure 8.2 shows how the multiset X of Equation (8.6) is represented as a tree.

[66] See `http://stdcxx.apache.org/doc/stdlibug/14-7.html`

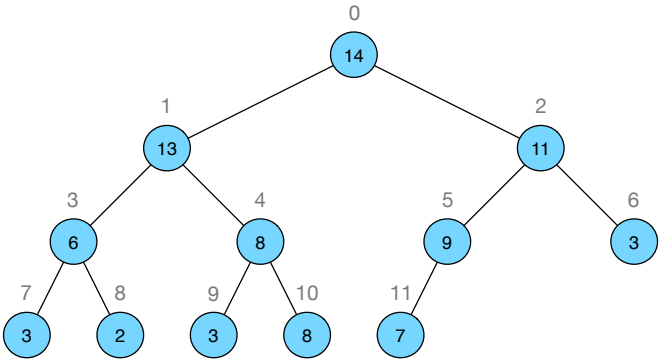


Figure 8.2.: Tree representation of the multiset X

The numbers outside the nodes in Figure 8.2 are the indices at which the respective node value is stored in the underlying array of a heap (cf. Figure 8.3).

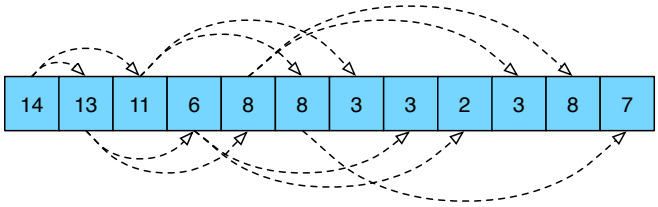


Figure 8.3.: Underlying array of a heap

The ACSL logic functions and predicate that formalize the basic properties of heaps are introduced in Section 8.2. The various heap algorithms are discussed in detail in the following sections.

8.2. Logic definition of heaps

Listing 8.4 shows three logic functions `HeapLeft`, `HeapRight` and `HeapParent` that correspond to the definitions (8.1), (8.2) and (8.3), respectively.

Listing 8.4 also contains some ACSL lemma that state that the `HeapParent` function that correspond the equations (8.4) and (8.5). The lemmas express that the function `HeapParent` is the *left inverse* to the `HeapLeft` and `HeapRight` functions.[67] We also have the lemma `HeapBounds` that is facilitated in the proof of lemma `HeapMaximum` (Listing 8.6) by expressing a simple property of integer division.

[67] See Section *Left and right inverses* at `http://en.wikipedia.org/wiki/Inverse_function`

```
/*@
  logic integer HeapLeft(integer i) = 2*i + 1;

  logic integer HeapRight(integer i) = 2*i + 2;

  logic integer HeapParent(integer i) = (i - 1) / 2;


  lemma HeapParentOfLeft:
    \forall integer i; 0 <= i ==> HeapParent(HeapLeft(i)) == i;

  lemma HeapParentOfRight:
    \forall integer i; 0 <= i ==> HeapParent(HeapRight(i)) == i;

  lemma HeapBounds:
    \forall integer a; 0 <= a  ==>  0 <= a/2 <= a;
*/
```

Listing 8.4: Logic functions for heap definition

On top of these basic definitions we introduce in Listing 8.5 the predicate `IsHeap`.

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

Listing 8.5: The predicate `IsHeap`

The root of a heap, that is the element at index 0, is always the largest element of the heap. Lemma `HeapMaximum` in Listing 8.6 expresses this property using the `MaxElement` predicate from Listing 4.6.

```
/*@
  lemma HeapMaximum{L} :
    \forall value_type* a, integer n;
      1 <= n ==> IsHeap(a, n) ==> MaxElement(a, n, 0);
*/
```

Listing 8.6: The lemma `HeapMaximum`

8.3. The `is_heap` algorithm

The `is_heap` algorithm of the STL in the Library works on generic sequences. For our purposes we have modified the generic implementation[68] to that of an array of type `value_type`. The signature now reads:

```
bool is_heap(const value_type* a, int n);
```

The algorithm `is_heap` checks whether a given array satisfies the heap properties we have semi-formally described in the beginning of this chapter. In particular, `is_heap` will return **`true`** called with the array argument from Figure 8.3.

[68] See `http://www.sgi.com/tech/stl/is_heap.html`.

8.3.1. Formal specification of `is_heap`

The ACSL specification of `is_heap` is shown in Listing 8.7. The function returns **`true`** if and only if its arguments satisfy the predicate `IsHeap` introduced in Section 8.2.

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

   assigns \nothing;

   ensures heap:  \result <==> IsHeap(a, n);
*/
bool is_heap(const value_type* a, size_type n);
```

Listing 8.7: Function Contract of `is_heap`

8.3.2. Implementation of `is_heap`

Listing 8.8 shows one way to implement the function `is_heap`. The algorithms starts at the index 1, which is the smallest index, where a child node of the heap might reside. The algorithms checks for each (child) index whether the value at the corresponding parent index is greater than or equal to the value at the child index.

```
bool is_heap(const value_type* a, size_type n)
{
  size_type parent = 0;

  /*@
    loop invariant bound:  0 <= parent < child <= n+1;
    loop invariant parent: parent == HeapParent(child);
    loop invariant heap:   IsHeap(a, child);

    loop assigns child, parent;
    loop variant n - child;
  */
  for (size_type child = 1u; child < n; ++child) {
    if (a[parent] < a[child]) {
      return false;
    }

    if ((child % 2u) == 0) {
      parent++;
    }
  }

  return true;
}
```

Listing 8.8: Implementation of `is_heap`

8.4. The `push_heap` algorithm

Whereas in the STL `push_heap` works on a range of random access iterators[69], our version operates on an array of `value_type`. We therefore use the following signature for `push_heap`

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

The `push_heap` algorithm expects that the first $n-1$ elements of the array form a heap. The last element of the array (whose index is $n-1$) is added (i.e. *pushed*) on the heap. Thus, in the end all n elements of the array form a heap.

8.4.1. Formal Specification of `push_heap`

Listing 8.9 shows our ACSL specification of `push_heap`. Note that the post condition `reorder` states that `push_heap` is not allowed to change the number of occurrences of an array element. Without this post condition, an implementation that assigns `0` to each array element would satisfy the post condition `heap`—surely not what the user of the algorithm has in mind.

```
1  /*@
2     requires nonempty: n > 0;
3     requires valid:    \valid(a + (0..n-1));
4     requires heap:     IsHeap(a, n-1);
5
6     assigns a[0..n-1];
7
8     ensures heap:      IsHeap(a, n);
9     ensures reorder:   MultisetUnchanged{Old,Here}(a, n);
10 */
11 void push_heap(value_type* a, size_type n);
```

Listing 8.9: Formal specification of `push_heap`

Pushing an element on a heap usually *rearranges* several elements of the array (cf. Figures 8.10 and 8.11). We therefore must be able express that `push_heap` only *reorders* the elements of the array. We re-use the predicate `MultisetUnchanged`, defined in Listing 6.5, to formally describe this property.

[69] See `http://www.sgi.com/tech/stl/push_heap.html`

8.4.2. Implementation of `push_heap`

The following two figures illustrate how `push_heap` affects an array, which is shown as a tree with blue and grey nodes, representing heap and non-heap nodes, respectively. Figure 8.10 shows the heap from Figure 8.2 together with the additional element `12` that is to be on the heap. To be quite clear about it: the new element `12` is the last element of the array and not yet part of the heap.

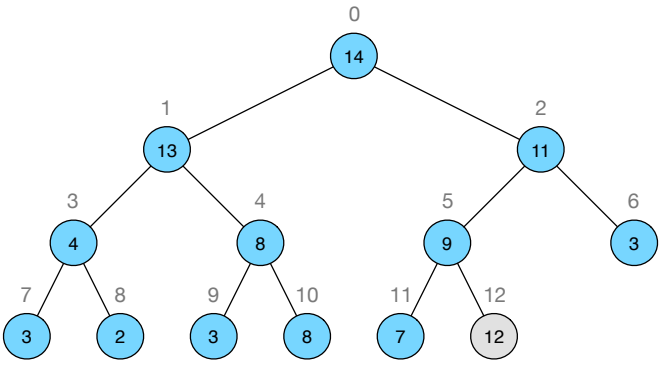


Figure 8.10.: Heap before the call of `push_heap`

Figure 8.11 shows the array after the call of `push_heap`. We can see that now all nodes are colored in blue, i.e., they are part of the heap. The dashed nodes changed their contents during the function call. The pushed element `12` is now at its correct position in the heap.

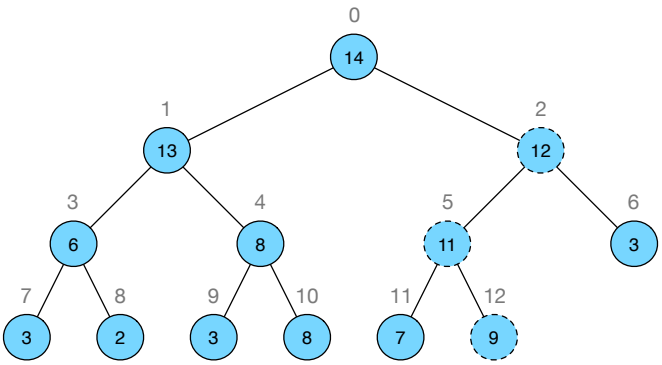


Figure 8.11.: Heap after the call of `push_heap`

8.4.2.1. Challenges during the verification

In order to properly describe different stages of `push_heap` and to accommodate the sheer size of our implementation we split the source code into three separate parts, to which we refer, respectively, as

- *prologue* (see Section 8.4.2.2)
- *main act* (see Section 8.4.2.3)
- *epilogue* (see Section 8.4.2.4)

We will illustrate the changes to the array after each stage by figures of the array in tree form, based on the `push_heap` example from Figure 8.10.

Verifying push_heap is no trivial undertaking and we will proceed, roughly speaking, as follows:

> Whereas we can establish the desired heap property of Listing 8.9 already in the prologue, the reorder property only holds at the function boundaries but is violated while push_heap manipulates the array. To be more precise: We loose the reorder property in the prologue and formally capture and maintain a slightly more general property in the main act. From this we will recover the reorder property in the epilogue.

8.4.2.2. Prologue

Listing 8.12 shows the implementation of the prologue. It consists of what is basically one run of the loop in the main act. However, unrolling the first loop run allows us to isolate the critical iteration, after which the MultisetUnchanged predicate is no longer true for the array, which is always the first loop run. This will ease the verification in the main act. The assertion heap states that the predicate IsHeap is true after the prologue.

```
void push_heap(value_type* a, size_type n)
{
  // start of prologue

  if (n == 1) {
    return;
  }

  const value_type val = a[n - 1];
  size_type hole = n - 1;

  size_type parent = (hole - 1) / 2;

  if (val > a[parent]) {
    /*@
       requires val:      a[hole] == val;
       assigns  a[hole];
       ensures  newhole:  Unchanged{Old, Here}(a, 0, hole);
       ensures  newhole:  Unchanged{Old, Here}(a, hole+1, n);
       ensures  val:      a[hole] == a[parent];
    */
    a[hole] = a[parent];
    hole = parent;
  }
  //@ assert heap:  IsHeap(a, n);

  if (hole == n - 1) {
    return;
  }

  // end of prologue
```

Listing 8.12: Prologue of push_heap implementation

Figure 8.13 shows the tree after the prologue. We can see that now all nodes are blue, meaning that the heap grew to the desired length. If we compare the tree to the tree in Figure 8.10 we notice that the element in the node with the dashed outlining changed it's value. The number of occurrences of `9` increased by one during the prologue, while the number of occurrences of `12` decreased by one. The number of occurrences for all other elements is maintained. We store the element `12` in the variable `val` so we can write it back into the array later.

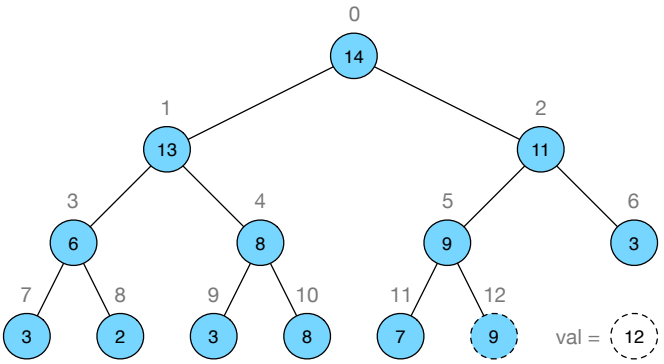


Figure 8.13.: Heap after the prologue of `push_heap`

The increased number of occurrences of `9` and the decreased number of elements `12` means that the `MultisetUnchanged` predicate is no longer true for this array. In order to formally express this deviation from `MultisetUnchanged` we introduce three auxiliary predicates:

The predicate `MultisetUnchangedExcept` (Listing 8.14) holds if the number of occurrences for all elements, except the two given ones, remains unchanged.

```
/*@
  predicate
    MultisetUnchangedExcept{K,L}(value_type* a, integer n,
                                 value_type  v, value_type w) =
      \forall value_type x;
        x != v  ==>  x != w  ==>  Count{L}(a, n, x) == Count{K}(a, n, x);
*/
```

Listing 8.14: The predicate `MultisetUnchangedExcept`

The predicate `MultisetAdd` in Listing 8.15 expresses that the number of occurrences of a specific element in a certain range has increased by one.

```
/*@
  predicate
    MultisetAdd{K,L}(value_type* a, integer n, value_type val) =
      Count{L}(a, n, val) == Count{K}(a, n, val) + 1;
*/
```

Listing 8.15: The predicate `MultisetAdd`

The predicate `MultisetMinus` in Listing 8.16, on the other hand, expresses that the number of occurrences of a specific element in a certain range has decreased by one.

```
/*@
  predicate
    MultisetMinus{K,L}(value_type* a, integer n, value_type val) =
      Count{L}(a, n, val) == Count{K}(a, n, val) - 1;
*/
```

Listing 8.16: The predicate `MultisetMinus`

Thus, at the end of the prologue, instead of the assertion

```
assert MultisetUnchanged{Pre,Here}(a, n);
```

the following three assertion hold

```
assert MultisetAdd{Pre,Here}(a, n, a[hole]);
assert MultisetMinus{Pre,Here}(a, n, val);
assert MultisetUnchangedExcept{Pre,Here}(a, n, a[hole], val);
```

However, instead of putting these *assertions* in the prologue, we express them as *loop invariants* in the so-called main act of `push_heap` (cf. Listing 8.17).

8.4.2.3. Main part

The goal of the main act is to locate the array index to which the new element can be assigned. Listing 8.17 shows the implementation of the main act.

The loop invariant `heap` ensures that the predicate `IsHeap` is true for the array throughout the main act. The `reorder` invariants in the loop specification contain the predicates shown at the end of the prologue description. It is important to understand the use of the variable `hole` in these loop invariants. Before each loop run `hole` is the index of the node whose value was assigned to one of it's children in the prior loop run, or the prologue for the first loop run. Therefore the variable is used as a parameter for the loop invariants with the predicates `MultisetAdd` and `MultisetUnchangedExcept`. However, verifying these loop invariants is not straightforward. Not only the count of the element at the current `hole` index but also the count of the element that was at the `hole` index during the last loop run has to be considered. Both in relation to their counts at the function start. To enable an automated verification various annotations to the loop body and further loop invariants had to be added.

```
// start of main act

/*@
   loop invariant bound:    0 <= hole < n-1;
   loop invariant heap:     IsHeap(a, n);
   loop invariant less:     a[hole] < val;
   loop invariant reorder:  MultisetMinus{Pre,Here}(a, n, val);
   loop invariant reorder:  MultisetAdd{Pre,Here}(a, n, a[hole]);
   loop invariant reorder:  MultisetUnchangedExcept{Pre,Here}(a, n, val, a[hole]);
   loop assigns   hole, a[0..n-1];
   loop variant   hole;
*/
while (hole > 0) {
  const size_type par = (hole - 1) / 2;

  /*@
    requires heap:        IsHeap(a, n);
    assigns  hole, a[hole];
    ensures  heap:        IsHeap(a, n);
    ensures  reorder:     MultisetMinus{Pre,Here}(a, n, val);
    ensures  reorder:     MultisetAdd{Pre,Here}(a, n, a[hole]);
    ensures  reorder:     MultisetUnchangedExcept{Pre,Here}(a, n, val, a[par]);
    ensures  less:        a[hole] < val;
    ensures  decreasing:  hole < \old(hole);
  */
  if (a[par] < val) {
    /*@
       requires heap:       IsHeap(a, n);
       requires heap:       par == HeapParent(hole);
       requires heap:       HeapParent(hole) < hole < n-1;
       assigns  a[hole];
       ensures  heap:       a[hole] == a[HeapParent(hole)];
       ensures  unchanged:  Unchanged{Old,Here}(a, 0, hole);
       ensures  unchanged:  Unchanged{Old,Here}(a, hole+1, n);
       ensures  heap:       IsHeap(a, n);
    */
    a[hole] = a[par];
    /*@
       assigns hole;
       ensures bound:    hole < \old(hole);
       ensures less:     a[hole] < val;
       ensures reorder:  MultisetUnchangedExcept{Pre,Here}(a, n, val, a[hole]);
    */
    hole = par;
    //@ assert reorder:  MultisetUnchangedExcept{Pre,Here}(a, n, val, a[hole]);
  } else {
    break;
  }
  //@ assert heap:  IsHeap(a, n);
}

// end of main act
```

Listing 8.17: Main part of `push_heap` implementation

Figure 8.18 shows the array after the main act. The contents of the dashed nodes have been overwritten with the values of their parents until `hole` reached a node, to which `val` can be assigned, whilst maintaining the `heap` property.

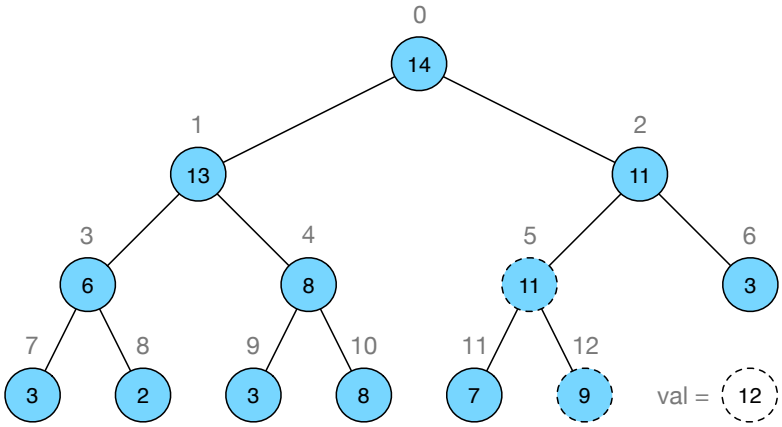


Figure 8.18.: Heap after the main act of `push_heap`

In this example the loop stops when `hole` is `2`. Only the value of the node with the index `2` had to be assigned to the node with the index `5`. At this point the new element `12` can be assigned to the node with the index `2` and the `heap` property stays intact. This assignment takes place in the epilogue.

8.4.2.4. Epilogue

The last part of the implementation is the epilogue. Listing 8.19 shows the implementation of this part. It consists of only one assignment but is elemental to the implementation nevertheless. It writes the new element back into the array. The statement contract for this operation describes the change of the value at `a[hole]`. The predicate `Unchanged`, described in Section 6.1, is used to help the automatic provers with the `reorder` property.

```
  // start of epilogue

  /*@
     requires val:  a[hole] != val;
     assigns  a[hole];
     ensures  val:  a[hole] == val;
     ensures  val:  Unchanged{Old,Here}(a, 0, hole);
     ensures  val:  Unchanged{Old,Here}(a, hole+1, n);
  */
  a[hole] = val;
  //@ assert heap:     IsHeap(a, n);
  //@ assert reorder:  MultisetUnchanged{Pre,Here}(a, n);

  // end of epilogue
}
```

Listing 8.19: Epilogue of `push_heap` implementation

Figure 8.20 shows the array after the final step of the function. All nodes are colored blue and part of the heap, thus the `heap` property can be inferred.

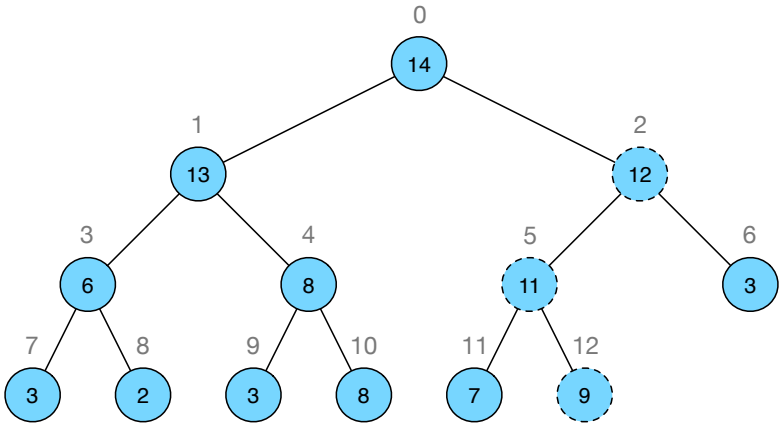


Figure 8.20.: Heap after the epilogue of `push_heap`

Concerning the `reorder` property, the main act finished with an increased count of nodes with the value `11` and a decreased count of nodes with the value `12`. The heap in the figure shows that `12` has been assigned to the node with the index `2`, which previously contained the value `12`. Hence the `reorder` property is reestablished and the function can return.

8.5. The `make_heap` algorithm

Whereas in the STL `make_heap` works on a pair of generic random access iterators,[70] our version operators on a range of `value_type`. Thus the signature of `make_heap` reads

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

The function `make_heap` rearranges the elements of the given array `a[0..n-1]` such that they form a heap. Figure 8.21 shows an array of the length 12 with elements in no particular order in them. Most notably the elements of the array do not form a heap, as indicated by the grey colouring.

8	8	7	3	14	11	3	6	2	3	13	9

Figure 8.21.: Array before the call of `make_heap`

Executing the `make_heap` algorithm on this array results in the array in Figure 8.3. The elements have been rearranged to ensure the right order between each element and its child elements.

8.5.1. Formal Specification of `make_heap`

Listing 8.22 shows the ACSL specification of `make_heap`. Like with `push_heap` the formal specification of `make_heap` must ensure that the resulting array is a heap of size `n` and contains the same multiset of elements as in the pre-state of the function. These properties are expressed by the `heap` and `reorder` postconditions respectively. The `reorder` postcondition uses the predicate `MultisetUnchanged` (see Listing 6.5) to ensure a reordering of the array elements.

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

   assigns a[0..n-1];

   ensures heap:    IsHeap(a, n);
   ensures max:     n > 0 ==> MaxElement(a, n, 0);
   ensures reorder: MultisetUnchanged{Old,Here}(a, n);
*/
void make_heap(value_type* a, size_type n);
```

Listing 8.22: The Specification of `make_heap`

The `max` property is an additional postcondition which can be derived directly from the `heap` property, see lemma `HeapMaximum` in Listing 8.6. The `MaxElement` predicate itself is introduced in Listing 4.6.

[70] See `http://www.sgi.com/tech/stl/make_heap.html`

8.5.2. Implementation of `make_heap`

The implementation of `make_heap`, shown in Listing 8.23, is straightforward. From low to high the array's elements are pushed to the growing heap. The iteration starts with two, because an array with length one is a heap already.

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

  /*@
     loop invariant bounds:     2 <= i <= n+1;
     loop invariant heap:       IsHeap(a, i-1);
     loop invariant reorder:    MultisetUnchanged{Pre,Here}(a, i);
     loop invariant unchanged:  Unchanged{Pre,Here}(a, i, n);
     loop assigns   i, a[0..n-1];
     loop   variant n - i;
  */
  for (size_type i = 2; i <= n; i++) {
    push_heap(a, i);
  }

  //@ assert  heap: IsHeap(a, n);
}
```

Listing 8.23: The Implementation of `make_heap`

Since the loop statement consists just of a call to `push_heap` we obtain the both loop invariants `heap` and `reorder` by simply lifting them from the contract of `push_heap` (see Section 8.4.1).

The postcondition of `push_heap` only specifies the multiset of elements from index 0 to `i-1`. We therefore also have to specify that the elements from index `i` to `n-1` are only reordered. This property can be derived from the `unchanged` property of `push_heap`.

8.6. The `sort_heap` algorithm

Whereas in the STL `sort_heap` works on a range of random access iterators[71], our version operates on an array of `value_type`. We therefore use the following signature for `sort_heap`

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

The function `sort_heap` rearranges the elements of a given heap `a[0..n-1]` into an array that is sorted in increasing order. Thus, applying `sort_heap` to the heap in Figure 8.3 produces the sorted array in Figure 8.24.

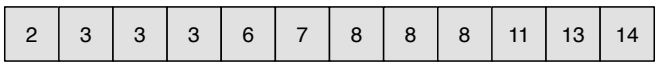


Figure 8.24.: Array after the call of `sort_heap`

8.6.1. Formal Specification of `sort_heap`

Listing 8.25 shows our ACSL specification of `sort_heap`. The formal specification of `sort_heap` must ensure that the resulting array is sorted. Furthermore the multiset contained by the array must be the same as in the pre-state of the function. The postconditions `sorted` and `reorder` express these properties, respectively. The specification effort is relatively simple because we can reuse the previously defined predicates `MultisetUnchanged` (Listing 6.5) and `Sorted` (Listing 5.1).

```
1  /*@
2     requires limit:  0 < n < (INT_MAX-2)/2;
3     requires valid:  \valid(a + (0..n-1));
4     requires heap:   IsHeap(a, n);
5
6     assigns a[0..n-1];
7
8     ensures sorted:   Sorted(a, n);
9     ensures reorder:  MultisetUnchanged{Old, Here}(a, n);
10 */
11 void sort_heap(value_type* a, size_type n);
```

Listing 8.25: Formal specification of `sort_heap`

8.6.2. Implementation of `sort_heap`

The implementation of `sort_heap` (Listing 8.27) is relatively simple because it relies on the `pop_heap` algorithm performing essential work. The `pop_heap` algorithm, whose formal specification is shown in Listing 8.26, rearranges the elements of an array that is a heap of size `n`. After calling `pop_heap`, the previously first element is transferred to the back of the array, while the remaining `n-1` elements still form a heap.

Note that in this version of the document we do *not* provide a verified implementation of `pop_heap`. This, however, does not prevent us to verify `sort_heap` because we only need for this task the function contract of `pop_heap`.

[71] See `http://www.sgi.com/tech/stl/sort_heap.html`

```
/*@
   requires limits: 1 <= n <  (UINT_MAX-2)/2;
   requires valid:  \valid(a + (0..n-1));
   requires heap:   IsHeap(a, n);
   requires max:    MaxElement(a, n, 0);

   assigns a[0..n-1];

   ensures heap:    IsHeap(a, n-1);
   ensures result:  a[n-1] == \old(a[0]);
   ensures max:     MaxElement(a, n, n-1);
   ensures reorder: MultisetUnchanged{Old, Here}(a, n);
*/
void pop_heap(value_type* a, size_type n);
```

Listing 8.26: Formal specification of `pop_heap`

Our implementation of `sort_heap` repeatedly calls `pop_heap` to extract the maximum of the shrinking heap and adding it to the sorted part of the array.

```
void sort_heap(value_type* a, size_type n)
{
  /*@
     loop invariant bound:    0 <= i <= n;
     loop invariant heap:     IsHeap(a, i);
     loop invariant sorted:   Sorted(a, i, n);
     loop invariant lower:    LowerBound(a, i, n, a[0]);
     loop invariant reorder:  MultisetUnchanged{Pre,Here}(a, 0, n);
     loop assigns i, a[0..n-1];
     loop variant i;
  */
  for (size_type i = n; i > 1; --i) {
    /*@
        requires heap:     IsHeap(a, i);
        assigns  a[0..i-1];
        ensures  heap:     IsHeap(a, i-1);
        ensures  max:      a[i-1] == \old(a[0]);
        ensures  max:      MaxElement(a, i, i-1);
        ensures  reorder:  MultisetUnchanged{Old,Here}(a, 0, i);
        ensures  reorder:  Unchanged{Old,Here}(a, i, n);
    */
    pop_heap(a, i);
    //@ assert lower:  LowerBound(a, i, n, a[i-1]);

  }
}
```

Listing 8.27: The Implementation of `sort_heap`

The loop invariants of `sort_heap` describe the content of the array in two parts. The first `i` elements form a heap and are described by the `heap` invariant. The last `n-i` elements are already sorted.

In order to facilitate the automatic verification of the property `sorted` we rely among others on the properties `lower` and `max` and the lemma `SortedUp` from Listing 8.28.

```
/*@
  lemma SortedUp{L}:
    \forall value_type *a, integer n;
      Sorted(a, n)                ==>
      UpperBound(a, 0, n, a[n])   ==>
      Sorted(a, n+1);
*/
```

Listing 8.28: The lemma `SortedUp`

In order to verify the property `reorder` we formulate in Listing 8.29 several lemmas that express that the properties

- `MultisetUnchanged{K,L}(a, 0, i)` and
- `Unchanged{Old,Here}(a, i, n)`

imply the desired loop invariant `MultisetUnchanged{K,L}(a, 0, n)`.

```
/*@
  lemma
    UnchangedImpliesMultisetUnchanged{L1,L2}:
      \forall value_type *a, integer k, n;
        Unchanged{L1,L2}(a, k, n) ==>
        MultisetUnchanged{L1,L2}(a, k, n);


  lemma
    MultisetUnchangedUnion{L1,L2}:
      \forall value_type *a, integer k, n;
        0 <= k <= n                          ==>
        MultisetUnchanged{L1,L2}(a, 0, k)  ==>
        MultisetUnchanged{L1,L2}(a, k, n)  ==>
        MultisetUnchanged{L1,L2}(a, 0, n);


  lemma
    MultisetUnchangedTransitive{L1,L2,L3}:
      \forall value_type *a, integer n;
        MultisetUnchanged{L1,L2}(a, n)  ==>
        MultisetUnchanged{L2,L3}(a, n)  ==>
        MultisetUnchanged{L1,L3}(a, n);
*/
```

Listing 8.29: Lemmas for `MultisetUnchanged`

9. 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 [17].

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 9.1). The *top* function of a stack returns the last element that has been pushed on a stack.

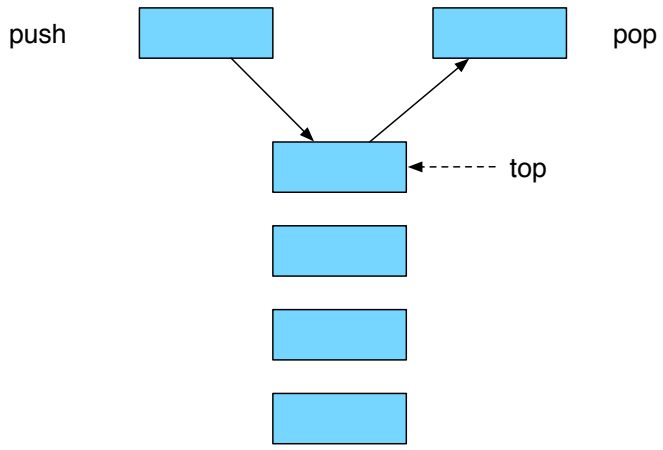


Figure 9.1.: Push and pop on a stack

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.

9.1. Methodology overview

Figure 9.2 gives an overview of our methodology to specify and verify abstract data types (verification of one axiom shown only).

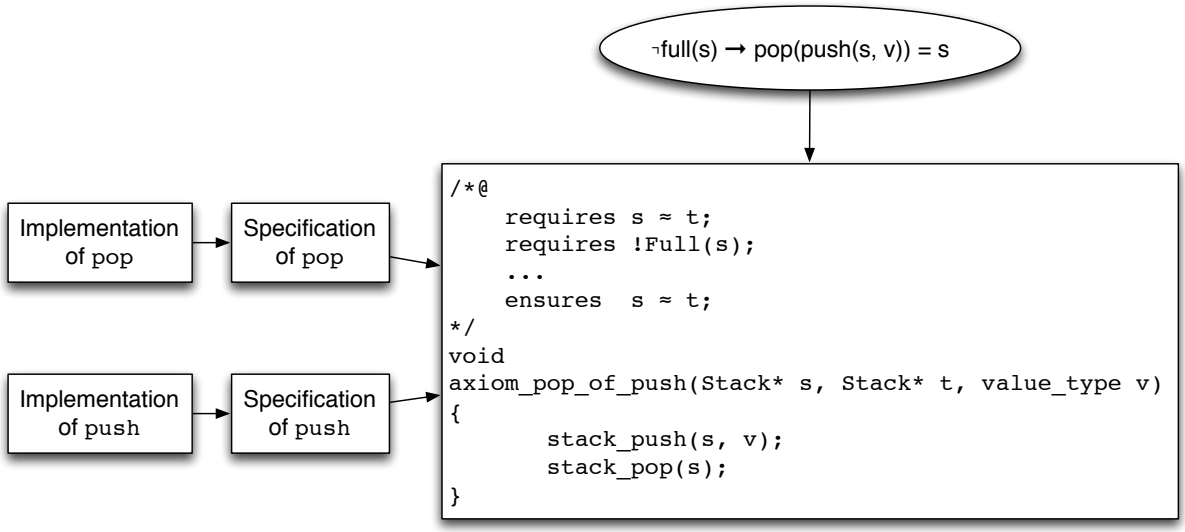


Figure 9.2.: Methodology Overview

What we will basically do is:

1. specify axioms about how the stack functions should interact with each other (Section 9.2),
2. define a basic implementation of C data structures (only one in our example, viz. `struct Stack`; see Section 9.3) and some invariants the instances of them have to obey (Section 9.4),
3. provide for each stack function an ACSL contract and a C implementation (Section 9.7),
4. verify each function against its contract (Section 9.7),
5. transform the axioms into ACSL-annotated C code (Section 9.8), and
6. verify that code, using access function contracts and data-type invariants as necessary (Section 9.8).

Section 9.5 provides an ACSL-predicate deciding whether two instances of a `struct Stack` are considered to be equal (indication by "≈" in Figure 9.2), while Section 9.6 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.

9.2. Stack axioms

To specify the interplay of the stack access functions, we use a set of axioms[72], 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*.

9.2.1. Stack initialization

After a stack has been initialized its size is 0.

$$\text{size}(\text{init}(s)) = 0. \tag{9.1}$$

The auxiliary functions empty and full are defined as follows

$$\text{empty}(s), \quad \text{iff} \quad \text{size(s)} = 0, \tag{9.2}$$

$$\text{full}(s), \quad \text{iff} \quad \text{size(s)} = c. \tag{9.3}$$

We expect that for every stack s the following condition holds

$$0 \leq \text{size}(s) \leq c. \tag{9.4}$$

9.2.2. 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

$$\text{size}(\text{push}(s, v)) = \text{size}(s) + 1, \qquad \text{if} \quad \neg\text{full}(s). \tag{9.5}$$

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

$$\text{top}(\text{push}(s, v)) = v, \qquad \text{if} \quad \neg\text{full}(s). \tag{9.6}$$

[72] There is an analogy in geometry: Euclid (e.g. [18]) invented the use of axioms there, but still kept definitions of *point*, *line*, *plane*, etc. Hilbert [19] recognized that the latter are not only unformalizable, but also unnecessary, and dropped them, keeping only the formal descriptions of relations between them.

9.2.3. 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

$$\text{size}(\text{pop}(s)) = \text{size}(s) - 1, \qquad \text{if} \quad \neg\text{empty}(s). \tag{9.7}$$

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

$$\text{pop}(\text{push}(s, v)) = s, \qquad \text{if} \quad \neg\text{full}(s). \tag{9.8}$$

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.

$$\text{push}(\text{pop}(s), \text{top}(s)) = s, \qquad \text{if} \quad \neg\text{empty}(s). \tag{9.9}$$

9.2.4. 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

$$\text{push}(s, \text{err}) = s. \tag{9.10}$$

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

$$\begin{aligned} \text{size}(s) &= \text{size}(\text{pop}(\text{push}(s, \text{err}))) && \text{by 9.8} \\ &= \text{size}(\text{pop}(s)) && \text{as assumed in 9.10} \\ &= \text{size}(s) - 1 && \text{by 9.7} \end{aligned}$$

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

9.3. 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 9.3.

```
struct Stack
{
  value_type* obj;

  size_type   capacity;

  size_type   size;
};

typedef struct Stack Stack;
```

Listing 9.3: 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 9.4 which attempts to interpret this definition according to Figure 9.1.

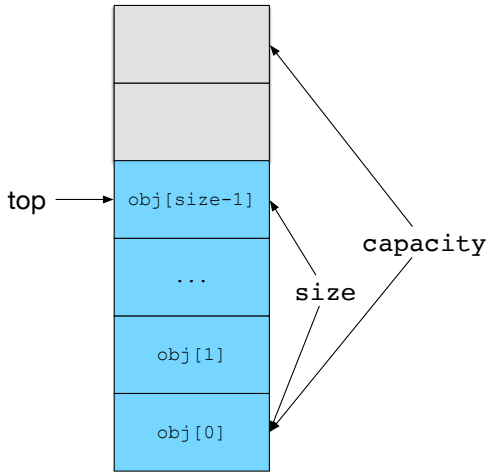


Figure 9.4.: Interpreting the data structure `Stack`

Based on the stack functions from Section 9.2, we declare in Listing 9.5 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 9.5: Declaration of functions of type `Stack`

Most of these functions directly correspond to methods of the C++ `std::stack` template class.[73] 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.

9.4. Stack invariants

Not every possible instance of type `Stack` is considered a valid one, e.g., with our definition of `Stack` in Listing 9.3, `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 9.6 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.

```
/*@
  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 9.6: The logical functions `Capacity`, `Size` and `Top`

We also introduce in Listing 9.7 two predicates that express the concepts of empty and full stacks by referring to a stack's size and capacity (see Equations (9.2) and (9.3)).

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

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

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

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

Listing 9.7: Predicates for empty an full stacks

- 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 9.8.

```
/*@
  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 9.8: 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`.

9.5. 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*[20, 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 9.9.

```
/*@
   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 9.9: Equality of stacks

Our use of labels in Listing 9.9 makes the specification somewhat hard to read (in particular in the last line where we reuse the predicate `EqualRanges` from Page 34). 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.[74]

If equal stacks can have different capacities then, according to our definition of the predicate `Full` in Listing 9.7, 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 9.10 are considered equal although there is are obvious differences in their internal arrays.

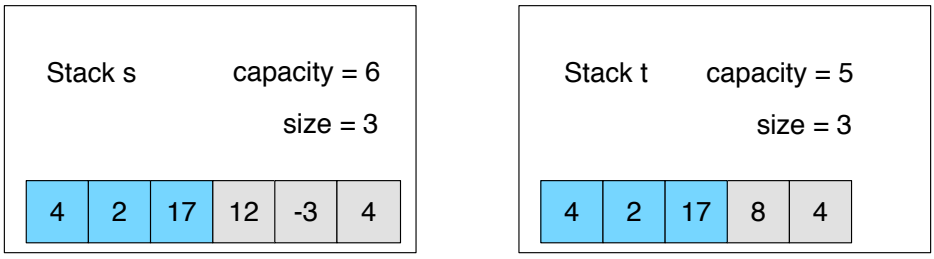


Figure 9.10.: Example of two equal stacks

[74] 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.

$$\text{reflexivity} \qquad \forall s \in S : s = s, \tag{9.11a}$$

$$\text{symmetry} \qquad \forall s, t \in S : s = t \implies t = s, \tag{9.11b}$$

$$\text{transitivity} \qquad \forall s, t, u \in S : s = t \wedge t = u \implies s = u. \tag{9.11c}$$

Any relation that satisfies the conditions (9.11) 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 9.11 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 9.9.

```
/*@
  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 9.11: Equality of stacks is an equivalence relation

The two stacks in Figure 9.10 show that an equivalence class of `Equal` can contain more than one element.[75] The stacks `s` and `t` in Figure 9.10 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*.[76] We ask, in other words, whether the function is *well-defined* on the set of all equivalence classes of the relation `Equal`.[77] The question of well-definition will play an important role when verifying the functions of the `Stack` (see Section 9.7).

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

[76] This is why mathematicians know that $\frac{1}{2} + \frac{3}{5}$ equals $\frac{7}{14} + \frac{3}{5}$.

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

9.6. 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 9.12. 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 9.12: Specification of `stack_equal`

The implementation of `stack_equal` in Listing 9.13 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 9.13: Implementation of `stack_equal`

9.7. Verification of stack functions

In this section we verify the functions `stack_init` (Section 9.7.1), `stack_size` (Section 9.7.2), `stack_empty` (Section 9.7.3), `stack_full` (Section 9.7.4), `stack_top` (Section 9.7.5), and `stack_push` (Section 9.7.6) `stack_pop` (Section 9.7.7), 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`[78] accompanied by a an ACSL contract that expresses that the implementation of `stack_foo` is well-defined. Figure 9.14 shows our methodology for the verification of well-definition in the `pop` example, (≈) again indicating the user-defined `Stack` equality.

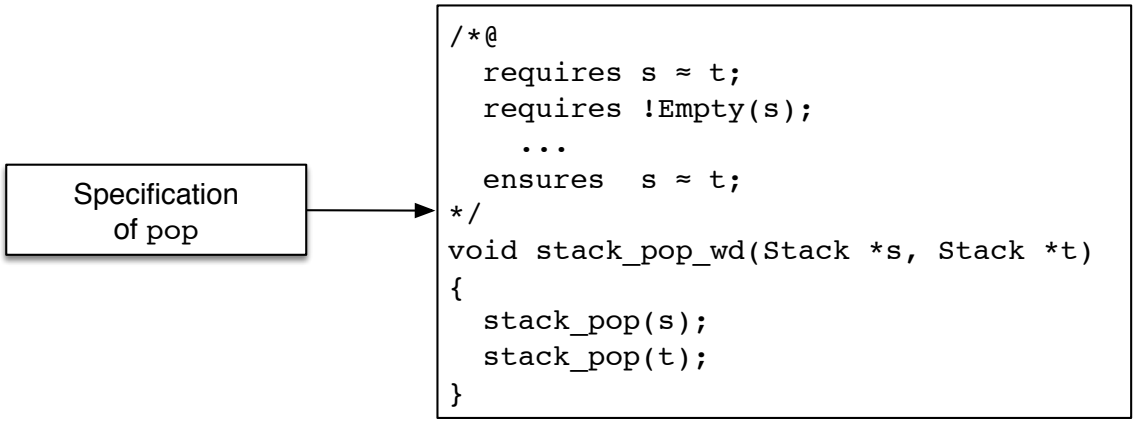


Figure 9.14.: Methodology for the verification of well-definition

Note that the specifications of the various functions will explicitly refer to the *internal state* of `Stack`. In Section 9.8 we will show that the *interplay* of these functions satisfy the stack axioms from Section 9.2.

[78] The suffix `_wd` stands for *well definition*

9.7.1. The function `stack_init`

Listing 9.15 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 9.15: Specification of `stack_init`

Listing 9.15 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 9.16: Implementation of `stack_init`

9.7.2. The function `stack_size`

The function `stack_size` is the runtime version of the logical function `Size` from Listing 9.6 on Page 152. The specification of `stack_size` in Listing 9.17 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 9.17: Specification of `stack_size`

As in the definition of the logical function `Size` the implementation of `stack_size` in Figure 9.18 simply returns the field `size`.

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

Listing 9.18: Implementation of `stack_size`

Listing 9.19 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 9.19: Well-definition of `stack_size`

9.7.3. The function `stack_empty`

The function `stack_empty` is the runtime version of the predicate `Empty` from Listing 9.7 on Page 153.

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

    assigns \nothing;

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

Listing 9.20: Specification of `stack_empty`

As in the definition of the predicate `Empty` the implementation of `stack_empty` in Figure 9.21 simply checks whether the size of the stack is zero.

```
bool stack_empty(const Stack* s)
{
  return stack_size(s) == 0;
}
```

Listing 9.21: Implementation of `stack_empty`

Listing 9.22 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 9.22: Well-definition of `stack_empty`

9.7.4. The function `stack_full`

The function `stack_full` is the runtime version of the predicate `Full` from Listing 9.7 on Page 153.

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

    assigns \nothing;

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

Listing 9.23: Specification of `stack_full`

As in the definition of the predicate `Full` the implementation of `stack_full` in Figure 9.24 simply checks whether the size of the stack equals its capacity.

```
bool stack_full(const Stack* s)
{
  return stack_size(s) == s->capacity;
}
```

Listing 9.24: Implementation of `stack_full`

Note that with our definition of stack equality (Section 9.5) there can be equal stack with different capacities. Accordingly, there can exist equal stacks where one is full while the other is not.

9.7.5. The function `stack_top`

The function `stack_top` is the runtime version of the logical function `Top` from Listing 9.6 on Page 152. The specification of `stack_top` in Listing 9.25 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 9.25: Specification of `stack_top`

For a non-empty stack the implementation of `stack_top` in Figure 9.26 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 9.26: Implementation of `stack_top`

Listing 9.27 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 9.27: Well-definition of `stack_top`

Since our axioms in Section 9.2 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.

9.7.6. The function `stack_push`

Listing 9.28 shows the ACSL specification of the function `stack_push`. In accordance with Axiom (9.5), `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 (9.6)).

```
/*@
  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), 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), 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 9.28: Specification of `stack_push`

The implementation of `stack_push` is shown in Listing 9.29. 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 9.29: Implementation of `stack_push`

The function `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 9.30 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 9.30: The predicate `Separated`

Listing 9.31 shows our formalization of the well-definition for `stack_push`.

```
/*@
  requires valid:     Valid(s) && Valid(t);
  requires equal:     Equal{Here,Here}(s, t);
  requires not_full:  !Full(s) && !Full(t);
  requires separated: Separated(s, t);

  assigns s->size, s->obj[s->size];
  assigns t->size, t->obj[t->size];

  ensures  valid:     Valid(s) && Valid(t);
  ensures  equal:     Equal{Here,Here}(s, t);
*/
void stack_push_wd(Stack* s, Stack* t, value_type v)
{
  stack_push(s, v);
  stack_push(t, v);
  //@ assert top:   Top(s) == v;
  //@ assert top:   Top(t) == v;
  //@ assert equal: EqualRanges{Here,Here}(Storage(s), Size{Pre}(s), Storage(t));
}
```

Listing 9.31: Well-definition of `stack_push`

In order to achieve an automatic verification of the well-definition of `stack_push` we added in Listing 9.31 the assertions `top` and `equal` and introduced the lemma `StackPushEqual` from Listing 9.32.

```
/*@
  lemma StackPushEqual{K,L}:
    \forall Stack *s, *t;
       Equal{K,K}(s,t)              ==>
       Size{L}(s) == Size{K}(s) + 1 ==>
       Size{L}(s) == Size{L}(t)     ==>
       Top{L}(s)  == Top{L}(t)      ==>
       EqualRanges{L,L}(Storage{L}(s), Size{K}(s), Storage{L}(t)) ==>
       Equal{L,L}(s,t);
*/
```

Listing 9.32: The lemma `StackPushEqual`

9.7.7. The function `stack_pop`

Listing 9.33 shows the ACSL specification of the function `stack_pop`. In accordance with Axiom (9.7) `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), 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), 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 9.33: Specification of `stack_pop`

The implementation of `stack_pop` is shown in Listing 9.34. 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 9.34: Implementation of `stack_pop`

Listing 9.35 shows our check whether `stack_pop` is well-defined. As in the case of `stack_push` we use the predicate `Separated` (Listing 9.30) 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 9.35: Well-definition of `stack_pop`

9.8. Verification of stack axioms

In this section we show that the stack functions defined in Section 9.7 satisfy the stack Axioms of Section 9.2.

The annotated code has been obtained from the axioms in a fully systematical way. In order to transform a condition equation $p \rightarrow s = t$:

- 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.

9.8.1. Resetting a stack

Our formulation in ACSL/C of the Axiom in Equation (9.1) on Page 149 is shown in Listing 9.36.

```
/*@
  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 9.36: Specification of Axiom (9.1)

9.8.2. Adding an element to a stack

Axioms (9.5) and (9.6) 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 9.37: Specification of Axiom (9.5)

Except for the `assigns` clauses, the ACSL-specification refers only to encapsulating logical functions and predicates defined in Section 9.4. 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.[79] 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 9.38: Specification of Axiom (9.6)

[79] In [9, §2.3.4], a powerful sublanguage to build memory location set expressions is defined. We will explore its capabilities in a later version.

9.8.3. Removing an element from a stack

This section shows the Listings for Axioms 9.7, 9.8 and 9.9 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 9.39: Specification of Axiom (9.7)

```
/*@
  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 9.40: Specification of Axiom (9.8)

```
/*@
  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 9.41: Specification of Axiom (9.9)

10. Results of formal verification with Frama-C

In this chapter we introduce the formal verification tools used in this tutorial. We will afterwards present to what extent the examples from Chapters 3–9 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. The precise settings for WP and the associated provers that we used in this release we refer to Section 171.

10.1. Verification settings

This section gives all settings that depend on the software release of Frama-C, Why3, or one if its employed provers. For our experiments we used the WP plugin-in of the Aluminium-20160502 release of Frama-C and version 0.87.2 of the Why3 platform.[80]

Each verification conditions was submitted to a set of automatic and interactive theorem provers. Each prover passes on to the next prover only those proof obligations that it could not verify. This *verification pipeline* is shown in Figure 10.1.

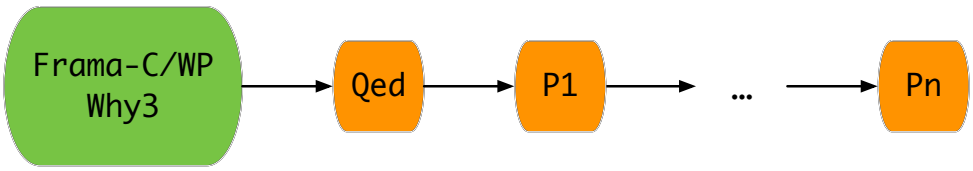


Figure 10.1.: Verification pipeline of automatic and interactive theorem provers

The individual provers (including their versions) are listed in the following Table 10.2.

Prover	ID	Version	Reference
Qed	QD		simplifier that is integrated into WP
CVC4	C4	1.4	`https://cvc4.cs.nyu.edu/web`
Z3	Z3	4.4	`https://github.com/Z3Prover/z3`
Alt-Ergo	AE	1.01	`http://alt-ergo.lri.fr`
CVC3	C3	2.41	`https://www.cs.nyu.edu/acsys/cvc3/`
E	EP	1.9.1	`https://www.eprover.org`
Coq	CQ	8.4.6	`https://coq.inria.fr`

Table 10.2.: Information of automatic and interactive theorem provers

[80] See `http://why3.lri.fr`

Here are the most important options of Frama-C that we used in for almost all functions.

```
-pp-annot -no-unicode
-wp -wp-rte -wp-model Typed+ref
-wp-timeout 10
-wp-steps 1000
-wp-coq-timeout 10
```

Note that for some algorithms, such as `remove` and the second version of `remove_copy`, we have increased the timeout in order to achieve automatic verification.

10.2. Tables of verification results

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 value zero is indicated by an empty cell. The tables show that all verification conditions could be verified. 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 a list of provers (see Figure 10.1 in Section 10.1).

The tables show that the majority of verification conditions could be verified by automatic provers. In Table 10.3 we explicitly list the ACSL lemmas that required induction proofs performed with Coq.

ACSL Lemma	**Listing**
`SortedShift`	5.13
`LowerBoundShift`	5.13
`HasValueCountInversion`	6.40
`CountShift`	6.42
`CountSectionMonotonic`	6.50
`RemoveCountMonotonic`	6.51
`HeapMaximum`	8.6
`MultisetUnchangedUnion`	8.29

Table 10.3.: ACSL lemmas that were proved with Coq

Algorithm		Verification Conditions		Individual Provers QD	C4	Z3	AE	C3	EP	CQ
`find`	§3.1	19/19	(100%)	8	11	·	·	·	·	·
`find` (2)	§3.2	19/19	(100%)	9	10	·	·	·	·	·
`find_first_of`	§3.3	31/31	(100%)	16	14	1	·	·	·	·
`adjacent_find`	§3.4	24/24	(100%)	13	9	·	2	·	·	·
`mismatch`	§3.5	20/20	(100%)	9	11	·	·	·	·	·
`equal`	§3.5	12/12	(100%)	6	5	1	·	·	·	·
`search`	§3.6	30/30	(100%)	18	12	·	·	·	·	·
`search_n`	§3.7	29/29	(100%)	13	11	·	5	·	·	·
`find_end`	§3.8	34/34	(100%)	22	12	·	·	·	·	·
`count`	§3.9	21/21	(100%)	8	12	1	·	·	·	·

Table 10.4.: Results for non-mutating algorithms

Algorithm		Verification Conditions		Individual Provers QD	C4	Z3	AE	C3	EP	CQ
properties of operator <	§4.1	6/ 6	(100%)	4	2	·	·	·	·	·
`max_element`	§4.2	25/25	(100%)	13	12	·	·	·	·	·
`max_element` (2)	§4.3	25/25	(100%)	12	13	·	·	·	·	·
`max_seq`	§4.4	8/ 8	(100%)	5	3	·	·	·	·	·
`min_element`	§4.5	25/25	(100%)	12	13	·	·	·	·	·

Table 10.5.: Results for maximum and minimum algorithms

Algorithm		Verification Conditions		Individual Provers QD	C4	Z3	AE	C3	EP	CQ
`lower_bound`	§5.1	22/22	(100%)	10	12	·	·	·	·	·
`upper_bound`	§5.2	22/22	(100%)	8	14	·	·	·	·	·
`equal_range`	§5.3	20/20	(100%)	15	5	·	·	·	·	·
`equal_range` (2)	§5.3	74/74	(100%)	32	37	3	·	·	·	2
`binary_search`	§5.4	14/14	(100%)	10	4	·	·	·	·	·
`binary_search` (2)	§5.4	15/15	(100%)	10	5	·	·	·	·	·

Table 10.6.: Results for binary search algorithms

Algorithm		Verification Conditions		Individual Provers QD	C4	Z3	AE	C3	EP	CQ
`fill`	§6.3	14/14	(100%)	4	10	·	·	·	·	·
`swap`	§6.4	8/ 8	(100%)	8	·	·	·	·	·	·
`swap_ranges`	§6.5	25/25	(100%)	5	20	·	·	·	·	·
`copy`	§6.6	17/17	(100%)	4	13	·	·	·	·	·
`copy_backward`	§6.7	19/19	(100%)	7	12	·	·	·	·	·
`reverse_copy`	§6.8	19/19	(100%)	4	15	·	·	·	·	·
`reverse`	§6.9	27/27	(100%)	7	20	·	·	·	·	·
`rotate_copy`	§6.10	17/17	(100%)	3	14	·	·	·	·	·
`replace_copy`	§6.11	21/21	(100%)	6	15	·	·	·	·	·
`replace`	§6.12	17/17	(100%)	4	13	·	·	·	·	·
`remove_copy`	§6.13	37/37	(100%)	10	24	3	·	·	·	·
`remove`	§6.14	46/46	(100%)	8	29	2	3	2	·	2
`remove_copy` 2	§6.15	95/95	(100%)	30	58	1	4	·	·	2
`random_shuffle`	§6.16	29/29	(100%)	13	16	·	·	·	·	·

Table 10.7.: Results for mutating algorithms

Algorithm		Verification Conditions		Individual Provers QD	C4	Z3	AE	C3	EP	CQ
`iota`	§7.1	18/18	(100%)	7	11	·	·	·	·	·
`accumulate`	§7.2	14/14	(100%)	6	8	·	·	·	·	·
`inner_product`	§7.3	19/19	(100%)	6	10	3	·	·	·	·
`partial_sum`	§7.4	39/39	(100%)	8	29	1	1	·	·	·
`adjacent_difference`	§7.5	33/33	(100%)	10	23	·	·	·	·	·
`partial_sum_inverse`	§7.6	18/18	(100%)	4	14	·	·	·	·	·
`adjacent_difference_inverse`	§7.7	26/26	(100%)	3	19	1	1	·	2	·

Table 10.8.: Results for numeric algorithms

Algorithm		Verification Conditions		Individual Provers QD	C4	Z3	AE	C3	EP	CQ
`is_heap`	§8.3	22/22	(100%)	8	14	·	·	·	·	·
`push_heap`	§8.4	87/87	(100%)	35	48	1	3	·	·	·
`make_heap`	§8.5	34/34	(100%)	8	23	2	·	·	·	1
`sort_heap`	§8.6	51/51	(100%)	13	32	1	3	·	·	2

Table 10.9.: Results for heap algorithms

Algorithm		Verification Conditions		Individual Provers QD	C4	Z3	AE	C3	EP	CQ
stack_init	§9.7.1	14/14	(100%)	3	11	·	·	·	·	·
stack_equal	§9.6	23/23	(100%)	7	15	1	·	·	·	·
stack_size	§9.7.2	6/6	(100%)	1	5	·	·	·	·	·
stack_empty	§9.7.3	10/10	(100%)	5	5	·	·	·	·	·
stack_full	§9.7.4	11/11	(100%)	5	6	·	·	·	·	·
stack_top	§9.7.5	16/16	(100%)	6	10	·	·	·	·	·
stack_push	§9.7.6	43/43	(100%)	28	15	·	·	·	·	·
stack_pop	§9.7.7	32/32	(100%)	20	12	·	·	·	·	·

Table 10.10.: Results for Stack functions

Algorithm		Verification Conditions		Individual Provers QD	C4	Z3	AE	C3	EP	CQ
stack_size_wd	§9.7.2	12/12	(100%)	8	4	·	·	·	·	·
stack_empty_wd	§9.7.3	12/12	(100%)	8	4	·	·	·	·	·
stack_top_wd	§9.7.5	12/12	(100%)	8	4	·	·	·	·	·
stack_push_wd	§9.7.6	15/15	(100%)	3	12	·	·	·	·	·
stack_pop_wd	§9.7.7	12/12	(100%)	6	6	·	·	·	·	·

Table 10.11.: Results for the well-definition of the Stack functions

Algorithm		Verification Conditions		Individual Provers QD	C4	Z3	AE	C3	EP	CQ
axiom_size_of_pop	§9.8.3	11/11	(100%)	8	3	·	·	·	·	·
axiom_size_of_push	§9.8.2	12/12	(100%)	9	3	·	·	·	·	·
axiom_top_of_push	§9.8.2	11/11	(100%)	8	3	·	·	·	·	·
axiom_pop_of_push	§9.8.3	10/10	(100%)	6	4	·	·	·	·	·
axiom_push_of_pop_top	§9.8.3	15/15	(100%)	9	6	·	·	·	·	·
axiom_size_of_init	§9.8.1	15/15	(100%)	12	3	·	·	·	·	·

Table 10.12.: Results for Stack axioms

A. Changes in previous releases

This chapter describes the changes in previous versions of this document. For the most recent changes see Page 3.

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[81] 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 13.1.0 (Aluminium, August 2016)

The most notable changes of this version are the re-introduction of heap algorithms in Chapter 8. This new description of heap algorithms is based to a large extend on the bachelor thesis of one of the authors [16].

- provide names ("labels") for more ACSL annotations
- non-mutating algorithms
  - reorder and improve description in chapter on non-mutating algorithms
  - add more figures to describe algorithms
  - add non-mutating algorithm `search_n`
  - rewrite logic function `Count` with new logic function `CountSection`
  - move lemmas `CountBounds` and `CountMonotonic` to separate files
  - use `integer` instead of `size_type` in `HasSubRange`
  - change index computation in `HasEqualNeighbors`
- maximum and minimum algorithms
  - isolate predicate `ConstantRange` from predicates on lower and upper bounds
  - fix typo in precondition of first version of `max_element`
- binary search algorithms
  - add version `Sorted` for subranges
  - add second (more efficient) version of `equal_range`

[81] See `http://en.wikipedia.org/wiki/Atomic_number`

        * add lemmas `SortedShift`, `LowerBoundShift`, `StrictLowerBoundShift`, `UpperBoundShift` and `StrictUpperBoundShift` to support the automatic verification of this version of `equal_range`
    - add figures to binary search algorithms and improve description
- mutating algorithms
    - greatly reduce the number of assertions needed to verify the first version `remove_copy`
    - temporarily remove the second version of `remove_copy` which also specified the *stability* of the algorithm
    - add `remove`, an in-place variant of `remove_copy`
    - rename predicate `RetainAllButOne` to `MultisetRetainRest`
- re-introduce chapter on heap algorithms
    - includes the heap algorithms `is_heap`, `push_heap`, `make_heap` and `sort_heap`
    - for `pop_heap` only a function contract is provided in this version
    - add lemma `SortedUp` to support verification of `sort_heap`
    - add several lemmas to combine the predicates `Unchanged` and `MultisetUnchanged`

A.2. New in Version 12.1.0 (Magnesium, February 2016)

A main goal of this release is to reduce the number of proof obligations that cannot be verified automatically and therefore must be tackled by an interactive theorem prover such as Coq. To this end, we analyzed the proof obligations (often using Coq) and devised additional assertions or ACSL lemmas to guide the automatic provers. Often we succeeded in enabling automatic provers to discharge the concerned obligations. Specifically, whereas the previous version 11.1.1 of *ACSL by Example* listed *nine* proof obligations that could only be discharged with Coq, the document at hand (version 12.1.0) only counts *five* such obligations. Moreover, all these remaining proof obligations are associated to ACSL lemmas, which are usually easier to tackle with Coq than proof obligations directly related to the C code. The reason for this is that ACSL lemmas usually have a much smaller set of hypotheses.

Adding assertions and lemmas also helps to alleviate a problem in WP Magnesium and Sodium where prover processes are not properly terminated.[82] Left-over "zombie processes" lead to a deterioration of machine performance which sometimes results in unpredictable verification results.

- mutating algorithms
    - simplify annotations of `replace_copy` and add new algorithm `replace`
        * add predicate `Replace` to write more compact post conditions and loops invariants
    - add several lemmas for predicate `Unchanged` and use predicate `Unchanged` in postconditions of mutating and numeric algorithms
    - simplify annotations of `reverse`
        * rename `Reversed` to `Reverse` (again) and provide another overloaded version
        * add figure to support description of the `Reverse` predicate

[82] See `https://bts.frama-c.com/view.php?id=2154`

  - changes regarding `remove_copy`
    * rename `PreserveCount` to `RetainAllButOne`
    * rename `StableRemove` to `RemoveMapping`
    * add statement contracts for both versions of `remove_copy` such that only ACSL lemmas require Coq proofs
- numeric algorithms
  - define limits `VALUE_TYPE_MIN` and `VALUE_TYPE_MAX`
  - simplify specification of `iota` by using new logic function `Iota`
  - simplify implementation of `accumulate`
    * add overloaded predicates `AccumulateBounds`
    * add lemmas `AccumulateDefault0`, `AccumulateDefault1`, `AccumulateDefaultNext`, and `AccumulateDefaultRead`
  - simplify implementation of `inner_product`
    * add predicates `ProductBounds` and `InnerProductBounds`
  - enable automatic verification of `partial_sum`
    * add lemmas `PartialSumSection`, `PartialSumUnchanged`, `PartialSumStep`, and `PartialSumStep2` to automatically discharge loop invariants
  - enable automatic verification of `adjacent_difference`
    * add logic function `Difference` and predicate `AdjacentDifference`
    * add predicate `AdjacentDifferenceBounds`
    * add lemmas `AdjacentDifferenceStep` and `AdjacentDifferenceSection` to automatically discharge proof obligation
  - add two auxiliary functions `partial_sum_inverse` and `adjacent_difference_inverse` in order to verify that `partial_sum` and `adjacent_difference` are inverse to each other
    * add lemmas `PartialSumInverse` and `AdjacentDifferenceInverse` to support the automatic verification of the auxiliary functions
- stack functions
  - add lemma `StackPushEqual` to enable the automatic verification of the well-definition of `stack_push`

A.3. New in Version 11.1.1 (Sodium, June 2015)

- add Chapter 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)

A.4. New in Version 11.1.0 (Sodium, 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.

A.5. New in Version 10.1.1 (Neon, 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.6. New in Version 10.1.0 (Neon, 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.7. New in Version 9.3.1 (Fluorine, 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.8. New in Version 9.3.0 (Fluorine, 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 [17] and now has a much better verification rate.

A.9. New in Version 8.1.0 (Oxygen, 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.10. New in Version 7.1.1 (Nitrogen, 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.11. New in Version 7.1.0 (Nitrogen, December 2011)

- changed to Frama-C Nitrogen
- changed to Why 2.30
- discussed both plug-ins WP and Jessie
- removed `swap_values` algorithm

A.12. New in Version 6.1.0 (Carbon, not published)

- changed definition of `Stack`
- renamed `reset_stack` to `init_stack`

A.13. New in Version 5.1.1 (Boron, 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.14. New in Version 5.1.0 (Boron, 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.15. New in Version 4.2.2 (Beryllium, 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.16. New in Version 4.2.1 (Beryllium, 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.17. New in Version 4.2.0 (Beryllium, January 2010)

- complete rewrite of pre-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. ACSL 1.11 Implementation in Aluminium-20160501. http://frama-c.com/download/acsl-implementation-Aluminium-20160501.pdf, May 2016.

[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] Lincoln E. Moses and Robert V. Oakford. *Tables of Randon Permutations*. Stanford University Press, 1963.

[15] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Graduate texts in computer science. Springer, New York, 1997.

[16] Timon Lapawczyk. Formale Verifikation von Heap-Algorithmen mit Frama-C. bachelor thesis, Humboldt-Universität zu Berlin, July 2016.

[17] Kim Völlinger. Einsatz des Beweisassistenten Coq zur deduktiven Programmverifikation. Diplomarbeit, Humboldt-Universität zu Berlin, August 2013.

[18] Richard Fitzpatrick J.L. Heiberg. *Euclid's Elements of Geometry*. `http://farside.ph.utexas.edu/euclid.html`, Austin/TX, 2008.

[19] David Hilbert. *Grundlagen der Geometrie*. B.G.Teubner, Stuttgart, 1968.

[20] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.