ACSL by Example

Towards a Formally Verified Standard Library

Version 20.0.1
for
Frama-C 20.0
March 2020

Jens Gerlach
Denis Efremov
Tim Sikatzki

Former Authors

Malte Brodmann, Jochen Burghardt,
Andreas Carben Robert Clausecker,
Liangliang Gu, Kerstin Hartig,
Timon Lapawczyk, Hans Werner Pohl,
Juan Soto, Kim Völlinger

Fraunhofer
FOKUS

This report was partially funded by the VESSEDIA project.[1]

The research leading to these results has received funding from the STANCE project 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]

This document is hosted at

`https://github.com/fraunhoferfokus/acsl-by-example`

From there, you can also download the source code of all algorithms discussed here, their contracts, and the employed predicate definitions and lemmas. All examples are developed and proved with the Frama-C/WP [1] plugin.[4] We recommend using the GitHub issue tracker

`https://github.com/fraunhoferfokus/acsl-by-example/issues`

to report suggestions or errors. Alternatively, you can email them also to

`jens.gerlach@fokus.fraunhofer.de`

[1]The project VESSEDIA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453. Project duration: 2017–2019, see `https://vessedia.eu`

[2]Project duration: 2012–2016, see `http://www.stance-project.eu`

[3]Project duration: 2009–2012

[4]There is also full support for the Frama-C/AstraVer plugin which is developed at ISP RAS and can be installed with the instruction available on `https://forge.ispras.ru/projects/astraver/wiki`

1. Changes

For changes in previous versions we refer to Appendix B on Page 267.

1.1. New in Version 20.0.1 (Calcium, March 2020)

This release is intended for Frama-C [2, v20.0] issued in December 2019. We are also using for this release the Why3 platform [3, v1.2.1] and the provers listed in the following table.

Prover	**Type**	**Version**	**Reference**
Alt-Ergo	automatic	2.3.1	[4]
CVC4	automatic	1.6	[5]
CVC3	automatic	2.4.1	[6]
Z3	automatic	4.8.6	[7]
Coq	interactive	8.9.1	[8]

Table 1.1.: Information on automatic and interactive theorem provers

Note that all automatic provers use the Why3 interface. However, the interactive prover Coq still relies on the native interface provided by Frama-C/WP.

New examples

- add a third version of `find` that is specified using the new logic function `Find`

Improvements

- improve text in many places
- improve specification of `remove_copy` and `remove`
  - provide an explicit definition of `RemovePartition` that allows to replace axioms by lemmas
  - rename predicate `ConstantRange` to `AllEqual` and add its negation `SomeNotEqual`
  - add logic functions `CountNotEqual` and `FindNotEqual`
- place all logic definitions in `axiomatic` blocks to better control generated names
- make names of ACSL predicates, functions and lemmas more uniform and place them together in files where appropriate
- among the renamed ACSL entities are
  - rename predicate `HasValue` to `SomeEqual` and add its negation `NoneEqual`
  - rename lemma `HasValueImpliesPositiveCount` to `SomeEqual_Count`

  - rename lemma `PositiveCountImpliesHasValue` to `Count_SomeEqual`
  - rename `RotatePreservesStrictLowerBound` to `CircularShift_StrictLowerBound`
  - rename `RotateImpliesMultisetUnchanged` to `CircularShift_MultisetUnchanged`

Open issues

The following algorithms and/or lemmas are not completely verified

- `pop_heap`
- `Reorder_Match`

Contents

List of Figures

List of Tables

Part I.

Basics

2. 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 [9]) and the Frama-C/WP plug-in [1] of Frama-C [2] (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). The C++ Standard Library 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**.

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.

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[5] 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, Allan Blanchard, Loïc Correnson, Zaynah Dargaye, Florent Kirchner, Virgile Prevosto, and Armand Puccetti from CEA LIST[6] and Pascal Cuoq from TrustInSoft[7].

We also like to express our gratitude to Claude Marché (LRI/INRIA)[8] and Yannick Moy (AdaCore)[9] for their helpful comments and detailed suggestions for improvement. Finally, we would like to thank Aaron Rocha who sent us valuable improvement suggestions and error reports.

[5]We suggest GitHub's issue tracker: `https://github.com/fraunhoferfokus/acsl-by-example/issues`
[6]`http://www-list.cea.fr/en`
[7]`http://trust-in-soft.com`
[8]`https://www.lri.fr/index_en.php?lang=EN`
[9]`http://www.adacore.com`

2.1. Frama-C

The Framework for Modular Analyses of C, Frama-C [2], 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 [10], a laboratory of applied research on software-intensive technologies, and INRIA Saclay [11], the French National Institute for Research in Computer Science and Control in collaboration with LRI [12], the Laboratory for Computer Science at Université Paris-Sud.

ACSL (ANSI/ISO-C Specification Language) [9] 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 Frama-C/WP plug-in [1] enables deductive verification of C programs that have been annotated with ACSL. The Frama-C/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 [13] 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 Frama-C/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 [14] first. The principles presented in this paper are also documented in the ACSL reference document [15].

2.2. 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 3 provides a short introduction into the Hoare Calculus. For a better understanding of Frama-C/WP and the theory behind it, we also recommend Allan Blanchard's ACSL tutorial [16].

We have grouped various standard algorithms in chapters as follows:

- non-mutating algorithms (Chapter 4)
- maximum/minimum algorithms (Chapter 5)
- binary search algorithms (Chapter 6)
- mutating algorithms (Chapter 7)
- numeric algorithms (Chapter 9)
- heap algorithms (Chapter 10)
- sorting algorithms and well-known classical implementations of sorting algorithms (Chapter 11)

The order of these chapters reflects their increasing complexity.

Using the example of a stack, we tackle in Chapter 12 the problem of how a data type and its associated C functions can be specified with ACSL and automatically verified with Frama-C.

Finally, Appendix A lists for each example the results of verification with Frama-C.

2.3. 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, say, 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.

2.3.1. Array and ranges

The C Standard describes an array as a "contiguously allocated nonempty set of objects" [17, §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.

2.3.2. 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`.

3. The Hoare calculus

In 1969, C.A.R. Hoare introduced a calculus for formal reasoning about properties of imperative programs [18], 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 [15, §2.2] in full detail. For the purposes of this tutorial, the notions shown in Table 3.1 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 3.1.: 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.[10] 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 3.2 to 3.12 in order to reason about programs. We will comment on them in the following sections.

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

3.1. 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 3.2, 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 3.2: 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 3.3 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 3.3: 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 3.3, 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 3.4.

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

Listing 3.4: Simpler, but *faulty* assignment rule

Listings 3.5–3.7 show some example instances of this faulty rule.

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

Listing 3.5: An example instance of the faulty rule from Listing 3.4

While Listing 3.5 happens to be ok, Listing 3.6 and 3.7 lead to postconditions that are obviously nonsensical formulas.

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

Listing 3.6: An example instance of the faulty rule from Listing 3.4

The reason is that in the assignment in Listing 3.6 the left-hand side variable `x` also appears in the right-hand side expression `e`, while the assignment in Listing 3.7 just destroys the property from its precondition.

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

Listing 3.7: An example instance of the faulty rule from Listing 3.4

Note that the correct example Listing 3.5 can as well be obtained as an instance of the correct rule from Listing 3.2, 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`.

3.2. The sequence rule

The sequence rule, shown in Listing 3.8, 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.

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

and

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

$\leadsto$

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

Listing 3.8: The sequence rule

3.3. The implication rule

The implication rule, shown in Listing 3.9, 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 3.9 by the right one.

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

$\leadsto$

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

Listing 3.9: The implication rule

3.4. The choice rule

The choice rule, depicted in Listing 3.10, 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`. For example, in the else-branch, we may use that `C` 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;
```

$\leadsto$

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

Listing 3.10: The choice rule

Listing 3.11 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 3.11: 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.

3.5. The loop rule

The loop rule, shown in Listing 3.12, 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.

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

$\leadsto$

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

Listing 3.12: 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 emphasize 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 3.13 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 3.11 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 3.13: An abstract ring buffer loop

To reuse the proof from Listing 3.11, we had to drop the conjunct `!done`, since we didn't consider it in Listing 3.11. 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 3.11, 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 3.11. 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 7.3 on Page 97.

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`[11] 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 3.11 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()`, for example since `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.

[11] 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`.

3.6. 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 [19] 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`.

Part II.

Nonmutating and simple search algorithms

4. Non-mutating algorithms

In this chapter, we consider *non-mutating* algorithms of the C++ Standard Library [20, §28.5]. These algorithms neither change their arguments nor any objects outside their scope. This requirement can be formally expressed with the following *assigns clause*:

```
assigns \nothing;
```

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

The specifications of these algorithms are not very complex. Nevertheless, we have tried to arrange them so that the earlier examples are simpler than the later ones. Each algorithm works on one-dimensional arrays.

- `find` (Section 4.1) provides *sequential* or *linear search* and returns the smallest index at which a given value occurs in a given range. In Section 4.2, a user-defined ACSL predicate is introduced in order to simplify the reuse of various specification elements. We refer to the simplified version as `find2`. We provide in Section 4.3 a third specification of `find` (called `find3`) that relies on a user-defined ACSL function that expresses the ideas of linear search on the logic level.
- `find_first_of` (Section 4.4) provides similar to `find` a *sequential search*. However, unlike `find` it does not search for a particular value, but for an arbitrary member of a set.
- `adjacent_find` (Section 4.5) can be used to find equal neighbors in an array.
- `equal` and `mismatch` (Section 4.6) are useful for comparing two ranges element-by-element and identifying where they differ.
- `search` and `search_n` (Sections 4.7 and 4.8) 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 4.10) returns the number of occurrences of a given value in a range. Here we will explicitly define a logic function for elements counting and show that the implementation comply with it.
- `count2` (Section 4.11) contains different specification for the `count` function. In this case an inductive predicate defined for elements counting. The section allows one to compare different approaches of writing specifications and demonstrates the ACSL inductive predicates.

4.1. The `find` algorithm

The `find` algorithm in the C++ Standard Library [20, §28.5.5] implements *sequential search* for general sequences. 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 4.1 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.

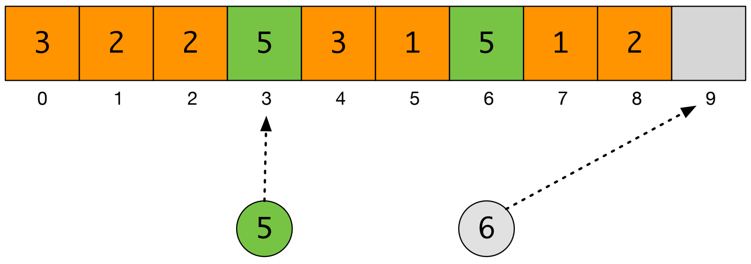


Figure 4.1.: Some simple examples for `find`

4.1.1. Formal specification of `find`

The formal specification of `find` in ACSL is shown in Listing 4.2.

```
/*@
  requires    \valid_read(a + (0..n-1));
  assigns     \nothing;
  ensures     0 <= \result <= n;

  behavior some:
    assumes   \exists integer i; 0 <= i < n && a[i] == val;
    assigns   \nothing;
    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;
    assigns   \nothing;
    ensures   \result == n;

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

Listing 4.2: Formal specification of `find`

The `requires`-clause indicates that `n` is non-negative and that the pointer `a` points to *n* contiguously allocated objects of type `value_type` (see Section 2.3). The `assigns`-clause indicates that `find` (as a non-mutating algorithm), does not modify any memory location outside its scope (see Page 31).

Generally, we only know that `find` returns a non-negative index that is less or equal the length of the array. However, once we assume more specific situations, we can also make more precise statements about the returned valued. This is the reason why 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` returns 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 4.2). 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*.

4.1.2. Implementation of **`find`**

Listing 4.3 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 = 0u; i < n; i++) {
    if (a[i] == val) {
      return i;
    }
  }

  return n;
}
```

Listing 4.3: 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 4.2). 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.

4.2. The `find` algorithm—reuse of specification elements

In this section we specify the `find` algorithm in a slightly different way when compared to Section 4.1. Our approach is motivated by a considerable number of closely related formulas. We have in Listings 4.2 and 4.3 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.

4.2.1. The predicates `SomeEqual` and `NoneEqual`

In order to be more explicit about the commonalities of these formulas we define a predicate, called `SomeEqual` (see Listing 4.4), which describes the situation that there is a valid index `i` where `a[i]` equals `val`.

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

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

    predicate
    NoneEqual(value_type* a, integer m, integer n, value_type v) =
      \forall integer i; m <= i < n  ==>  a[i] != v;

    predicate
    NoneEqual(value_type* a, integer n, value_type v) =
      NoneEqual(a, 0, n, v);

    lemma NotSomeEqual_NoneEqual:
      \forall value_type *a, v, integer m, n;
        !SomeEqual(a, m, n, v)  ==>  NoneEqual(a, m, n, v);

    lemma NoneEqual_NotSomeEqual:
      \forall value_type *a, v, integer m, n;
       NoneEqual(a, m, n, v)   ==>  !SomeEqual(a, m, n, v);
  }
*/
```

Listing 4.4: The predicates `SomeEqual` and `NoneEqual`

We first remark that the predicates `SomeEqual` and `NoneEqual` are encapsulated in an `axiomatic` block. This is a *feeble* attempt to establish some modularization for the various predicates, logic functions and lemmas. We say *feeble* because `axiomatic` blocks are, in contrast to ACSL `modules`, *not* name spaces. ACSL modules, however, are not yet implemented by Frama-C.

Note that we have provided a label, viz. `A`, to the predicate `SomeEqual`. Its purposes to express that the evaluation of 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 `SomeEqual` or `NoneEqual`, the label `A` is the only available label. With these predicates we can encapsulate all uses of the universal and existential quantifiers in both the function contract of `find2` and in its loop annotations.

4.2.2. Formal specification of `find`

Th revised contract for `find` in Listing 4.5 is more concise than the previous one in Listing 4.2. 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 `NoneEqual` predicate, as depicted with the last postcondition of behavior `some` shown in Listing 4.5.

```
/*@
  requires valid:    \valid_read(a + (0..n-1));
  assigns            \nothing;
  ensures result:    0 <= \result <= n;

  behavior some:
    assumes          SomeEqual(a, n, val);
    assigns          \nothing;
    ensures  bound:  0 <= \result < n;
    ensures  result: a[\result] == val;
    ensures  first:  NoneEqual(a, \result, val);

  behavior none:
    assumes          NoneEqual(a, n, val);
    assigns          \nothing;
    ensures  result: \result == n;

  complete behaviors;
  disjoint behaviors;
*/
size_type
find2(const value_type* a, size_type n, value_type val);
```

Listing 4.5: Formal specification of `find` using the predicates `SomeEqual` and `NoneEqual`

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.

4.2.3. Implementation of `find`

The predicate `NoneEqual` 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
find2(const value_type* a, size_type n, value_type val)
{
  /*@
    loop invariant bound:     0 <= i <= n;
    loop invariant not_found: NoneEqual(a, i, val);
    loop assigns i;
    loop variant n-i;
   */
  for (size_type i = 0u; i < n; i++) {
    if (a[i] == val) {
      return i;
    }
  }

  return n;
}
```

Listing 4.6: Implementation of `find` with loop annotations based on `NoneEqual`

4.3. A different approach to specify the `find` algorithm

In this section we specify the `find` algorithm in yet another way. This third approach requires more preparing work but results in a more compact specification.

4.3.1. The logic function `Find`

We start with a *recursive* definition of the ACSL function `Find`. Due to the considerable number of associated lemmas of the function `Find` we display its definition into two listing 4.7 and 4.8.

```
/*@
  axiomatic Find
  {
    logic integer
    Find(value_type* a, integer m, integer n, value_type v) =
      (n <= m) ?
       0 : ((0 <= Find(a, m, n-1, v) < n-m-1) ?
         Find(a, m, n-1, v) : ((a[n-1] == v) ? n-m-1 : n-m));

    logic integer
    Find(value_type* a, integer n, value_type v) = Find(a, 0, n, v);

    lemma Find_Empty:
      \forall value_type *a, v, integer m, n;
        n <= m   ==>  Find(a, m, n, v) == 0;

    lemma Find_Hit:
      \forall value_type *a, v, integer m, n;
        m <= n                   ==>
        Find(a, m, n, v) < n-m  ==>
        Find(a, m, n+1, v) == Find(a, m, n, v);

    lemma Find_MissHit:
      \forall value_type *a, v, integer m, n;
        m <= n                      ==>
        a[n] == v                   ==>
        Find(a, m, n, v)   == n-m   ==>
        Find(a, m, n+1, v) == n-m;

    lemma Find_MissMiss:
      \forall value_type *a, v, integer m, n;
        m <= n                      ==>
        a[n] != v                   ==>
        Find(a, m, n, v)   == n-m   ==>
        Find(a, m, n+1, v) == (n+1)-m;
```

Listing 4.7: The logic function `Find` (1)

Listing 4.7 contains lemmas that express elementary properties directly related to an incremental increase of the array `a[0..n-1]`. Listing 4.8, on the other hand, shows somewhat more higher-level that will be useful for the verification of the contract in Listing 4.9.

```
    lemma Find_Lower:
      \forall value_type *a, v, integer m, n;
        0 <= Find(a, m, n, v);

    lemma Find_Upper:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  Find(a, m, n, v) <= n-m;

    lemma Find_WeaklyIncreasing:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  Find(a, m, n, v) <= Find(a, m, n+1, v);

    lemma Find_Extend:
      \forall value_type *a, v, integer k, m, n;
        m <= k < n                 ==>
        a[k] == v                  ==>
        Find(a, m, k, v) == k-m    ==>
        Find(a, m, n, v) == k-m;

    lemma Find_Increasing:
      \forall value_type *a, v, integer k, m, n;
        m <= k <= n  ==>
        Find(a, m, k, v) <= Find(a, m, n, v);

    lemma Find_Limit:
      \forall value_type *a, v, integer k, m, n;
        m <= k < n  ==>
        a[k] == v   ==>
        Find(a, m, n, v) <= k-m;

    lemma Find_NoneEqual:
      \forall value_type *a, v, integer m, n;
        m <= n                  ==>
        NoneEqual(a, m, n, v)   ==>
        Find(a, m, n, v) == n-m;

    lemma Find_SomeEqual:
      \forall value_type *a, v, integer k, m, n;
        m <= k < n              ==>
        a[k] == v               ==>
        NoneEqual(a, m, k, v)   ==>
        Find(a, m, n, v) == k-m;

    lemma Find_ResultNoneEqual:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  NoneEqual(a, m, m + Find(a, m, n, v), v);

    lemma Find_ResultEqual:
      \forall value_type *a, v, integer m, n;
        0 <= Find(a, m, n, v) < n-m  ==>
        a[m + Find(a, m, n, v)] == v;
  }
*/
```

Listing 4.8: The logic function `Find` (2)

4.3.2. Formal specification of `find`

Using the logic function `Find` we can now give in Listing 4.9 a third and much shorter specification of the algorithm `find`.

```
/*@
  requires valid:    \valid_read(a + (0..n-1));
  assigns            \nothing;
  ensures result:    0 <= \result <= n;
  ensures result:    \result == Find(a, n, v);
*/
size_type
find3(const value_type* a, size_type n, value_type v);
```

Listing 4.9: Formal specification of `find` using `Find`

4.3.3. Implementation of `find`

Listing 4.10 shows the implementation of this version of `find`.

```
size_type
find3(const value_type* a, size_type n, value_type v)
{
  /*@
    loop invariant bound:     0 <= i <= n;
    loop invariant not_found: Find(a, i, v) == i;
    loop assigns i;
    loop variant n-i;
   */
  for (size_type i = 0u; i < n; i++) {
    if (a[i] == v) {
      //@ assert found: Find(a, n, v) == i;
      return i;
    }
  }

  return n;
}
```

Listing 4.10: Implementation of `find` with annotations relying on `Find`

4.4. The `find_first_of` algorithm

The `find_first_of` algorithm [20, §28.5.7] is closely related to `find` (see Sections 4.1 and 4.2).

```
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, the function `find_first_of` returns the least index `i` such that `a[i]` is equal to one of the values `b[0..n-1]`.

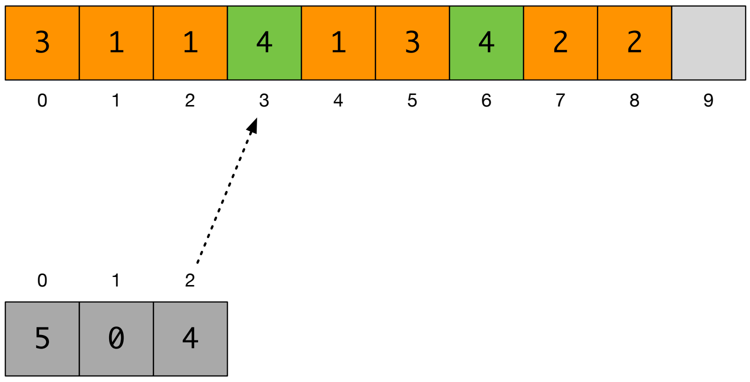


Figure 4.11.: A simple example for `find_first_of`

As an example, we consider in Figure 4.11 two arrays. The arrow indicates the smallest index where one of the elements of the three-element array occurs.

4.4.1. The predicate `HasValueOf`

Similar to our approach in Section 4.2, 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 `SomeEqual` (Listing 4.4) to define `HasValueOf` (Listing 4.12).

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

Listing 4.12: The predicate `HasValueOf`

4.4.2. Formal specification of `find_first_of`

The formal specification of `find_first_of` is shown Listing 4.13. The function contract uses the predicates `HasValueOf` and `SomeEqual` thereby making it very similar the specification `find` (Listing 4.5).

```
/*@
  requires valid:   \valid_read(a + (0..m-1));
  requires valid:   \valid_read(b + (0..n-1));
  assigns           \nothing;
  ensures result:   0 <= \result <= m;

  behavior found:
    assumes         HasValueOf(a, m, b, n);
    assigns         \nothing;
    ensures bound:  0 <= \result < m;
    ensures result: SomeEqual(b, n, a[\result]);
    ensures first:  !HasValueOf(a, \result, b, n);

  behavior not_found:
    assumes         !HasValueOf(a, m, b, n);
    assigns         \nothing;
    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 4.13: Formal specification of find_first_of

4.4.3. Implementation of find_first_of

Our implementation of find_first_of is shown in Listing 4.14. Note the call of the find function. 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.

```
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 = 0u; i < m; i++) {
    if (find2(b, n, a[i]) < n) {
      return i;
    }
  }

  return m;
}
```

Listing 4.14: Implementation of find_first_of

4.5. The `adjacent_find` algorithm

The `adjacent_find` algorithm of the C++ Standard Library [20, §28.5.8]

```
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 4.15 indicates the smallest index where two adjacent elements are equal.

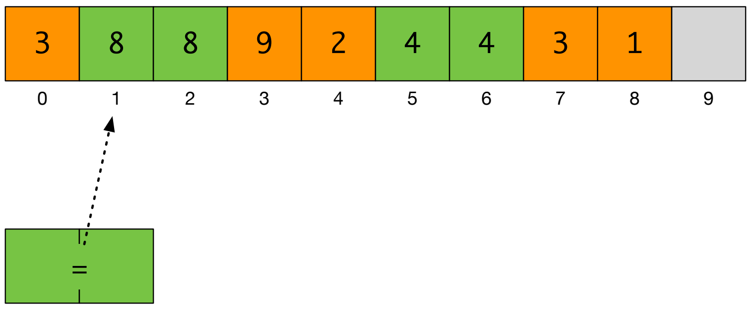


Figure 4.15.: A simple example for `adjacent_find`

4.5.1. The predicate `HasEqualNeighbors`

As in the case of other search algorithms, we first define a predicate `HasEqualNeighbors` (see Listing 4.16) that captures the essence of finding two adjacent indices at which the array holds equal values.

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

Listing 4.16: The predicate `HasEqualNeighbors`

4.5.2. Formal specification of `adjacent_find`

We use the predicate `HasEqualNeighbors` to define the formal specification of `adjacent_find` (see Listing 4.17).

```
/*@
  requires valid:       \valid_read(a + (0..n-1));
  assigns               \nothing;
  ensures result:       0 <= \result <= n;

  behavior some:
    assumes             HasEqualNeighbors(a, n);
    assigns             \nothing;
    ensures  result:    0 <= \result < n-1;
    ensures  adjacent:  a[\result] == a[\result+1];
    ensures  first:     !HasEqualNeighbors(a, \result);

  behavior none:
    assumes             !HasEqualNeighbors(a, n);
    assigns             \nothing;
    ensures  result:    \result == n;

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

Listing 4.17: Formal specification of `adjacent_find`

4.5.3. Implementation of `adjacent_find`

The implementation of `adjacent_find`, including loop annotations is shown in Listing 4.18. At the beginning we check whether the array contains at least two elements. Otherwise, there is no point in looking for adjacent neighbors. Note the use of the predicate `HasEqualNeighbors` in the loop invariant to match the similar postcondition of behavior `some`.

```
size_type
adjacent_find(const value_type* a, size_type n)
{
  if (1u < n) {
    /*@
      loop invariant bound:  0 <= i < n;
      loop invariant none:   !HasEqualNeighbors(a, i+1);
      loop assigns i;
      loop variant n-i;
    */
    for (size_type i = 0u; i + 1u < n; ++i) {
      if (a[i] == a[i + 1u]) {
        return  i;
      }
    }
  }

  return n;
}
```

Listing 4.18: Implementation of `adjacent_find`

4.6. The equal and mismatch algorithms

The equal [20, §28.5.11] and mismatch [20, §28.5.10] 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.

4.6.1. 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.[12] 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.

We start with introducing in Listing 4.19 several *overloaded* versions of the predicate EqualRanges.

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

    predicate
    EqualRanges{K,L}(value_type* a, integer m, integer n, value_type* b) =
      \forall integer i; m <= i < n  ==>  \at(a[i],K) == \at(b[i],L);

    predicate
    EqualRanges{K,L}(value_type* a, integer m, integer n,
                     value_type* b, integer p) = EqualRanges{K,L}(a+m, n-m, b+p);

    predicate
    EqualRanges{K,L}(value_type* a, integer m, integer n, integer p) =
      EqualRanges{K,L}(a, m, n, a, p);
  }
*/
```

Listing 4.19: Overloaded versions of predicate EqualRanges

The letters K and L in the definition of EqualRanges are so-called *labels*[13] that refer to program states in which the ranges a[..] and b[..] are evaluated. Frama-C defines several standard labels, e.g. Old

[12]Compare http://en.wikipedia.org/wiki/Don't_repeat_yourself
[13]Labels are used in C to name the target of the *goto* jump statement.

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 [15, §2.6.9].

4.6.2. Formal specification of equal and mismatch

Using predicate EqualRanges we can formulate the specification of equal in Listing 4.20, 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 4.20: Formal specification of equal

The formal specification of mismatch in Listing 4.21 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 4.5). 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;
  ensures  result:   0 <= \result <= n;

  behavior all_equal:
    assumes          EqualRanges{Here,Here}(a, n, b);
    assigns          \nothing;
    ensures result: \result == n;

  behavior some_not_equal:
    assumes          !EqualRanges{Here,Here}(a, n, b);
    assigns          \nothing;
    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 4.21: 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.

4.6.3. Implementation of `equal` and `mismatch`

Listing 4.22 shows an implementation of the `equal` algorithm by a simple call of `mismatch`.

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

Listing 4.22: Implementation of `equal` with `mismatch`

Listing 4.23 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 = 0u; i < n; i++) {
    if (a[i] != b[i]) {
      return i;
    }
  }

  return n;
}
```

Listing 4.23: Implementation of `mismatch`

We use the predicate `EqualRanges` 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.

4.7. The `search` algorithm

The `search` algorithm in the C++ Standard Library [20, §28.5.13] 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`. The signature now reads:

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

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 < p` (see Figure 4.24). If no such index exists, then `search` returns the length `n` of the array `a`.

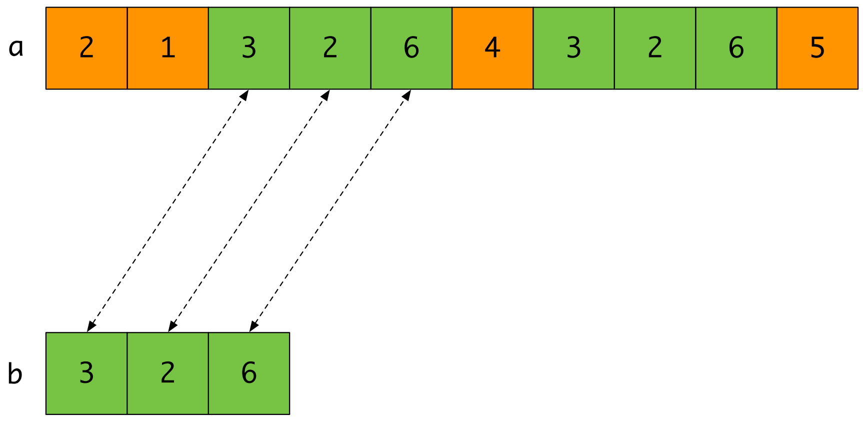


Figure 4.24.: Searching the first occurrence of `b[0..p-1]` in `a[0..n-1]`

4.7.1. The predicate `HasSubRange`

Our specification of `search` starts with introducing the predicate `HasSubRange` in Listing 4.25. This predicate formalizes, using the predicate `EqualRanges` defined in Listing 4.19, that the sequence `a` contains a subsequence which equal the sequence `b`. Of course, in order to contain a subsequence of length `p`, `a` must be at least that large; this is expressed by lemma `HasSubRangeSizes`.

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

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

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

Listing 4.25: The predicate `HasSubRange`

4.7.2. Formal specification of search

The ACSL specification of search is shown in Listing 4.26.

```
/*@
  requires valid:   \valid_read(a + (0..n-1));
  requires valid:   \valid_read(b + (0..p-1));
  assigns           \nothing;
  ensures  result:  0 <= \result <= n;

  behavior has_match:
    assumes         HasSubRange(a, n, b, p);
    assigns         \nothing;
    ensures bound:  0 <= \result <= n-p;
    ensures result: EqualRanges{Here,Here}(a+\result, p, b);
    ensures first:  !HasSubRange(a, \result+p-1, b, p);

  behavior no_match:
    assumes         !HasSubRange(a, n, b, p);
    assigns         \nothing;
    ensures result: \result == n;

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

Listing 4.26: Formal specification of search

Conceptually, the specification of search is very similar to that of find (Section 4.1). 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.

  The ensures clause bound of behavior has_match indicates that the returned index value must be in the range [0..n-p]. The clause result expresses that search returns an index where a copy of b can be found in a. Clause first indicates that the least index with that property is returned, i.e. that b can't be found in a[0..\result+p-2].

- The behavior no_match covers the case that there is no subsequence a that equals b. In this case, search must return the length n 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*.

4.7.3. Implementation of search

Our implementation of search is shown in Listing 4.27. It follows the C++ Standard Library implementation in being easy to understand, but needing an order of magnitude of n*p operations. In contrast, the sophisticated algorithm from [21] needs only n+p operations.[14]

The loop invariant not_found is needed for the proof of the postconditions of the behavior has_match (see Listing 4.26). It expresses that the subsequence b has not been found up to the current iteration step. Neither p == 0 nor n == 0 need to be handled separately, not even for efficiency reasons: in the former case, equal(a+i, p, b) will succeed in the first iteration, while in the latter, p > n will apply.

```
size_type
search(const value_type* a, size_type n,
       const value_type* b, size_type p)
{
  if (p <= n) {
    /*@
      loop invariant bound:     i <= n-p+1;
      loop invariant not_found: !HasSubRange(a, p+i-1, b, p);
      loop assigns i;
      loop variant n-i;
    */
    for (size_type i = 0u; i <= n - p; ++i) {
      if (equal(a + i, p, b)) {
        //@ assert has_match: HasSubRange(a, n, b, p);
        return i;
      }
    }
  }

  //@ assert no_match: !HasSubRange(a, n, b, p);
  return n;
}
```

Listing 4.27: Implementation of search

[14] 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

4.8. The `search_n` algorithm

The `search_n` algorithm in the C++ Standard Library [20, §28.5.13] finds the first place where a given value starts 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`. The signature now reads:

```
size_type
search_n(const value_type* a, size_type n, size_type p, value_type v)
```

Note the similarity to the signature of `search` (Section 4.7). The only difference is that `v` now is a single value rather than an array.

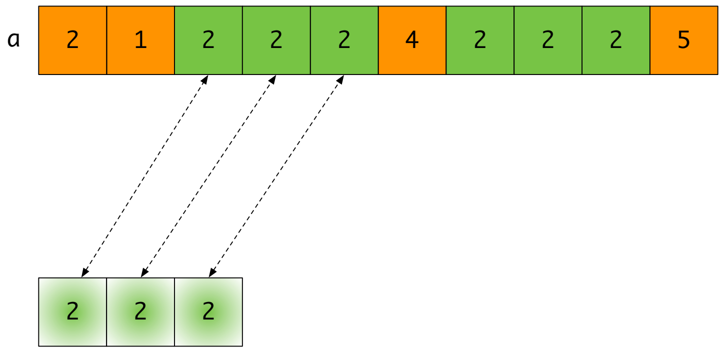


Figure 4.28.: Searching the first occurrence a given constant sequence in `a[0..n-1]`

The function `search_n` returns the first index `s` of the array `a` where the condition `a[s+k] == v` holds for each index `k` with `0 <= k < p` (see Figure 4.28). If no such index exists, then `search_n` returns the length `n` of the array `a`.

4.8.1. The predicates `AllEqual` and `HasConstantSubRange`

Our specification of `search_n` starts with introducing the predicate `AllEqual` in Listing 4.29, which expresses that each member `a[m..n-1]` equals `v`. We also introduce the predicate `SomeNotEqual` which is the negation of `AllEqual`. These predicates complement the predicates `SomeEqual` and `NoneEqual` from Listing 4.4.

There are two additional overloaded versions of `AllEqual`. The first version uses the value `a[m]` as `v`. We will use this particular version later on during the specification of `unique_copy` (Listing 8.3). The second version is just a shortcut when the first index `m` equals 0.

```
/*@
  axiomatic AllSomeNot
  {
    predicate
    AllEqual(value_type* a, integer m, integer n, value_type v) =
      \forall integer i; m <= i < n  ==>  a[i] == v;

    predicate
    AllEqual(value_type* a, integer m, integer n) =
      AllEqual(a, m, n, a[m]);

    predicate
    AllEqual(value_type* a, integer n, value_type v) =
      AllEqual(a, 0, n, v);

    predicate
    SomeNotEqual{A}(value_type* a, integer m, integer n, value_type v) =
      \exists integer i; m <= i < n && a[i] != v;

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

    lemma NotAllEqual_SomeNotEqual:
      \forall value_type *a, v, integer m, n;
        !AllEqual(a, m, n, v)  ==>  SomeNotEqual(a, m, n, v);

    lemma SomeNotEqual_NotAllEqual:
      \forall value_type *a, v, integer m, n;
       SomeNotEqual(a, m, n, v)   ==>  !AllEqual(a, m, n, v);
  }
*/
```

Listing 4.29: The predicates `AllEqual` and `SomeNotEqual`

Based on that, the predicate `HasConstantSubRange` in Listing 4.30 formalizes that the sequence `a` of length `n` contains a subsequence of `p` times the value `v`. Similar to `HasSubRange`, in order to contain `p` repetitions, `a` must be at least that large; this is what lemma `HasConstantSubRangeSizes` says.

```
/*@
  axiomatic HasConstantSubRange
  {
    predicate
    HasConstantSubRange{L}(value_type* a, integer m, integer n, value_type v, integer
         p) =
      \exists integer k; m <= k <= n-p && AllEqual(a, k, k+p, v);

    predicate
    HasConstantSubRange{L}(value_type* a, integer n, value_type v, integer p) =
      HasConstantSubRange(a, 0, n, v, p);

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

Listing 4.30: The predicate `HasConstantSubRange`

4.8.2. Formal specification of search_n

The ACSL specification of search_n is shown in Listing 4.31. Like for search, the specification of search_n is very similar to that of find.

```
/*@
  requires valid:      \valid_read(a + (0..n-1));
  assigns              \nothing;
  ensures  result:     0 <= \result <= n;

  behavior has_match:
    assumes            HasConstantSubRange(a, n, v, p);
    assigns            \nothing;
    ensures result:    0 <= \result <= n-p;
    ensures match:     AllEqual(a, \result, \result+p, v);
    ensures first:     !HasConstantSubRange(a, \result+p-1, v, p);

  behavior no_match:
    assumes            !HasConstantSubRange(a, n, v, p);
    assigns            \nothing;
    ensures result:    \result == n;

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

Listing 4.31: Formal specification of search_n

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. The result ensures clause of behavior has_match indicates that the return value must be in the range [0..n-p]. The match ensures clause expresses that the return value of search_n actually points to an index where b can be found p 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 n of the range a.

4.8.3. Implementation of search_n

Although the specification of search_n strongly resembles that of search, their implementations in the C++ Standard Library are significantly different. The former has a time complexity of $O(n)$, whereas the latter employs an easy, but a non-optimal algorithm needing $O(n \cdot p)$ time, cf. Section 4.7.3.

Our implementation maintains in the variable start the beginning of the most recent consecutive range of values v. The loop invariant not_found states that we didn't find an p-fold repetition of b up to now; if we find one, we terminate the loop, returning start. We handle the boundary cases n < p and p == 0 in explicit else branches. We found this easier when trying to ensure a verification by automatic provers.

```
size_type
search_n(const value_type* a, size_type n, value_type v, size_type p)
{
  if (0u < p) {
    if (p <= n) {
      size_type start = 0u;

      /*@
        loop invariant match:     AllEqual(a, start, i, v);
        loop invariant start:     0 < start ==> a[start-1] != v;
        loop invariant bound:     start <= i + 1 <= start + p;
        loop invariant not_found: !HasConstantSubRange(a, i, v, p);
        loop assigns i, start;
        loop variant n - i;
      */
      for (size_type i = 0u; i < n; ++i) {
        if (a[i] != v) {
          start = i + 1u;
          //@ assert not_found: !HasConstantSubRange(a, i+1, v, p);
        }
        else {
          //@ assert match: a[i] == v;
          //@ assert match: AllEqual(a, start, i+1, v);
          if (p == i + 1u - start) {
            //@ assert bound: start + p == i + 1;
            //@ assert match: AllEqual(a, start, start+p, v);
            //@ assert match: \exists integer k; 0 <= k <= n-p && AllEqual(a, k, k+p
                , v);
            //@ assert match: HasConstantSubRange(a, n, v, p);
            return start;
          }
          else {
            //@ assert bound: i + 1 < start + p;
            continue;
          }
        }

        //@ assert not_found: !HasConstantSubRange(a, i+1, v, p);
      }

      //@ assert not_found: !HasConstantSubRange(a, n, v, p);
      return n;
    }
    else {
      //@ assert not_found: n < p;
      //@ assert not_found: !HasConstantSubRange(a, n, v, p);
      return n;
    }
  }
  else {
    //@ assert bound: p == 0;
    //@ assert match: AllEqual(a, 0, 0, v);
    //@ assert match: HasConstantSubRange(a, n, v, 0);
    return 0u;
  }
}
```

Listing 4.32: Implementation of `search_n`

4.9. The `find_end` algorithm

The `find_end` algorithm in the C++ Standard Library [20, §28.5.6] 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`. The signature now reads:

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

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 < p` (see Figure 4.33). If no such index exists, then `find_end` returns the length `n` of the array `a`. One has to remark the special case `p == 0`. In this case the last position of the empty string is found (the length `n`) and returned.

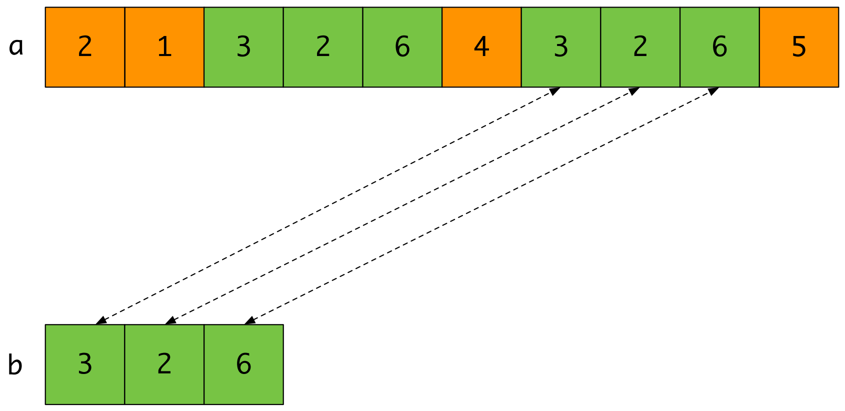


Figure 4.33.: Finding the last occurrence `b[0..p-1]` in `a[0..n-1]`

4.9.1. Formal specification of `find_end`

The ACSL specification of `find_end` is shown in Listing 4.34. Conceptually, the specification of the function `find_end` is very similar to that of `find` in Section 4.2. We therefore use again behaviors to capture the essential aspects of `find_end`. It is quite clear that these behaviors are *complete* and *disjoint*.

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..n-p`. 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 `n` of the range `a`.

```
/*@
  requires valid:    \valid_read(a + (0..n-1));
  requires valid:    \valid_read(b + (0..p-1));
  assigns            \nothing;
  ensures result:    0 <= \result <= n;

  behavior has_match:
    assumes          HasSubRange(a, n, b, p);
    assigns          \nothing;
    ensures bound:   0 <= \result <= n-p;
    ensures result:  EqualRanges{Here,Here}(a + \result, p, b);
    ensures last:    !HasSubRange(a, \result + 1, n, b, p);

  behavior no_match:
    assumes          !HasSubRange(a, n, b, p);
    assigns          \nothing;
    ensures result:  \result == n;

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

Listing 4.34: Formal specification of `find_end`

4.9.2. Implementation of `find_end`

Our implementation of `find_end` is shown in Listing 4.35. Similar to our `search` implementation (Section 4.7), it follows the C++ Standard Library implementation in being easy to understand, but needing an order of magnitude of `n*p` rather than only `n+p` operations.

```
size_type
find_end(const value_type* a, size_type n,
         const value_type* b, size_type p)
{
  size_type r = n;

  if ((0u < p) && (p <= n)) {
    /*@
      loop invariant bound    :  r <= n - p || r == n;
      loop invariant not_found: r == n ==> !HasSubRange(a, p+i-1, b, p);
      loop invariant found:     r < n  ==> EqualRanges{Here,Here}(a+r, p, b);
      loop invariant last:      r < n  ==> !HasSubRange(a, r+1, i+p-1, b, p);
      loop assigns i, r;
      loop variant n - i;
    */
    for (size_type i = 0u; i <= n - p; ++i) {
      if (equal(a + i, p, b)) {
        r = i;
      }
    }
  }

  return r;
}
```

Listing 4.35: Implementation of `find_end`

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

The invariant `bound` states that `r` either still has the value `n` or has a value up to `n-p`. For the former case, invariant `not_found` indicates that no occurrence of `b` has been found. For the latter case, the loop invariant `found` indicates that an occurrence `b[0..p-1]` at `r` has indeed been found. The invariant `last`, on the other hand states that none was found *after* the index `r`.

4.10. The count algorithm

The count algorithm in the C++ Standard Library [20, §28.5.9] counts the frequency of occurrences for a particular element in a sequence. For our purposes we have modified the generic implementation 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.

4.10.1. The logic function Count

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.[15] We therefore start with an axiomatic definition of *logic function* Count that captures the basic intuitive features of counting on an array section. The expression Count(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 the Listings 4.36 and 4.37.[16]

```
/*@
  axiomatic Count
  {
    logic integer
    Count(value_type* a, integer m, integer n, value_type v) =
      n <= m ? 0 : Count(a, m, n-1, v) + (a[n-1] == v ? 1 : 0);

    logic integer
    Count(value_type* a, integer n, value_type v) = Count(a, 0, n, v);

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

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

    lemma Count_Miss:
      \forall value_type *a, v, integer n, m;
        m < n        ==>
        a[n-1] != v  ==>
        Count(a, m, n, v) == Count(a, m, n-1, v);

    lemma Count_Read{K,L}:
      \forall value_type *a, v, integer m, n;
        Unchanged{K,L}(a, m, n)  ==>  Count{K}(a, m, n, v) == Count{L}(a, m, n, v);
```

Listing 4.36: The logic function Count (1)

[15] This statement is not quite true because the ACSL documentation lists numof as one of several *higher order logic constructions* [15, §2.6.7]. However, these *extended quantifiers* are mentioned only as experimental features.
[16] This definition of Count is a generalization of the *logic function* nb_occ of the ACSL specification [15, p. 55].

```
    lemma Count_One:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  Count(a, m, n+1, v) == Count(a, m, n, v) + Count(a, n, n+1, v);

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

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

    lemma Count_Increasing:
      \forall value_type *a, v, integer m, n, p;
        m <= n <= p  ==>  Count(a, m, n, v) <= Count(a, m, p, v);

    lemma Count_Shift:
      \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 4.37: The logic function `Count` (2)

- The ACSL keyword `axiomatic` is used to structure the specification and gather the logic function `Count` and related lemmas. Note that the interval bounds `m` and `n` and the return value for `Count` are of type `integer`.
- The logic functions `Count` is defined explicitly with the recursion. It consist of two checks: for the range emptiness and for the value of the "current" element in the array. The recursion goes down on the range length. We also provide an overloaded version of `Count` that accepts only the length of an array, thus relieving the use the supply the argument $m = 0$ for the case of a complete array.
- Lemma `Count_Empty` covers the cases of empty ranges.
- Lemmas `Count_Hit` and `Count_Miss` reduce counting of a range of length $n - m$ to a range of length $n - m - 1$.
- The logic function `Count` depends only on the set `a[m..n-1]` of memory locations. Lemma `Count_Read` 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 `K` and `L`. We use the predicate `Unchanged` (Listing 7.1 in Section 7.1) to express the premise of the lemma `Count_Read`.

4.10.2. Formal specification of count

Listing 4.38 shows how we use the logic function Count from Listing 4.36 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 4.38: Formal specification of count

4.10.3. Implementation of count

Listing 4.39 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 = 0u;

  /*@
    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 = 0u; i < n; ++i) {
    if (a[i] == val) {
      counted++;
    }
  }

  return counted;
}
```

Listing 4.39: Implementation of count

4.11. The `count2` algorithm

In this section, we specify the `count` algorithm in a different way, namely using the *inductively* defined predicate `CountInd` from Listing 4.40.

```
/*@
  inductive CountInd{L}(value_type *a, integer n, value_type v, integer sum)
  {
    case Nil{L}:
      \forall value_type *a, v, integer n;
        n <= 0  ==>  CountInd{L}(a, n, v, 0);

    case Hit{L}:
      \forall value_type *a, v, integer n, sum;
        0 < n  &&  a[n-1] == v  &&  CountInd{L}(a, n-1, v, sum)  ==>
        CountInd{L}(a, n, v, sum + 1);

    case Miss{L}:
      \forall value_type *a, v, integer n, sum;
        0 < n  &&  a[n-1] != v  &&  CountInd{L}(a, n-1, v, sum)  ==>
        CountInd{L}(a, n, v, sum);
  }
*/
```

Listing 4.40: The inductive predicate `CountInd`

The definition consists of three cases.

- The `Nil` case states for arrays of negative pf zero length, the predicate only holds is `sum` is zero.
- The `Hit` and `Miss` define `CountInd` for arrays `a[0..n-1]` of size `n` referring to the array `a[0..n-2]` and the value `a[n-1]`.

We remark that the cases are very similar to the lemmas `Count_Empty`, `Count_Hit`, `Count_Miss` from Listing 4.36, except we use the additional argument `sum` to refer to the number of counted elements since this is the predicate.

We have intentionally used the scheme $n - 1 \Rightarrow n$ instead of $n \Rightarrow n + 1$. In this particular case, it allows theorem provers to match loop indices with premises without additional hints to prove loop invariants.

4.11.1. Additional lemmas for the inductive predicate

Listing 4.41 complements Listing 4.36 and demonstrates how analogous lemmas could be rewritten for an inductive predicate. These lemmas are not required to prove the `count` function, but we provide them to complete the illustrative example of how inductive predicates could be utilized in the specifications.

The inductive definition is the "completes" definition. This means that a predicate does not hold in every case outside the definition. We state this property explicitly for `CountInd` with Lemma `CountInd_Inverse` from Listing 4.42. Frama-C does not add such axiom in the context, and thus the Lemma is not proved automatically. The reason for not adding such an axiom that it "could confuse first-order theorem provers".[17]

[17]`https://stackoverflow.com/a/32457870`

```
/*@
  axiomatic CountIndImplicit
  {
    lemma CountInd_Empty{L}:
      \forall value_type *a, v, integer n;
       n <= 0  ==>  CountInd(a, n, v, 0);

    lemma CountInd_Hit{L}:
      \forall value_type *a, v, integer n, sum;
        0 < n                        ==>
        a[n-1] == v                  ==>
        CountInd(a, n-1, v, sum)   ==>
        CountInd(a,   n, v, sum+1);

    lemma CountInd_Miss{L}:
      \forall value_type *a, v, integer n, sum;
        0 < n                        ==>
        a[n-1] != v                  ==>
        CountInd(a, n-1, v, sum)   ==>
        CountInd(a,   n, v, sum);

    lemma CountInd_Read{K,L}:
      \forall value_type *a, v, integer n, sum;
        Unchanged{K,L}(a, n)   ==>
        (CountInd{K}(a, n, v, sum)   <==>  CountInd{L}(a, n, v, sum));
  }
*/
```

Listing 4.41: Implicit definition of CountInd

Listing 4.42 also contains Lemma CountInd_NonNegative which states that the lower bound for the number of the counted elements is zero. The relation between the inductive definition CountInd and the explicit definition of Count is expressed by Lemma CountInd_Count from Listing 4.42.

```
/*@
  axiomatic CountIndLemmas
  {
    lemma CountInd_Inverse:
      \forall value_type *a, v, integer n, sum;
        CountInd(a, n, v, sum)   ==>
          (n <= 0 && sum == 0) ||
          (0 < n && a[n-1] != v && CountInd(a, n-1, v, sum)) ||
          (0 < n && a[n-1] == v && CountInd(a, n-1, v, sum-1));

    lemma CountInd_NonNegative{L}:
      \forall value_type *a, v, integer n, sum;
        CountInd(a, n, v, sum)   ==>  0 <= sum;

    lemma CountInd_Count{L}:
      \forall value_type *a, v, integer n;
        CountInd(a, n, v, Count(a, n, v));
  }
*/
```

Listing 4.42: More lemmas for CountInd

4.11.2. Specification of count

Listing 4.43 shows the use of the inductive predicate CountInd from Listing 4.40 to specify count.

```
/*@
  requires valid: \valid_read(a + (0..n-1));
  assigns         \nothing;
  ensures bound:  0 <= \result <= n;
  ensures count:  CountInd(a, n, val, \result);
*/
size_type
count2(const value_type* a, size_type n, value_type val);
```

Listing 4.43: The count contract reconsidered

4.11.3. Implementation of count

The only difference of Listing 4.44 to Listing 4.39 is that we have to supply the value counted as an argument of the CountInd predicate.

```
size_type
count2(const value_type* a, size_type n, value_type val)
{
  size_type counted = 0u;

  /*@
    loop invariant bound: 0 <= i <= n;
    loop invariant bound: 0 <= counted <= i;
    loop invariant count: CountInd(a, i, val, counted);
    loop assigns i, counted;
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    if (a[i] == val) {
      counted++;
      //@ assert count: CountInd(a, i+1, val, counted);
    }
  }

  return counted;
}
```

Listing 4.44: The count implementation reconsidered

5. Maximum and minimum algorithms

In this chapter we discuss the formal specification of algorithms in the C++ Standard Library [20, §28.7.8] that compute the maximum or minimum values of their arguments. As the algorithms in Chapter 4, 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.

- In Section 5.1 we discuss some properties of relations operators.
- In Section 5.2 we introduce various predicates that describe basic order properties for arrays whose elements are of `value_type`.
- `clamp`, which is discussed in Section 5.3, is a very simple algorithms that "clamps" (or "clips") a value between a pair of boundary values.
- `max_element` returns an index to a maximum element in a range. Similar to `find` it also returns the smallest of all possible indices. This algorithm is discussed in Section 5.4. In Section 5.5, we introduce an alternative specification `max_element2` which relies on user-defined predicates.
- `max_seq` (Section 5.6 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` can be used to find the smallest element in an array (Section 5.8).
- `minmax_element` is used to find simultaneously the smallest and largest element in a given range (Section 5.9). This algorithms relies on the auxiliary function `make_pair` (Section 5.7).

First, however, we discuss in Section 5.1 general properties that must be satisfied by the relational operators.

5.1. A note on relational operators

Note that in order to compare values, algorithms in the C++ Standard Library [20, §28.7.8] usually rely solely on the *less than* operator < or special function objects. To be precise, the operator < must be a *partial order*,[18] which means that the following rules must 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 to check that the operator < of our `value_type`[19] satisfies these properties you can formulate lemmas in ACSL and verify them with Frama-C (see Listing 5.1).

[18]See `http://en.wikipedia.org/wiki/Partially_ordered_set`
[19]See Section 2.3

```
/*@
  axiomatic Less
  {
    lemma Less_Irreflexivity:
      \forall value_type a; !(a < a);

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

    lemma Less_Transitivity:
      \forall value_type a, b, c; (a < b) && (b < c)  ==>  (a < c);

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

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

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

Listing 5.1: 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 5.1 therefor also contains lemmas on representing the operator >, <=, and >= through operator <.

5.2. Predicates for bounds and extrema of arrays

We define in Listing 5.2 the predicates `MaxElement` and `MinElement` that we will use for the specification of various algorithms. We will discuss these predicates in more detail in §5.5 and §5.8.

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

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

Listing 5.2: Predicates for extremal values of arrays

The aforementioned predicates rely on the predicates LowerBound and UpperBound These predicates the related predicates StrictUpperBound and StrictLowerBound are defined Listing 5.3.

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

    predicate
    LowerBound{L}(value_type* a, integer n, value_type v) =
      LowerBound{L}(a, 0, n, v);

    predicate
    StrictLowerBound{L}(value_type* a, integer m, integer n, value_type v) =
      \forall integer i; m <= i < n  ==>  v < a[i];

    predicate
    StrictLowerBound{L}(value_type* a, integer n, value_type v) =
      StrictLowerBound{L}(a, 0, n, v);

    predicate
    UpperBound{L}(value_type* a, integer m, integer n, value_type v) =
      \forall integer i; m <= i < n  ==>  a[i] <= v;

    predicate
    UpperBound{L}(value_type* a, integer n, value_type v) =
      UpperBound{L}(a, 0, n, v);

    predicate
    StrictUpperBound{L}(value_type* a, integer m, integer n, value_type v) =
      \forall integer i; m <= i < n  ==>  a[i] < v;

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

Listing 5.3: Predicates for lower and upper bounds of an array

These predicates concisely express the comparison of the elements in an array (segment) with a given value. We will heavily rely on these predicates both in this chapter and in Chapter 6.

5.3. The clamp algorithm

The clamp algorithm in the C++ Standard Library [20, §28.7.9] "clamps" a value between a pair of boundary values. The signature of our version of clamp reads:

```
value_type
clamp(value_type v, value_type lower, value_type upper);
```

The function clamp returns v if the value is greater than lower and smaller than upper. Otherwise, if v is smaller than lower, the return value is lower instead. Similar to that if v is greater than upper, upper will be the return value of clamp.

5.3.1. Formal specification of clamp

The ACSL specification of clamp is shown in Listing 5.4. Note that we state in the precondition proper that lower must less or equal to upper.

```
/*@
  requires bound:   lower < upper;
  assigns           \nothing;
  ensures bound:    lower <= \result <= upper;

  behavior lower_bound:
    assumes         v < lower;
    assigns         \nothing;
    ensures result: \result == lower;

  behavior between:
    assumes         lower <= v <= upper;
    assigns         \nothing;
    ensures result: \result == v;

  behavior upper_bound:
    assumes         upper < v;
    assigns         \nothing;
    ensures result: \result == upper;

  complete behaviors;
  disjoint behaviors;
*/
value_type
clamp(value_type v, value_type lower, value_type upper);
```

Listing 5.4: Formal specification of clamp

5.3.2. Implementation of clamp

The Listing 5.5 shows an implementation of `clamp`.

```
value_type
clamp(value_type v, value_type lower, value_type upper)
{
  return (v < lower) ? lower : (upper < v) ? upper : v;
}
```

Listing 5.5: Implementation of `clamp`

5.4. The max_element algorithm

The max_element algorithm in the C++ Standard Library [20, §28.7.8] 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-1]. 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.

5.4.1. Formal specification of max_element

A formal specification of max_element in ACSL is shown in Listing 5.6. Note that we have subdivided the specification of max_element into the 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 ensures clause max of behavior not_empty indicates that the returned valid index k refers to a maximum value of the array. The postcondition first expresses that k is indeed the *first* occurrence of a maximum value in the array.

```
/*@
  requires valid:   \valid_read(a + (0..n-1));
  assigns           \nothing;
  ensures  result:  0 <= \result <= n;

  behavior empty:
    assumes          n == 0;
    assigns          \nothing;
    ensures result:  \result == 0;

  behavior not_empty:
    assumes          0 < n;
    assigns          \nothing;
    ensures result:  0 <= \result < n;
    ensures upper:   \forall integer i; 0 <= i < n       ==> a[i] <= a[\result];
    ensures first:   \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 5.6: Formal specification of max_element

5.4.2. Implementation of `max_element`

Listing 5.7 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 (0u < n) {
    size_type max = 0u;

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

    return max;
  }

  return n;
}
```

Listing 5.7: Implementation of `max_element`

Loop invariant `max` is needed to prove postcondition `result` of behavior `not_empty` in Listing 5.6. Using loop invariant `upper` we prove postcondition `upper` of behavior `not_empty` in Listing 5.6. Finally, postcondition `first` of this behavior can be proved with loop invariant `first`.

5.5. 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 the predicate `UpperBound` from Listing 5.3 which basically expresses that a given value is greater or equal than all elements of a given array. Closely related to the predicate `UpperBound` is the predicate `StrictUpperBound` that is also defined in Listing 5.3.

We also employ the predicate `MaxElement` from Listing 5.2. This predicate states that the element at a given index `max` is an *upper bound* of the sequence `a[0..n-1]`, and, by construction, a member of that sequence.

5.5.1. Formal specification of `max_element`

The new formal specification of `max_element` in ACSL is shown in Listing 5.8. Note that we also use the predicate `StrictUpperBound` from Listing 5.3 in order to express that `max_element` returns the *first* maximum position in `a[0..n-1]`.

```
/*@
  requires valid:    \valid_read(a + (0..n-1));
  assigns            \nothing;
  ensures  result:   0 <= \result <= n;

  behavior empty:
    assumes          n == 0;
    assigns          \nothing;
    ensures result:  \result == 0;

  behavior not_empty:
    assumes          0 < n;
    assigns          \nothing;
    ensures result:  0 <= \result < n;
    ensures max:     MaxElement(a, n, \result);
    ensures first:   StrictUpperBound(a, \result, a[\result]);

  complete behaviors;
  disjoint behaviors;
*/
size_type
max_element2(const value_type* a, size_type n);
```

Listing 5.8: Formal specification of `max_element`

5.5.2. Implementation of max_element

Listing 5.9 shows implementation of max_element with only the loop invariants changed.

```
size_type
max_element2(const value_type* a, size_type n)
{
  if (0u < n) {
    size_type max = 0u;

    /*@
      loop invariant bound:   0 <= i <= n;
      loop invariant max:     0 <= max < n;
      loop invariant upper:   UpperBound(a, i, a[max]);
      loop invariant first:   StrictUpperBound(a, max, a[max]);
      loop assigns max, i;
      loop variant n-i;
    */
    for (size_type i = 0u; i < n; i++) {
      if (a[max] < a[i]) {
        max = i;
      }
    }

    return max;
  }

  return n;
}
```

Listing 5.9: Implementation of max_element

5.6. The max_seq algorithm

In this section we consider the function max_seq (see Chapter 3, [14]) which is very similar to the function max_element of Section 5.4. 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 5.11). Moreover, using only the formal specification of max_element in Listing 5.8 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 5.8 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.

5.6.1. Formal specification of max_seq

A formal specification of max_seq in ACSL is shown in Listing 5.10.

```
/*@
  requires    0 < n;
  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 5.10: Formal specification of max_seq

Using the first requires-clause we express that max_seq needs a *non-empty* range as input. Our postconditions formalize that max_seq indeed returns the maximum value of the range.

5.6.2. Implementation of max_seq

Listing 5.11 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_element2(p, n)];
}
```

Listing 5.11: Implementation of max_seq

5.7. The auxiliary function make_pair

In order to be able to specify functions that work on pairs of indices we introduce here the type size_type_pair (Listing 5.12).

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

typedef struct size_type_pair size_type_pair;
```

Listing 5.12: The type size_type_pair

We will also use the auxiliary function make_pair which turns two indices first and second into an object of size_type_pair. The specification and implementation of make_pair is shown in Listing 5.13.

```
/*@
    assigns         \nothing;
    ensures result: \result.first  == first;
    ensures result: \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 5.13: The function make_pair

5.8. The `min_element` algorithm

The `min_element` algorithm in the C++ Standard Library [20, §28.7.8] 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 `a[i]` is minimal among the values `a[0]`, ..., `a[n-1]`, and `i` is the first position with that property. The return value of `min_element` is `n` if and only if `n == 0`.

We use from Listing 5.3 the predicate `LowerBound` that basically expresses that a given value is less or equal than all elements of a given array (section). Closely related to the predicate `LowerBound` is the predicate `StrictLowerBound` that is also defined in Listing 5.3. We also use predicate `MinElement` in Listing 5.2. Thus predicate states that the element at a given index `min` is a *lower bound* of the sequence `a[0..n-1]`, and, by construction, a member of that sequence.

5.8.1. Formal specification of `min_element`

The ACSL specification of `min_element` is shown in Listing 5.14. Note that we also use the predicate `StrictLowerBound` from Listing 5.3 in order to express that `min_element` returns the *first* minimum position in `a[0..n-1]`.

```
/*@
  requires valid:   \valid_read(a + (0..n-1));
  assigns           \nothing;
  ensures  result:  0 <= \result <= n;

  behavior empty:
    assumes         n == 0;
    assigns         \nothing;
    ensures result: \result == 0;

  behavior not_empty:
    assumes         0 < n;
    assigns         \nothing;
    ensures result: 0 <= \result < n;
    ensures min:    MinElement(a, n, \result);
    ensures first:  StrictLowerBound(a, \result, a[\result]);

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

Listing 5.14: Formal specification of `min_element`

5.8.2. Implementation of min_element

Listing 5.15 shows the implementation of min_element with loop invariants where we also employ the predicates LowerBound and StrictLowerBound.

```
size_type
min_element(const value_type* a, size_type n)
{
  if (0u < n) {
    size_type min = 0u;

    /*@
      loop invariant bound:  0 <= i   <= n;
      loop invariant min:    0 <= min <  n;
      loop invariant lower:  LowerBound(a, i, a[min]);
      loop invariant first:  StrictLowerBound(a, min, a[min]);
      loop assigns min, i;
      loop variant n-i;
    */
    for (size_type i = 0u; i < n; i++) {
      if (a[i] < a[min]) {
        min = i;
      }
    }

    return min;
  }

  return n;
}
```

Listing 5.15: Implementation of min_element

5.9. The `minmax_element` algorithm

The `minmax_element` algorithm in the C++ Standard Library [20, §28.7.8] searches the minimum and the maximum in a sequence. The signature of our version of `min_element` reads:

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

Note that `minmax_element` returns a *pair* of indices (see Section 5.7). This pair contains the *first* position where the minimum occurs in the sequence `a[0..n-1]` and the *last* position where maximum occurs.

The properties of the index for the minimum value are the same as the properties for the `min_element` in Section 5.8. In opposition to that, the properties of the index that marks the maximum element, are slightly altered to the properties of `max_element` in Section 5.4. The `max_element` algorithm returns the position of the *first* occurrence of the maximum element if it occurs multiple times in the sequence. The `minmax_element` algorithm returns the position of the last occurrence of the maximum element.

5.9.1. Formal specification of `minmax_element`

The ACSL specification of `minmax_element` is shown in Listing 5.16. Note that we use the predicates `StrictLowerBound` and `StrictUpperBound` from Listing 5.3 in order to express that the algorithm returns the positions of both the *first minimum* and the *last maximum*. We also use the predicates `MinElement` and `MaxElement` from Listing 5.2. Thus reflects of course the use of this predicates for the algorithms `min_element` and `max_element`.

```
/*@
  requires valid:    \valid_read(a + (0..n-1));
  assigns            \nothing;
  ensures result:    0 <= \result.first  <= n;
  ensures result:    0 <= \result.second <= n;

  behavior empty:
    assumes          0 == n;
    assigns          \nothing;
    ensures result:  \result.first == 0;
    ensures result:  \result.second == 0;

  behavior not_empty:
    assumes          0 < n;
    assigns          \nothing;
    ensures result:  0 <= \result.first < n;
    ensures result:  0 <= \result.second < n;

    ensures min:     MinElement(a, n, \result.first);
    ensures first:   StrictLowerBound(a, \result.first, a[\result.first]);
    ensures max:     MaxElement(a, n, \result.second);
    ensures last:    StrictUpperBound(a, \result.second+1, n, a[\result.second]);
*/
size_type_pair
minmax_element(const value_type* a, size_type n);
```

Listing 5.16: Formal specification of `minmax_element`

The specification is similar to the specifications of min_element and max_element. The only difference lies in the postcondition last. Here the postcondition states that after the position of the maximum element there is no value greater or equal the maximum element. This differs from the specification of max_element, where the first occurrence of the maximum value has to be returned.

5.9.2. Implementation of **minmax_element**

Listing 5.17shows an implementation of minmax_element. It uses the auxiliary function make_pair from Section 5.7 to construct a pair of indices. We will focus on the loop invariant last, because it is the only loop invariant differs from the implementations of min_element and max_element.

```
size_type_pair
minmax_element(const value_type* a, size_type n)
{
  if (0u < n) {
    size_type min = 0u;
    size_type max = 0u;

    /*@
      loop invariant bound: 0 <= i    <= n;
      loop invariant min:   0 <= min  <  n;
      loop invariant max:   0 <= max  <  n;
      loop invariant lower: LowerBound(a, i, a[min]);
      loop invariant upper: UpperBound(a, i, a[max]);
      loop invariant first: StrictLowerBound(a, min, a[min]);
      loop invariant last:  StrictUpperBound(a, max+1, i, a[max]);
      loop assigns min, max, i;

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

      if (a[i] < a[min]) {
        min = i;
      }
    }

    return make_pair(min, max);
  }

  return make_pair(n, n);
}
```

Listing 5.17: Implementation of minmax_element

As already mentioned we had to alter the range for the predicate StrictUpperBound to fit into the property of returning the last maximum position that occurred.

6. Binary search algorithms

In this chapter, we consider the four *binary search* algorithms of the C++ Standard Library [20, §28.7.3], namely

- `lower_bound` in Section 6.1
- `upper_bound` in Section 6.2
- two variants for the implementation of `equal_range` which we refer to as `equal_range` and `equal_range2` in Sections 6.3
- two variants for the formal specification of `binary_search` which we refer to as `binary_search` and `binary_search2` in Section 6.4

All binary search algorithms require that their input array is arranged in increasing order. There are two versions of predicate `Increasing` in Listing 6.1. The first one defines when a section of an array is in increasing order. The second version uses the first one to express that the whole array is in increasing order. There is also the overloaded predicate `WeaklyIncreasing` that we will user for the verification of other algorithms.

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

    predicate
    Increasing{L}(value_type* a, integer n) = Increasing{L}(a, 0, n);

    predicate
    WeaklyIncreasing{L}(value_type* a, integer m, integer n) =
      \forall integer i; m <= i < n-1  ==>  a[i] <= a[i+1];

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

Listing 6.1: The predicate `Increasing`

As in the case of the of maximum/minimum algorithms from Chapter 5 the binary search algorithms primarily use the less-than operator < (and the derived operators <=, > and >=) to determine whether a particular value is contained in an increasing range. Thus, different to the `find` algorithm in Section 4.1, 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 re-use the following predicates `LowerBound`, `StrictLowerBound`, `UpperBound`, and `StrictUpperBound` from Listing 5.3

6.1. The `lower_bound` algorithm

The `lower_bound` algorithm is one of the four binary search algorithms of the C++ Standard Library [20, §28.7.3.1]. For our purposes we have modified the generic implementation 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 in increasing order. The index `lb`, that `lower_bound` returns satisfies the inequality

$$0 \leq \texttt{lb} \leq n \tag{6.1}$$

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

$$0 \leq k < \texttt{lb} \quad \Longrightarrow \quad a[k] < \texttt{val} \tag{6.2}$$
$$\texttt{lb} \leq k < n \quad \Longrightarrow \quad \texttt{val} \leq a[k] \tag{6.3}$$

Conditions (6.2) and (6.3) 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 6.2 an increasingly ordered 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.

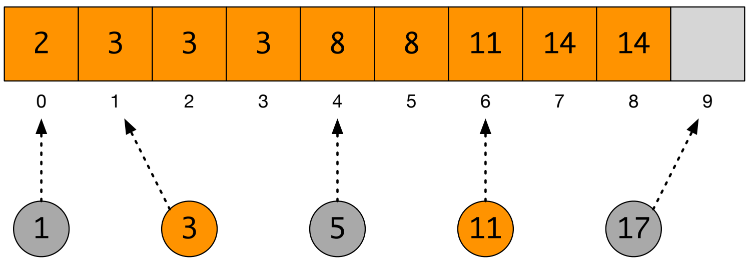


Figure 6.2.: Some examples for `lower_bound`

Figure 6.2 also clarifies that care must be taken when interpreting the return value of `lower_bound`. An important difference to the algorithms in Chapter 4 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.

6.1.1. Formal specification of `lower_bound`

The ACSL specification of `lower_bound` is shown in Listing 6.3. The preconditions `increasing` expresses that the array values need to be in increasing order. The postconditions reflect the conditions listed above and can be expressed using the predicates `LowerBound` and `StrictUpperBound` from Listing 5.3, namely,

- Condition (6.1) becomes postcondition `result`

- Condition (6.2) becomes postcondition `left`
- Condition (6.3) becomes postcondition `right`

```
/*@
  requires valid:      \valid_read(a + (0..n-1));
  requires increasing: Increasing(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 6.3: Formal specification of `lower_bound`

6.1.2. Implementation of `lower_bound`

Our implementation of `lower_bound` is shown in Listing 6.4. 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 `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  = 0u;
  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) / 2u;

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

  return left;
}
```

Listing 6.4: Implementation of `lower_bound`

6.2. The `upper_bound` algorithm

The `upper_bound` algorithm of the C++ Standard Library [20, §28.7.3.2] is a variant of binary search and closely related to `lower_bound` of Section 6.1. 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 in increasing order. The index `ub` returned by `upper_bound` satisfies the inequality

$$0 \leq \texttt{ub} \leq n \tag{6.4}$$

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

$$\begin{aligned} 0 \leq k < \texttt{ub} \quad &\Longrightarrow \quad a[k] \leq \texttt{val} && (6.5)\\ \texttt{ub} \leq k < n \quad &\Longrightarrow \quad \texttt{val} < a[k] && (6.6) \end{aligned}$$

Conditions (6.5) and (6.6) 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 6.5 is a variant of Figure 6.2 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 6.2, only the arrows from values that *are present* in the array change their target index.

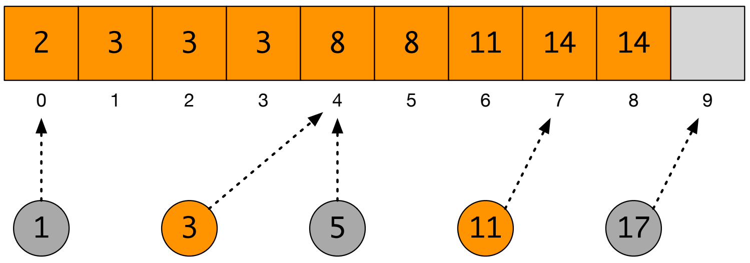


Figure 6.5.: Some examples for `upper_bound`

6.2.1. Formal specification of `upper_bound`

The ACSL specification of `upper_bound` is shown in Listing 6.6. The specification is quite similar to the specification of `lower_bound` (see Listing 6.3). The precondition `increasing` expresses that the array values need to be in increasing order. The postconditions reflect the conditions listed above and can be expressed using predicates `UpperBound` and `StrictLowerBound` from Listing 5.3, namely,

```
/*@
  requires valid:      \valid_read(a + (0..n-1));
  requires increasing: Increasing(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 6.6: Formal specification of `upper_bound`

- Condition (6.4) becomes postcondition `result`
- Condition (6.5) becomes postcondition `left`
- Condition (6.6) becomes postcondition `right`

6.2.2. Implementation of `upper_bound`

Our implementation of `upper_bound` is shown in Listing 6.7.

```
size_type
upper_bound(const value_type* a, size_type n, value_type val)
{
  size_type left  = 0u;
  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) / 2u;

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

  return right;
}
```

Listing 6.7: Implementation of `upper_bound`

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`.

6.3. The equal_range algorithm

The `equal_range` algorithm is one of the four binary search algorithms of the C++ Standard Library [20, §28.7.3.3]. As with the other binary search algorithms `equal_range` requires that its input array is in increasing order. The specification of `equal_range` states that it *combines* the results of the algorithms `lower_bound` (Section 6.1) and `upper_bound` (Section 6.2).

For our purposes we have modified `equal_range` 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 5.12. Thus, the signature of `equal_range` now reads:

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

Figure 6.8 combines Figure 6.2 with Figure 6.5 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.

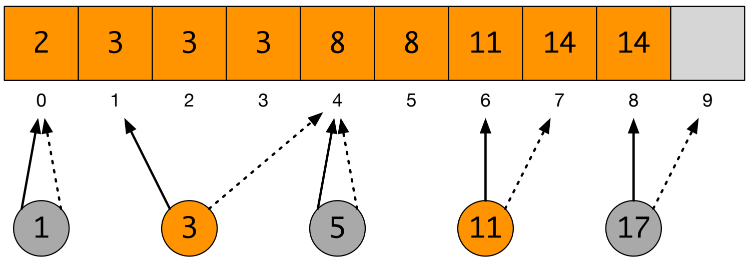


Figure 6.8.: Some examples for `equal_range`

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 6.10). The second, more elaborate, implementation follows the original STL code by attempting to minimize duplicate computations (see Listing 6.11).

Let (lb, ub) be the return value `equal_range`, then the conditions (6.1)–(6.6) can be merged into the inequality

$$0 \le \mathtt{lb} \le \mathtt{ub} \le n \tag{6.7}$$

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

$$0 \le k < \mathtt{lb} \implies a[k] < \mathtt{val} \tag{6.8}$$

$$\mathtt{lb} \le k < \mathtt{ub} \implies a[k] = \mathtt{val} \tag{6.9}$$

$$\mathtt{ub} \le k < n \implies a[k] > \mathtt{val} \tag{6.10}$$

Here are some justifications for these conditions.

- Conditions (6.8) and (6.10) are just the Conditions (6.2) and (6.6), respectively.
- The Inequality (6.7) follows from the Inequalities (6.1) and (6.4) and the following considerations: If ub were less than lb, then according to (6.8) we would have $a[\mathtt{ub}] < \mathtt{val}$. One the other hand, we know from (6.10) that opposite inequality $\mathtt{val} < a[\mathtt{ub}]$ holds. Therefore, we have $\mathtt{lb} \le \mathtt{ub}$.

- Condition (6.9) follows from the combination of (6.3) and (6.5) and the fact that ≤ is a total order on the integers.

6.3.1. Formal specification of `equal_range`

The ACSL specification of `equal_range` is shown in Listing 6.9.

```
/*@
  requires valid:      \valid_read(a + (0..n-1));
  requires increasing: Increasing(a, n);
  assigns              \nothing;
  ensures result:      0 <= \result.first <= \result.second <= n;
  ensures left:        StrictUpperBound(a, 0, \result.first, val);
  ensures middle:      AllEqual(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 6.9: Formal specification of `equal_range`

The ACSL specification of `equal_range` is shown in Listing 6.9. The precondition `increasing` expresses that the array values need to be in increasing order.

The postconditions reflect the conditions listed above and can be expressed using the already introduced predicates `AllEqual` (Listing 4.29), and `StrictUpperBound` and `StrictLowerBound` (Listing 5.3), namely,

- Condition (6.7) becomes postcondition `result`
- Condition (6.8) becomes postcondition `left`
- Condition (6.9) becomes postcondition `middle`
- Condition (6.10) becomes postcondition `right`

6.3.2. First implementation of `equal_range`

Our first implementation of `equal_range` is shown in Listing 6.10. We just 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 6.10: 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 6.6). The benefit of this reformulation is that both the assertion aux and the postcondition first <= second can now be verified automatically.

6.3.3. Second implementation of equal_range

The first implementation of equal_range does more work than needed. In Listing 6.11 we show that it is possible to perform as much range reduction as possible before calling upper_bound and lower_bound on the reduced ranges.

```
size_type_pair
equal_range2(const value_type* a, size_type n, value_type val)
{
  size_type first  = 0u;
  size_type middle = 0u;
  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) / 2u;

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

  if (first < last) {
    //@ assert increasing: Increasing(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 increasing: Increasing(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 6.11: 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 6.12.

```
/*@
  axiomatic ArrayBoundsShift
  {
    lemma IncreasingShift{L}:
      \forall value_type *a, integer l, r;
        0 <= l <= r             ==>
        Increasing{L}(a, l, r)  ==>
        Increasing{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 6.12: Some lemmas to support the verification of `equal_range`

The `increasing` properties

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

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

A three-argument form of the `Increasing` predicate from Listing 6.1 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` and `constant` 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.

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.

6.4. The `binary_search` algorithm

The `binary_search` algorithm is one of the four binary search algorithms of the C++ Standard Library [20, §28.7.3.4]. For our purposes we have modified the generic implementation 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 in increasing order. It will return **`true`** if there exists an index `i` in `a` such that `a[i] == val` holds.[20]

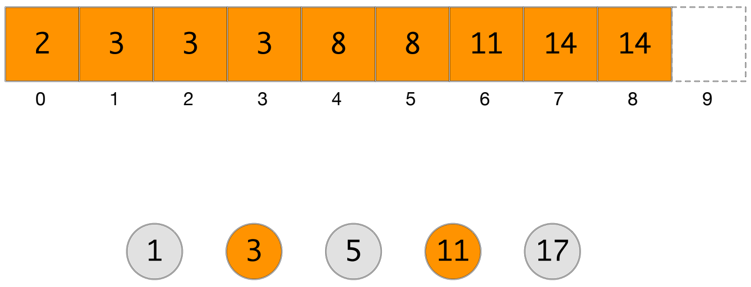


Figure 6.13.: Some examples for `binary_search`

In Figure 6.13 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.

6.4.1. Formal specification of `binary_search`

The ACSL specification of `binary_search` is shown in Listing 6.14.

[20]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 2.3) this means that `val` equals `a[i]`.

```
/*@
  requires valid:      \valid_read(a + (0..n-1));
  requires increasing: Increasing(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 6.14: Formal specification of `binary_search`

Note that instead of the somewhat lengthy existential quantification in Listing 6.14 we can use our previously introduced predicate `SomeEqual` (Listing 4.4) in order to achieve the more concise formal specification in Listing 6.15.

```
/*@
  requires valid:      \valid_read(a + (0..n-1));
  requires increasing: Increasing(a, n);
  assigns              \nothing;
  ensures  result:     \result <==> SomeEqual(a, n, val);
*/
bool
binary_search2(const value_type* a, size_type n, value_type val);
```

Listing 6.15: Formal specification of `binary_search` using the predicate `SomeEqual`

It is interesting to compare this specification with that of `find` shown in Listing 4.5. 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.

6.4.2. Implementation of `binary_search`

Our implementation of `binary_search` is shown in Listing 6.16.

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

Listing 6.16: Implementation of `binary_search`

The function `binary_search` first calls `lower_bound` from Section 6.1. Remember that if the latter returns an index `0 <= i < n` then we can be sure that `val <= a[i]` holds.

Part III.

Mutating and numeric algorithms

7. Mutating algorithms

Let us now turn our attention to another class of algorithms, viz. *mutating* algorithms of the C++ Standard Library [20, §28.6], 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 [15, §2.3.2]. Below are the algorithms we will discuss in this chapter.

- In order to allow for a finer control of which parts of an array, we introduce in Section 7.1 the auxiliary predicate `Unchanged`.
- `fill` in Section 7.2 initializes each element of an array by a given fixed value.
- `swap` in Section 7.3 exchanges two values.
- `swap_ranges` in Section 7.4 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 7.5 copies a source array to a destination array.
- `copy_backward` in Section 7.6 also copies a source array to a destination array. This version, however, uses another separation condition than `copy`.
- `reverse_copy` and `reverse` in Sections 7.7 and 7.8, 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 7.9 rotates a source array by `m` positions and copies the results to a destination array.
- `rotate` in Section 7.10 rotates *in place* a source array by `m` positions.
- `replace_copy` and `replace` in Sections 7.11 and 7.12, 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` and `remove` in Sections 7.13–7.16 *filter* all occurrences of a given value from an array. Whereas `remove_copy` copies the result to a separate array, the `remove` algorithm works in place. Note that we provide altogether three versions of how to specify `remove_copy`. This shall help the reader to understand that finding appropriate contracts is an iterative process and that it is usually a good idea to *not* strive for a "complete" contract right from the beginning.

  Note that in Chapter 8 we give an even more detailed presentation of the related algorithm `unique_copy`.
- `shuffle` in Section 7.17 randomly reorders the elements of an array thereby relying on the simple random number generator `random_number` in Section 7.18.

7.1. 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 7.1 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.

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

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

Listing 7.1: The predicate Unchanged

We also provide in Listing 7.2 a few lemmas for Unchanged that we need for the verification of various algorithms.

```
/*@
  axiomatic UnchangedLemmas
  {
    lemma Unchanged_Shrink{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);

    lemma Unchanged_Extend{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);

    lemma Unchanged_Shift{K,L}:
      \forall value_type *a, integer p, q, r;
        Unchanged{K,L}(a+p, q, r)   ==>  Unchanged{K,L}(a, p+q, p+r);

    lemma Unchanged_Transitive{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 7.2: Some lemmas for predicate Unchanged

- Lemma `Unchanged_Shrink` states that if the range `a[m..n-1]` does not change when going from state `K` to state `L`, then `a[p..q-1]` does not change either, provided the latter is a subrange of the former, i.e. provided $0 \leq m \leq p \leq q \leq n$ holds.
- Lemma `Unchanged_Extend` expresses the simple fact that "unchangedness" is an inductive property.
- Lemma `Unchanged_Shift` states how `Unchanged` behaves under pointer additions.
- Lemma `Unchanged_Transitive` expresses the transitivity of `Unchanged` with respect to program states.

7.2. The `fill` algorithm

The `fill` algorithm in the C++ Standard Library [20, §28.6.6] initializes general sequences with a particular value. For our purposes we have modified the generic implementation to that of an array of type `value_type`. The signature now reads:

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

7.2.1. Formal specification of `fill`

Listing 7.3 shows the formal specification of `fill` in ACSL. We can express the postcondition of `fill` simply by using the overloaded predicate `AllEqual` from Listing 4.29.

```
/*@
  requires valid:   \valid(a + (0..n-1));
  assigns           a[0..n-1];
  ensures constant: AllEqual(a, n, val);
*/
void
fill(value_type* a, size_type n, value_type val);
```

Listing 7.3: 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 [15, §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 4.

7.2.2. Implementation of `fill`

Listing 7.4 shows an implementation of `fill`.

```
void
fill(value_type* a, size_type n,  value_type val)
{
  /*@
    loop invariant bound:    0 <= i <= n;
    loop invariant constant: AllEqual(a, i, val);
    loop assigns i, a[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    a[i] = val;
  }
}
```

Listing 7.4: 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 `AllEqual` from Listing 4.29.

7.3. The swap algorithm

The swap algorithm [20, §28.6.3] in the C++ Standard Library exchanges the contents of two variables. Similarly, the iter_swap algorithm [20, §28.6.3] 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.

7.3.1. Formal specification of swap

The ACSL specification for the swap function is shown in Listing 7.5. The preconditions are formalized by the requires-clauses which state that both pointer arguments of the swap function must be dereferenceable.

```
/*@
  requires valid:    \valid(p);
  requires valid:    \valid(q);
  assigns            *p;
  assigns            *q;
  ensures  exchange: *p == \old(*q);
  ensures  exchange: *q == \old(*p);
*/
void
swap(value_type* p, value_type* q);
```

Listing 7.5: 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.

7.3.2. Implementation of swap

Listing 7.6 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 7.6: Implementation of swap

7.4. The swap_ranges algorithm

The swap_ranges algorithm in the C++ Standard Library [20, §28.6.3] 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.

7.4.1. Formal specification of swap_ranges

Listing 7.7 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{Old,Here}(a, n, b);
  ensures equal:   EqualRanges{Old,Here}(b, n, a);
*/
void
swap_ranges(value_type* a, size_type n, value_type* b);
```

Listing 7.7: 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 clause sep 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 4.19) together with the label Old and Here to express the postcondition of swap_ranges. In our specification in Listing 7.7, 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.

7.4.2. Implementation of `swap_ranges`

Listing 7.8 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{Pre,Here}(a, i, b);
    loop invariant equal:  EqualRanges{Pre,Here}(b, i, a);

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

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

Listing 7.8: Implementation of `swap_ranges`

For the postcondition of the specification in Listing 7.7 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 7.1 is used to express this property.

7.5. The copy algorithm

The copy algorithm in the C++ Standard Library [20, §28.6.1] implements a duplication algorithm for general sequences. For our purposes we have modified the generic implementation 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 7.9.

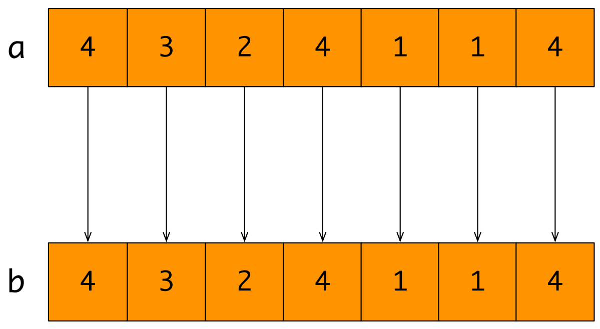


Figure 7.9.: Effects of copy

7.5.1. Formal specification of copy

Figure 7.9 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 7.10.

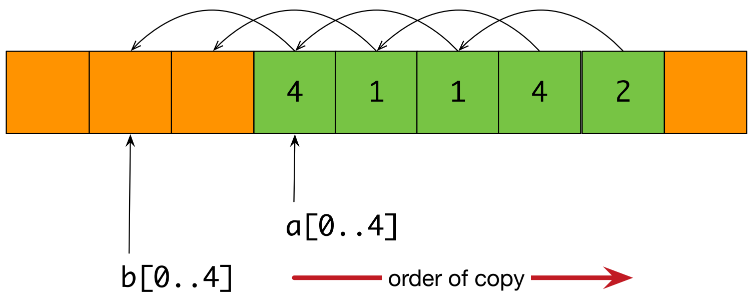


Figure 7.10.: Possible overlap of copy ranges

The ACSL specification of copy is shown in Listing 7.11. 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.

```
/*@
  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 7.11: Formal specification of copy

Again, we can use the EqualRanges predicate from Section 4.6 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.

7.5.2. Implementation of copy

Listing 7.12 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 unchanged: Unchanged{Pre,Here}(a, i, n);
    loop assigns   i, b[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    b[i] = a[i];
  }
}
```

Listing 7.12: 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.[21]

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.

[21]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.

7.6. The `copy_backward` algorithm

The `copy_backward` algorithm in the C++ Standard Library [20, §28.6.1] implements another duplication algorithm for general sequences. For our purposes we have modified the generic implementation 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 7.11.

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.[22] 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 7.13 gives an example where `copy_backward`, but *not* `copy`, can be applied.

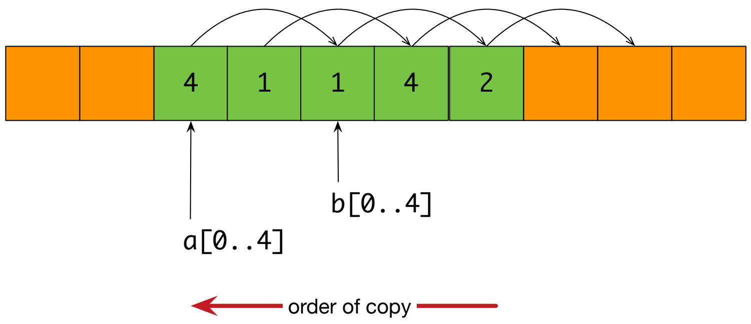


Figure 7.13.: Possible overlap of `copy_backward` ranges

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.

7.6.1. Formal specification of `copy_backward`

The ACSL specification of `copy_backward` is shown in Listing 7.14. 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.

[22]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.

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

Listing 7.14: 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 4.6 to express that the array `a` equals `b` after `copy_backward` has been called.

7.6.2. Implementation of `copy_backward`

Listing 7.15 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 unchanged: Unchanged{Pre,Here}(a, 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 7.15: 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`).

7.7. The reverse_copy algorithm

The reverse_copy algorithm of the C++ Standard Library [20, §28.6.10] 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);
```

Informally, reverse_copy copies the elements from the array a into array b such that the copy is a reverse of the original array.

```
/*@
  axiomatic Reverse
  {
    predicate
    Reverse{K,L}(value_type* a, integer n, value_type* b) =
      \forall integer i; 0 <= i < n  ==>  \at(a[i],K) == \at(b[n-1-i], L);

    predicate
    Reverse{K,L}(value_type* a, integer m, integer n,
                 value_type* b, integer p) = Reverse{K,L}(a+m, n-m, b+p);

    predicate
    Reverse{K,L}(value_type* a, integer m, integer n, value_type* b) =
      Reverse{K,L}(a, m, n, b, m);

    predicate
    Reverse{K,L}(value_type* a, integer m, integer n, integer p) =
      Reverse{K,L}(a, m, n, a, p);

    predicate
    Reverse{K,L}(value_type* a, integer m, integer n) =
      Reverse{K,L}(a, m, n, m);

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

Listing 7.16: The predicate Reverse

In order to concisely formalize these conditions we define in Listing 7.16 the predicate Reverse (see also Figure 7.17). We also define several overloaded variants of Reverse that provide default values for some of the parameters. These overloaded versions enable us to write more concise ACSL annotations.

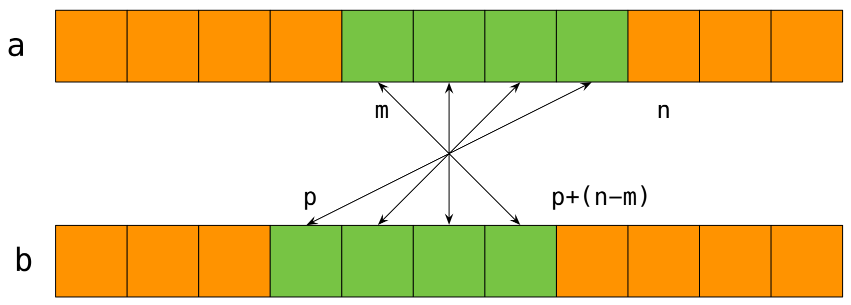


Figure 7.17.: Sketch of predicate Reverse

7.7.1. Formal specification of reverse_copy

The ACSL specification of reverse_copy is shown in Listing 7.18. We use the second version of predicate Reverse from Listing 7.16 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{Old,Here}(a, n, b);
  ensures unchanged:    Unchanged{Old,Here}(a, n);
*/
void
reverse_copy(const value_type* a, size_type n, value_type* b);
```

Listing 7.18: Formal specification of reverse_copy

7.7.2. Implementation of reverse_copy

Listing 7.19 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 7.16.

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

Listing 7.19: Implementation of reverse_copy

7.8. The reverse algorithm

The reverse algorithm of the C++ Standard Library [20, §28.6.10] inverts the order of elements in a sequence. Note that reverse works *in place*, meaning that it modifies its input sequence. Our modified signature reads:

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

7.8.1. Formal specification of reverse

The ACSL specification for the reverse function is shown in listing 7.20.

```
/*@
  requires valid:  \valid(a + (0..n-1));
  assigns          a[0..n-1];
  ensures reverse: Reverse{Old,Here}(a, n);
*/
void
reverse(value_type* a, size_type n);
```

Listing 7.20: Formal specification of reverse

7.8.2. Implementation of reverse

Listing 7.21 shows an implementation of the reverse function. Since the reverse algorithm operates *in place* we use the swap function from Section 7.3 in order to exchange the elements of the first half of the array with the corresponding elements of the second half. We reuse the predicates Reverse (Listing 7.16) and Unchanged (Listing 7.1) in order to write concise loop invariants.

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

  //@ assert half: half <= n - half;
  //@ assert half: 2*half <= n <= 2*half + 1;
  /*@
    loop invariant bound:    0 <= i <= half <= n-i;
    loop invariant left:     Reverse{Pre,Here}(a, 0, i, n-i);
    loop invariant middle: Unchanged{Pre,Here}(a, i, n-i);
    loop invariant right:    Reverse{Pre,Here}(a, n-i, n, 0);
    loop assigns i, a[0..n-1];
    loop variant half - i;
  */
  for (size_type i = 0u; i < half; ++i) {
    swap(&a[i], &a[n - 1u - i]);
  }
}
```

Listing 7.21: Implementation of reverse

7.9. The `rotate_copy` algorithm

The `rotate_copy` algorithm of the C++ Standard Library [20, §28.6.11] copies, in a particular way, the elements of one sequence of length n into a separate sequence. More precisely,

- the first m elements of the first sequence become the last m elements of the second sequence, and
- the last $n - m$ elements of the first sequence become the first $n - m$ elements of the second sequence.

Figure 7.22 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 subranges of the array `b[0..n-1]`.

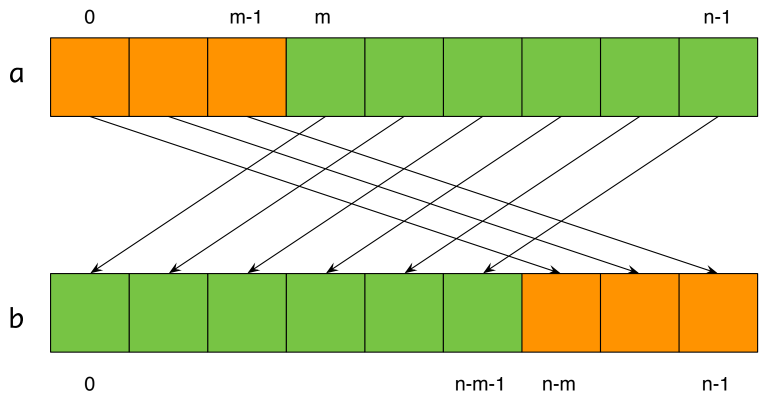


Figure 7.22.: Effects of `rotate_copy`

For our purposes we have modified the generic implementation 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);
```

7.9.1. Formal specification of `rotate_copy`

The ACSL specification of `rotate_copy` is shown in Listing 7.23. Note that we require explicitly that both ranges do not overlap and that we are only able to *read* from the range `a[0..n-1]`.

```
/*@
  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 left:        EqualRanges{Old,Here}(a, 0, m,   b, n-m);
  ensures right:       EqualRanges{Old,Here}(a, m, n-m, b, 0);
  ensures unchanged:   Unchanged{Old,Here}(a, n);
*/
void
rotate_copy(const value_type* a, size_type m, size_type n, value_type* b);
```

Listing 7.23: Formal specification of `rotate_copy`

7.9.2. Implementation of `rotate_copy`

Listing 7.24 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 7.24: Implementation of `rotate_copy`

7.10. The `rotate` algorithm

The algorithm `rotate` is an *in-place* variant of the algorithm `rotate_copy` of Section 7.9. We have modified the generic specification of `rotate` [20, §28.6.11] such that it refers to a range of objects of `value_type`. The signature now reads:

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

7.10.1. Formal specification of `rotate`

Figure 7.25 shows informally the behavior of `rotate`. The figure is of course very similar to the one for `rotate_copy` (see Figure 7.22). The notable difference is that `rotate` operates *in place* of the array `a[0..n-1]`.

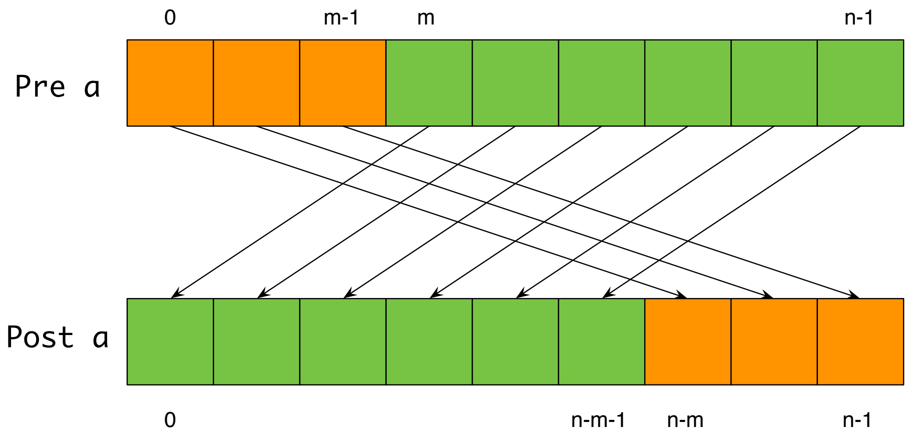


Figure 7.25.: Effects of `rotate`

The ACSL specification of `rotate` is shown in Listing 7.26.

```
/*@
  requires valid: \valid(a + (0..n-1));
  requires bound: m <= n;
  assigns         a[0..n-1];
  ensures result: \result == n-m;
  ensures left:   EqualRanges{Old,Here}(a, 0, m, n-m);
  ensures right:  EqualRanges{Old,Here}(a, m, n, 0);
*/
size_type
rotate(value_type* a, size_type m, size_type n);
```

Listing 7.26: Formal specification of `rotate`

7.10.2. Implementation of `rotate`

Listing 7.27 shows an implementation of the `rotate` function together with several ACSL annotations. Actually, there are several ways to implement `rotate`. We have chosen a particularly simple one that is derived from an implementation of `std::rotate` for *bidirectional iterators* [20, §27.2.6] and which essentially consists of several calls to the algorithm `reverse` of Section 7.8.

Note the statement contract of the final call of `reverse` Listing 7.27. Here we use both the labels `Pre` and `Old` which refer to the pre-states of `reverse` and the function `rotate` itself, respectively.

```
size_type
rotate(value_type* a, size_type m, size_type n)
{
  // if one subrange is empty, then nothings needs to be done
  if ((0u < m) && (m < n)) {
    reverse(a,   m);
    reverse(a + m, n - m);
    /*@
      requires left:   Reverse{Pre,Here}(a, 0, m, 0);
      requires right:  Reverse{Pre,Here}(a, m, n, m);

      assigns          a[0..n-1];

      ensures left:    Reverse{Old,Here}(a, 0, m, n-m);
      ensures right:   Reverse{Old,Here}(a, m, n, 0);
    */
    reverse(a, n);
    //@ assert left:   EqualRanges{Pre,Here}(a, 0, m, n-m);
    //@ assert right:  EqualRanges{Pre,Here}(a, m, n, 0);
  }

  return n - m;
}
```

Listing 7.27: Implementation of `rotate`

7.11. The `replace_copy` algorithm

The `replace_copy` algorithm of the C++ Standard Library [20, §28.6.5] substitutes specific elements from general sequences. Here, the general implementation 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.

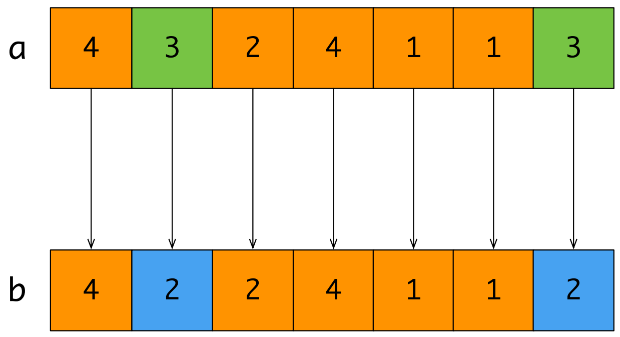


Figure 7.28.: Effects of `replace`

Figure 7.28 shows the behavior of `replace_copy` at hand of an example where all occurrences of the value 3 in `a[0..n-1]` are replaced with the value 2 in `b[0..n-1]`.

7.11.1. The predicate `Replace`

We start with defining in Listing 7.29 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 [15, §2.2]).

```
/*@
  axiomatic Replace
  {
    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 7.29: The predicate `Replace`

Listing 7.29 also contains a second, overloaded version of Replace which we will use for the specification of the related in-place algorithm replace in Section 7.12.

7.11.2. Formal specification of **replace_copy**

Using predicate Replace the ACSL specification of replace_copy is as simple as in Listing 7.30. We also require that the input range a[0..n-1] and output range b[0..n-1] do not overlap.

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

Listing 7.30: Formal specification of the replace_copy

7.11.3. Implementation of **replace_copy**

An implementation (including loop annotations) of replace_copy is shown in Listing 7.31. Note how the structure of the loop annotations resembles the specification of Listing 7.30.

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

  return n;
}
```

Listing 7.31: Implementation of the replace_copy algorithm

7.12. The replace algorithm

The replace algorithm of the C++ Standard Library [20, §28.6.5] substitutes specific values in a general sequence. Here, the general implementation 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.

7.12.1. Formal specification of replace

Using the second predicate Replace from Listing 7.29 the ACSL specification of replace can be expressed as in Listing 7.32.

```
/*@
  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 7.32: Formal specification of the replace

7.12.2. Implementation of replace

An implementation of replace is shown in Listing 7.33. 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 = 0u; i < n; ++i) {
    if (a[i] == v) {
      a[i] = w;
    }
  }
}
```

Listing 7.33: Implementation of the replace algorithm

7.13. The `remove_copy` algorithm (basic contract)

The `remove_copy` algorithm of the C++ Standard Library [20, §28.6.8] copies all elements of a sequence other than a given value. Here, the general implementation has been altered to process `value_type` ranges. The new signature reads:

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

The requirements of `remove_copy` are:

Requirements	Description
Remove Copy Size	The output range has to fit in all the elements of the input range, except the ones that equal the value `v` by `remove_copy`.
Remove Copy Separated	The input range and the output range do not overlap
Remove Copy Elements	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]`.
Remove Copy Stability	The algorithm is stable, that is, the relative order of the elements in `b` is the same as in `a`.
Remove Copy Return	The return value is the length of the resulting range.
Remove Copy Complexity	The algorithm takes *n* comparisons in every case.

Table 7.34.: Properties of `remove_copy`

Figure 7.35 shows an example of how `remove_copy` is supposed to copy elements that differ from `4` from the input range to the output range.

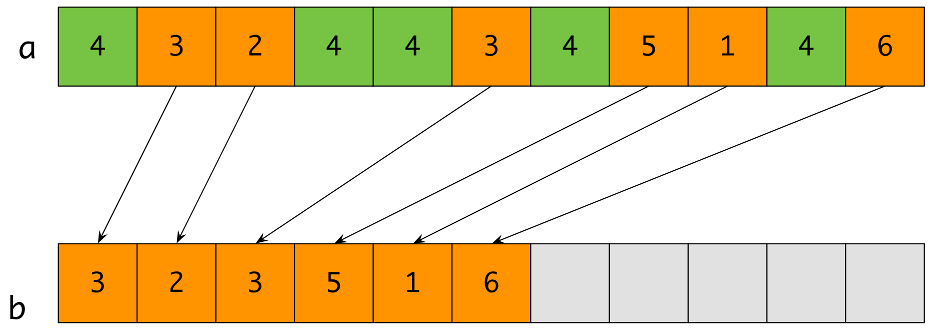


Figure 7.35.: Effects of `remove_copy`

7.13.1. Formal specification of `remove_copy`

Listing 7.36 shows our first attempt to specify `remove_copy`. In postcondition `discard` we use of the predicate `NoneEqual` from Listing 4.4 to show that the value `v` does not occur in the range `b[0..\result]`.

```
/*@
  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 discard:   NoneEqual(b, \result, v);
  ensures unchanged: Unchanged{Old,Here}(a, n);
  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 7.36: Formal specification of `remove_copy`

One shortcoming of this specification is that the postcondition `bound` only makes very general and not very precise statements about the number of copied elements. We will address this problem in Section 7.14. A more serious shortcoming is, however, that we haven't specified what the relationship between the elements of the input range `a[0..n-1]` and the output range `b[0..\result-1]` looks like. This problem will be discussed in Section 7.15.

7.13.2. Implementation of `remove_copy`

An implementation of `remove_copy` is shown in Listing 7.37.

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

  /*@
    loop invariant bound:     0 <= k <= i <= n;
    loop invariant discard:   NoneEqual(b, k, v);
    loop invariant unchanged: Unchanged{Pre,Here}(b, k, n);
    loop assigns   k, i, b[0..n-1];
    loop variant   n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    if (a[i] != v) {
      b[k++] = a[i];
    }
  }

  return k;
}
```

Listing 7.37: Implementation of `remove_copy`

Here we also need to add another loop invariant `discard` which basically checks if `v` occurs in `b[0..k]` for each iteration of the loop.

7.14. The `remove_copy` algorithm (number of copied elements)

This section rests on Section 7.13 and improves on the contract from Listing 7.36 by formally specifying the number `\result` of elements copied by `remove_copy`.

The number of copied elements equals of course the number of elements in the input range `a[0..n-1]` that are different from `v`. One can formally describe this number by relying on the logic function `Count` from Listing 4.36.

```
logic integer
CountNotEqual(value_type* a, integer n, value_type v) =  n - Count(a, n, v);
```

In fact, we have used this kind of definition in earlier version of this document. We have found it, however, worthwhile to provide a separate definition of `CountNotEqual` and express the relationship with `Count` as a lemma. This definition is shown in the Listings 7.38 and 7.39.

```
/*@
  axiomatic CountNotEqual
  {
    logic integer
    CountNotEqual(value_type* a, integer m, integer n, value_type v) =
      n <= m ? 0 : CountNotEqual(a, m, n-1, v) + (a[n-1] == v ? 0 : 1);

    logic integer
    CountNotEqual(value_type* a, integer n, value_type v) =
      CountNotEqual(a, 0, n, v);

    lemma CountNotEqual_Empty:
      \forall value_type *a, v, integer m, n;
        n <= m  ==>  CountNotEqual(a, m, n, v) == 0;

    lemma CountNotEqual_Hit:
      \forall value_type *a, v, integer m, n;
        m <= n      ==>
        a[n] != v   ==>
        CountNotEqual(a, m, n+1, v) == CountNotEqual(a, m, n, v) + 1;

    lemma CountNotEqual_Miss:
      \forall value_type *a, v, integer m, n;
        m <= n      ==>
        a[n] == v   ==>
        CountNotEqual(a, m, n+1, v) == CountNotEqual(a, m, n, v);

    lemma CountNotEqual_Lower:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  0 <= CountNotEqual(a, m, n, v);

    lemma CountNotEqual_Upper:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  CountNotEqual(a, m, n, v) <= n-m;
```

Listing 7.38: The logic function `CountNotEqual` (1)

The above mentioned relationship with `Count` is expressed as lemma `CountNotEqual_Count` in Listing 7.39.

```
    lemma CountNotEqual_WeaklyIncreasing:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  CountNotEqual(a, m, n, v) <= CountNotEqual(a, m, n+1, v);

    lemma CountNotEqual_Increasing:
      \forall value_type *a, v, integer k, m, n;
        m <= k <= n  ==>  CountNotEqual(a, m, k, v) <= CountNotEqual(a, m, n, v);

    lemma CountNotEqual_Read{K,L}:
      \forall value_type *a, v, integer m, n;
        Unchanged{K,L}(a, m, n)  ==>
        CountNotEqual{K}(a, m, n, v) == CountNotEqual{L}(a, m, n, v);

    lemma CountNotEqual_Count:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  CountNotEqual(a, m, n, v) == n - m - Count(a, m, n, v);

    lemma CountNotEqual_Union:
      \forall value_type *a, v, integer k, m, n;
        0 <= k <= m <= n  ==>
        CountNotEqual(a, k, n, v) ==
        CountNotEqual(a, k, m, v) + CountNotEqual(a, m, n, v);
  }
*/
```

Listing 7.39: The logic function `CountNotEqual` (2)

7.14.1. Formal specification of `remove_copy`

To extend our formal specification by using `CountNotEqual` we add the new postcondition `size`, which states that the returning value of `remove_copy` equals `CountNotEqual`. Listing 7.40 shows the extended formal specification with the new postcondition.

```
/*@
  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 size:      \result == CountNotEqual(a, n, v);
  ensures bound:     0 <= \result <= n;
  ensures discard:   NoneEqual(b, \result, v);
  ensures unchanged: Unchanged{Old,Here}(a, n);
  ensures unchanged: Unchanged{Old,Here}(b, \result, n);
*/
size_type
remove_copy2(const value_type* a, size_type n, value_type* b, value_type v);
```

Listing 7.40: Specification with `CountNotEqual`

7.14.2. Implementation of remove_copy

Listing 7.41 shows the implementation of our extended specification of remove_copy. Here we added the loop invariant size which corresponds to the postcondition in Listing 7.40. In order to ensure that the loop invariant size can be verified we have added the assertions size and unchanged.

```
size_type
remove_copy2(const value_type* a, size_type n, value_type* b, value_type v)
{
  size_type k = 0u;

  /*@
    loop invariant size:      k == CountNotEqual(a, i, v);
    loop invariant bound:     0 <= k <= i <= n;
    loop invariant discard:   NoneEqual(b, k, v);
    loop invariant unchanged: Unchanged{Pre,Here}(b, k, n);
    loop assigns   k, i, b[0..n-1];
    loop variant   n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    if (a[i] != v) {
      b[k++] = a[i];
      //@ assert unchanged: Unchanged{LoopCurrent,Here}(a, n);
      //@ assert size:      k == CountNotEqual(a, 0, i+1, v);
    }
  }

  return k;
}
```

Listing 7.41: Implementation with CountNotEqual

While we now can precisely speak of the number of copied elements, it is still not possible to say something about the exact relationship between the elements of range a[0..n-1] and range b[0..n-1]. We will address this question in Section 7.15.

7.15. The `remove_copy` algorithm (final contract)

In this section we extend the contract of `remove_copy` from §7.13 and §7.14 by introducing a logic function, which describes the relationship between the elements of input range `a[0..n-1]` and the output range `b[0..\result-1]`. Note that we have shown in the previous section that `\result` equals `CountNotEqual(a, n, v)`.

7.15.1. A closer look on the properties of `remove_copy`

Figure 7.42 shows a modified version of the Figure 7.35. We left out the indices of values that were not copied into the target array. Furthermore we have added a dashed arrow which points to the index that corresponds to the *one past the end* location of the input and output range.

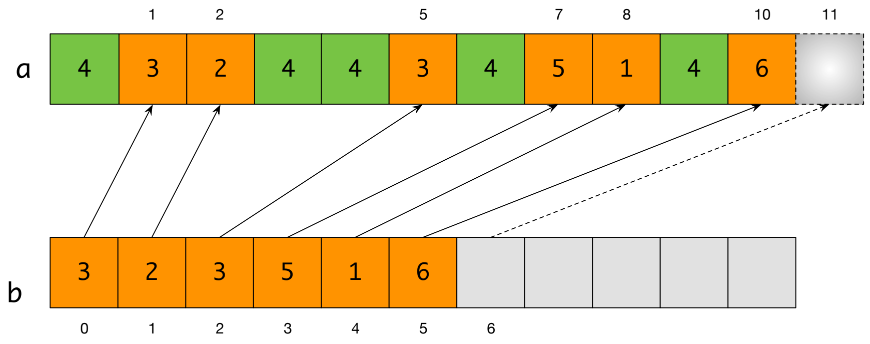


Figure 7.42.: Partitioning the input of `remove_copy`

These arrows between the indices of the array `b` and array `a` define the following sequence p of seven indices. The index of the *one past the end* is underlined. $p = (1, 2, 5, 7, 8, 10, \underline{11})$

More generally, we refer to the sequence p as *partitioning sequence* of `remove_copy` for the array `a[0..n-1]`. For the **length of a partitioning sequence** m we get the following **strictly monotone increasing** sequence:

$$0 \le p_0 < \ldots < p_m = n \tag{7.1}$$

and the open index intervals

$$(p_i, p_{i+1}) \qquad \forall i : 0 \le i < m$$

mark **consecutive ranges** of the value `v` in the source array, that is,

$$a[k] = v \qquad \forall k : p_i < k < p_{i+1} \tag{7.2}$$

Additionally, the half open interval

$$[0, p_0)$$

also marks another **consecutive range** of the value `v` in the source array:

$$a[k] = v \qquad \forall k : 0 \le k < p_0 \tag{7.3}$$

Another observation is that

$$a[p_i] \neq v \qquad \forall i : 0 \leq i < m \tag{7.4}$$

holds. Finally, we have

$$a[p_i] = b[i] \qquad \forall i : 0 \leq i < m \tag{7.5}$$

which, together with the inequality (7.4) states, that the target does not contain the value `v`

$$b[i] \neq v \qquad \forall i : 0 \leq i < m$$

7.15.2. More auxiliary predicates, logic functions and lemmas

Our formalization the properties of Section 7.15.1 relies on the logic function `CountNotEqual` which we already introduced in Listing 7.38. We also rely on the logic Function `FindNotEqual` (Listings 7.43 and 7.44). This function is very similar to `Find` (Listing 4.7) except that it finds the first element in a sequence that *differs* from a given value.

```
/*@
  axiomatic FindNotEqual
  {
    logic integer
    FindNotEqual(value_type* a, integer m, integer n, value_type v) =
      (n <= m) ?
       0 : ((0 <= FindNotEqual(a, m, n-1, v) < n-m-1) ?
         FindNotEqual(a, m, n-1, v) : ((a[n-1] != v) ? n-m-1 : n-m));

    logic integer
    FindNotEqual(value_type* a, integer n, value_type v) =
      FindNotEqual(a, 0, n, v);

    lemma FindNotEqual_Empty:
      \forall value_type *a, v, integer m, n;
        n <= m  ==>  FindNotEqual(a, m, n, v) == 0;

    lemma FindNotEqual_Hit:
      \forall value_type *a, v, integer m, n;
        m <= n                         ==>
        FindNotEqual(a, m, n, v) < n-m  ==>
        FindNotEqual(a, m, n+1, v) == FindNotEqual(a, m, n, v);

    lemma FindNotEqual_MissHit:
      \forall value_type *a, v, integer m, n;
        m <= n                            ==>
        a[n] != v                         ==>
        FindNotEqual(a, m, n, v)    == n-m  ==>
        FindNotEqual(a, m, n+1, v) == n-m;

    lemma FindNotEqual_MissMiss:
      \forall value_type *a, v, integer m, n;
        m <= n                            ==>
        a[n] == v                         ==>
        FindNotEqual(a, m, n, v)    == n-m  ==>
        FindNotEqual(a, m, n+1, v) == (n+1)-m;
```

Listing 7.43: The logic function `FindNotEqual` (1)

```
    lemma FindNotEqual_Lower:
      \forall value_type *a, v, integer m, n;
        0 <= FindNotEqual(a, m, n, v);

    lemma FindNotEqual_Upper:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  FindNotEqual(a, m, n, v) <= n-m;

    lemma FindNotEqual_Read{K,L}:
      \forall value_type *a, v, integer m, n;
        Unchanged{K,L}(a, m, n)  ==>
        FindNotEqual{K}(a, m, n, v) == FindNotEqual{L}(a, m, n, v);

    lemma FindNotEqual_WeaklyIncreasing:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  FindNotEqual(a, m, n, v) <= FindNotEqual(a, m, n+1, v);

    lemma FindNotEqual_Extend:
      \forall value_type *a, v, integer k, m, n;
        m <= k < n                          ==>
        a[k] != v                           ==>
        FindNotEqual(a, m, k, v) == k-m     ==>
        FindNotEqual(a, m, n, v) == k-m;

    lemma FindNotEqual_Increasing:
      \forall value_type *a, v, integer k, m, n;
        m <= k <= n  ==>  FindNotEqual(a, m, k, v) <= FindNotEqual(a, m, n, v);

    lemma FindNotEqual_Limit:
      \forall value_type *a, v, integer k, m, n;
        m <= k < n  ==>
        a[k] != v   ==>
        FindNotEqual(a, m, n, v) <= k-m;

    lemma FindNotEqual_AllEqual:
      \forall value_type *a, v, integer m, n;
        m <= n                 ==>
        AllEqual(a, m, n, v)   ==>
        FindNotEqual(a, m, n, v) == n-m;

    lemma FindNotEqual_SomeNotEqual:
      \forall value_type *a, v, integer k, m, n;
        m <= k < n             ==>
        a[k] != v              ==>
        AllEqual(a, m, k, v)   ==>
        FindNotEqual(a, m, n, v) == k-m;

    lemma FindNotEqual_ResultAllEqual:
      \forall value_type *a, v, integer m, n;
        m <= n  ==>  AllEqual(a, m, m + FindNotEqual(a, m, n, v), v);

    lemma FindNotEqual_ResultNotEqual:
      \forall value_type *a, v, integer m, n;
        0 <= FindNotEqual(a, m, n, v) < n-m  ==>
        a[m + FindNotEqual(a, m, n, v)] != v;
  }
*/
```

Listing 7.44: The logic function `FindNotEqual` (2)

We also rely on the lemmas in Listing 7.45 that relate the functions `CountNotEqual` and `FindNotEqual`.

```
/*@
  axiomatic CountFindNotEqual
  {
    lemma CountNotEqual_AllEqual:
      \forall value_type *a, v, integer m, n;
        0 <= m <= n              ==>
        AllEqual(a, m, n, v)     ==>
        CountNotEqual(a, m, n, v) == 0;

    lemma CountNotEqual_SomeNotEqual:
      \forall value_type *a, v, integer m, n;
        0 <= m < n                      ==>
        0 < CountNotEqual(a, m, n, v)   ==>
        SomeNotEqual(a, m, n, v);

    lemma CountNotEqual_FindNotEqual:
      \forall value_type *a, v, integer m, n;
        0 <= m < n                      ==>
        0 < CountNotEqual(a, m, n, v)   ==>
        FindNotEqual(a, m, n, v) < n-m;

    lemma CountNotEqual_Zero:
      \forall value_type *a, v, integer m, n;
        0 <= m < n    ==>
        CountNotEqual(a, m, m + FindNotEqual(a, m, n, v), v) == 0;

    lemma CountNotEqual_Decrement:
      \forall value_type *a, v, integer m, n;
        0 <= m < n   ==>
        CountNotEqual(a, m + FindNotEqual(a, m, n, v), n, v) ==
        CountNotEqual(a, 0, n, v) - CountNotEqual(a, 0, m, v);
  }
*/
```

.

Listing 7.45: More lemma on relating `CountNotEqual` and `FindNotEqual`

7.15.3. Formalizing the properties of the partitions of `remove_copy`

The function `RemovePartition`, whose axiomatic definition is given in Listing 7.46, defines the partitioning sequence p from Section 7.15.1. Before we begin to relate the lemmas from Listings 7.46 and 7.47 to the formulas from Section 7.15.1 we want to remind the reader that logic functions (and predicates) must be total that is they must be defined for all possible argument values.

The lemmas for `RemovePartition` are related to the properties of Section 7.15.1 in the following way.

- Property (7.1) is expressed by the lemmas `RemovePartition_Empty`, `RemovePartition_Left` `RemovePartition_Right`, and `RemovePartition_StrictlyIncreasing`
- Properties (7.2) and (7.3) are described by lemmas `RemovePartition_Segment`.
- Property (7.4) is expressed by lemma `RemovePartition_NotEqual`.
- Property (7.5) is formulated using the predicate `Remove` from Listing 7.48.

```
/*@
  axiomatic RemovePartition
  {
    logic integer
    NextNotEqual(value_type* a, integer x, integer n, value_type v) =
      x + FindNotEqual(a, x, n, v);

    logic integer
    RemovePartition(value_type* a, integer n, value_type v, integer p) =
      \let c = CountNotEqual(a, n, v);
      \let x = RemovePartition(a, n, v, p-1) + 1;
        p < 0 ? -1 : // 0 <= p
          (n <= 0 ? 0 : // 0 < n
            p < c ? NextNotEqual(a, x, n, v) : n
          );

    lemma RemovePartition_Empty:
      \forall value_type *a, v, integer n, p;
        n <= 0 <= p  ==>
        RemovePartition(a, n, v, p) == 0;

    lemma RemovePartition_Left:
      \forall value_type *a, v, integer n, p;
        p < 0  ==>  RemovePartition(a, n, v, p) == -1;

    lemma RemovePartition_Right:
      \forall value_type *a, v, integer n, p;
        0 <= n                        ==>
        CountNotEqual(a, n, v) <= p   ==>  RemovePartition(a, n, v, p) == n;

    lemma RemovePartition_Next:
      \forall value_type *a, v, integer n, p;
        \let x = RemovePartition(a, n, v, p-1) + 1;
        0 <= n                            ==>
        0 <= p < CountNotEqual(a, n, v)   ==>
        RemovePartition(a, n, v, p) == x + FindNotEqual(a, x, n, v);

    lemma RemovePartition_Lower:
      \forall value_type *a, v, integer i, n, p;
        0 < n                             ==>
        0 <= p < CountNotEqual(a, n, v)   ==>
        0 <= RemovePartition(a, n, v, p);

    lemma RemovePartition_Core:
      \forall value_type *a, v, integer i, n, p;
        \let R = RemovePartition(a, n, v, p);
        0 < n                             ==>
        0 <= p < CountNotEqual(a, n, v)   ==>
        (R < n      &&
         a[R] != v  &&
         CountNotEqual(a, R, n, v) == CountNotEqual(a, 0, n, v) - p);
```

Listing 7.46: The logic function `RemovePartition` (1)

Note that Listing 7.46 also contains the logic function `NextNotEqual` which is a workaround for a problem in the current Frama-C release.[23]

[23]See `https://bts.frama-c.com/view.php?id=2501`

```
    lemma RemovePartition_Upper:
      \forall value_type *a, v, integer i, n, p;
        0 < n                           ==>
        0 <= p < CountNotEqual(a, n, v)  ==>
        RemovePartition(a, n, v, p) < n;

    lemma RemovePartition_NotEqual:
      \forall value_type *a, v, integer n, p;
        0 < n  ==>
        0 <= p < CountNotEqual(a, n, v) < n  ==>
        a[RemovePartition(a, n, v, p)] != v;

    lemma RemovePartition_Count:
      \forall value_type *a, v, integer n, p;
        0 < n                           ==>
        0 <= p < CountNotEqual(a, n, v)  ==>
        CountNotEqual(a, RemovePartition(a, n, v, p), n, v) ==
        CountNotEqual(a, 0, n, v) - p;

    lemma RemovePartition_StrictlyWeakIncreasing:
      \forall value_type *a, v, integer n, p;
        0 < n                          ==>
        0 < p < CountNotEqual(a, n, v)  ==>
        RemovePartition(a, n, v, p-1) < RemovePartition(a, n, v, p);

    lemma RemovePartition_StrictlyIncreasing:
      \forall value_type *a, v, integer n, p, q;
        0 < n                          ==>
        0 <= p < q < CountNotEqual(a, n, v)  ==>
        RemovePartition(a, n, v, p) < RemovePartition(a, n, v, q);

    lemma RemovePartition_Segment:
      \forall value_type *a, v, integer i, n, p;
        0 < n                           ==>
        0 < p <= CountNotEqual(a, n, v)  ==>
          AllEqual(a, RemovePartition(a, n, v, p-1) + 1,
                      RemovePartition(a, n, v, p), v);

    lemma RemovePartition_Extend:
      \forall value_type *a, v, integer n, p;
        0 < n                           ==>
        0 <= p < CountNotEqual(a, n, v)  ==>
        RemovePartition(a, n, v, p) == RemovePartition(a, n+1, v, p);

    lemma RemovePartition_Read{K,L}:
      \forall value_type *a, v, integer n, p;
        Unchanged{K,L}(a, n)  ==>
        RemovePartition{K}(a, n, v, p) == RemovePartition{L}(a, n, v, p);
  }
*/
```

Listing 7.47: The logic function `RemovePartition` (2)

The predicate `Remove` from Listing 7.48 primarily serves in order to improve the readability of our specification `remove_copy`. As mentioned before this predicate encapsulates the Property (7.5) from Section 7.15.1.

```
/*@
  axiomatic Remove
  {
    predicate
    Remove{K,L}(value_type* a, integer n, value_type* b, value_type v) =
      \forall integer k; 0 <= k < CountNotEqual{K}(a, n, v)  ==>
        \at(b[k],L) == \at(a[RemovePartition(a, n, v, k)],K);

    predicate
    Remove{K,L}(value_type* a, integer n, value_type v) =
      \forall integer k; 0 <= k < CountNotEqual{K}(a, n, v)  ==>
        \at(a[k],L) == \at(a[RemovePartition(a, n, v, k)],K);

    lemma Remove_Update{K,L}:
      \forall value_type *a, v, integer m;
        \let k = CountNotEqual{K}(a, m+1, v) - 1;
        0 <= m                                                    ==>
        Remove{K,L}(a, m, v)                                      ==>
        \at(a[m],K) != v                                          ==>
        \at(a[k],L) == \at(a[RemovePartition(a, m+1, v, k)],K)   ==>
        Remove{K,L}(a, m+1, v);
  }
*/
```

Listing 7.48: The predicate `Remove`

Note that Listing 7.48 also contains an overloaded version of `Remove` which will be used for the *in-place* variant `remove` from Section 7.16 of `remove_copy`.

7.15.4. Formal specification of `remove_copy`

The Listing 7.49 shows the extended version of the formal specification of `remove_copy`. The additional postcondition `remove` makes use of the predicate `Remove` which we have just described. Furthermore, we have again the postcondition `unchanged` which states that the source array `a[0..n-1]` does not change.

```
/*@
  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 size:      \result == CountNotEqual{Old}(a, n, v);
  ensures bound:     0 <= \result <= n;
  ensures remove:    Remove{Old,Here}(a, n, b, v);
  ensures discard:   NoneEqual(b, \result, v);
  ensures unchanged: Unchanged{Old,Here}(a, n);
  ensures unchanged: Unchanged{Old,Here}(b, \result, n);
*/
size_type
remove_copy3(const value_type* a, size_type n, value_type* b, value_type v);
```

Listing 7.49: An extended specification for `remove_copy`

7.15.5. Implementation of remove_copy

The listing 7.50 shows the more detailed implementation of remove_copy. In addition to the loop invariant remove and unchanged which are used by analogy with the postconditions remove and unchanged. Note that there is no need for the loop invariant discard as the corresponding postcondition is automatically deduced from the properties of RemovePartition, in particular, lemma RemovePartition_NotEqual.

```
size_type
remove_copy3(const value_type* a, size_type n, value_type* b, value_type v)
{
  size_type k = 0u;

  /*@
    loop invariant size:       k == CountNotEqual{Pre}(a,i,v);
    loop invariant bound:      0 <= k <= i <= n;
    loop invariant remove:     Remove{Pre,Here}(a, i, b, v);
    loop invariant mapping:    i <= RemovePartition{Pre}(a, n, v, k);
    loop invariant unchanged:  Unchanged{Pre,Here}(a, n);
    loop invariant unchanged:  Unchanged{Pre,Here}(b, k, n);
    loop assigns   k, i, b[0..n-1];
    loop variant   n-i;
  */
  for (size_type i = 0u; i < n; ++i) {
    if (a[i] != v) {
      b[k++] = a[i];
      //@ assert size:     k == CountNotEqual{Pre}(a, 0, i+1, v);
      //@ assert mapping: i == RemovePartition{Pre}(a, n, v, k-1);
      //@ assert update:  b[k-1] == \at(a[\at(i,Here)], Pre);
      //@ assert remove:  Remove{Pre,Here}(a, i,   b, v);
      //@ assert remove:  Remove{Pre,Here}(a, i+1, b, v);
    }
  }

  return k;
}
```

Listing 7.50: The extended annotations for remove_copy

We also introduce the loop invariant mapping. This invariant states that the variable i will always be smaller or equal to the result of RemovePartition(a, n, v, k). We also add the assertion mapping to our implementation as stepping stone for the provers to verify the correctness of this loop invariant.

Regarding the assertion update, one might wonder why we do not simply write \at(a[i], Pre). However, this expression would be wrong because the index i would then be interpreted as \at(i,Pre) which doesn't makes sense for a local variable. Frama-C/WP consequently rejects this expression with the following error message.

```
Warning: unbound logic variable i. Ignoring code annotation
```

As a solution we explicitly refer to the current value of i by using the subexpression \at(i,Here) inside the assertion update.

7.16. The `remove` algorithm

The C++ Standard Library also contains a function `remove` [20, 28.6.8] performing the same operation as `remove_copy` as an in-place algorithm. Its signature is very similar to that of `remove_copy`, except that there is no need for an output array.

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

Figure 7.51 shows how `remove` is supposed to remove all occurrences of the given value 4 from a range.

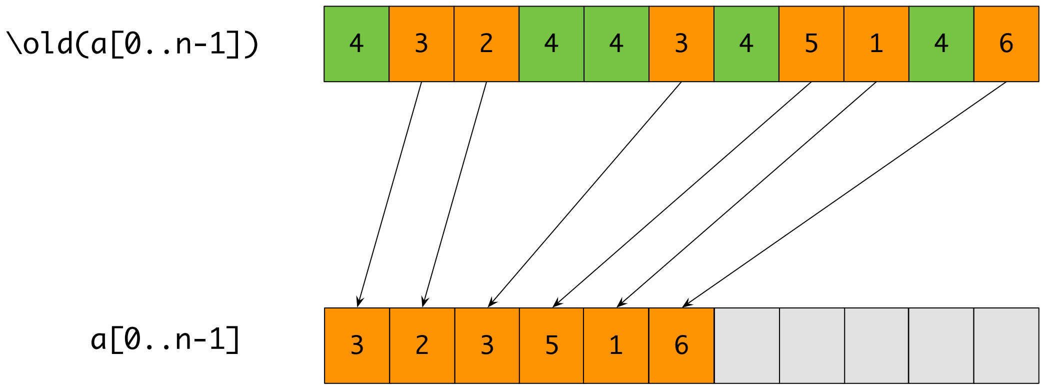


Figure 7.51.: Effects of `remove`

7.16.1. Formal specification of `remove`

Listing 7.52 shows a formal specification of the `remove` function. Our specification is very similar to the one of `remove_copy` in Listing 7.49 except that we using a version of `Remove` that takes only one pointer argument (see Listing 7.48).

```
/*@
  requires valid:     \valid(a + (0..n-1));
  assigns             a[0..n-1];
  ensures size:       \result == CountNotEqual{Old}(a, n, v);
  ensures bound:      0 <= \result <= n;
  ensures remove:     Remove{Old,Here}(a, n, v);
  ensures discard:    NoneEqual{Here}(a, \result, v);
  ensures unchanged: Unchanged{Old,Here}(a, \result, n);
*/
size_type
remove(value_type* a, size_type n, value_type v);
```

Listing 7.52: Formal specification of `remove`

7.16.2. Implementation of `remove`

Listing 7.53 shows our implementation of `remove` together with the additional loop annotations. Again, the annotations are very similar to those in Listing 7.50.

```
size_type
remove(value_type* a, size_type n, value_type v)
{
  size_type k = 0u;

  /*@
    loop invariant size:       k == CountNotEqual{Pre}(a,i,v);
    loop invariant bound:      0 <= k <= i <= n;
    loop invariant remove:     Remove{Pre,Here}(a, i, v);
    loop invariant mapping:    i <= RemovePartition{Pre}(a, n, v, k);
    loop invariant unchanged:  Unchanged{Pre,Here}(a, k, n);
    loop invariant unchanged:  a[k] == At{Pre}(a, k);
    loop assigns k, i, a[0..n-1];
    loop variant n-i;
  */
  for (size_type i = 0u; i < n; ++i ) {
    if (a[i] != v) {
      a[k++] = a[i];
      //@ assert size:    k == CountNotEqual{Pre}(a, 0, i+1, v);
      //@ assert mapping: i == RemovePartition{Pre}(a, n, v, k-1);
      //@ assert update:  a[k-1] == At{Pre}(a, RemovePartition{Pre}(a, i+1, v, k-1));
      //@ assert remove:  Remove{Pre,Here}(a, i, v);
      //@ assert remove:  Remove{Pre,Here}(a, i+1, v);
    }
  }

  return k;
}
```

Listing 7.53: Implementation of `remove`

Also note the use of the predicate `At` (Listing 7.54) in the assertion `update`. We use this predicate to simplify the comparison with array elements in the pre-state of the function.

```
/*@
  axiomatic At
  {
    logic value_type At{L}(value_type* x, integer i) = \at(x[i],L);
  }
*/
```

Listing 7.54: The predicate `At`

The second argument `At` is interpreted at the programme point here it appears, that is, `Here`. Without this this auxiliary logic function the tight hand side of assertion `update` would look like this.

```
\at(a[\at(RemovePartition{Pre}(a, i+1, v, k-1),Here)], Pre)
```

The second loop invariant `unchanged` benefits from using the function `At` since otherwise one would have to write

```
a[k] == \at(a[\at(k, Here)], Pre);
```

7.17. The `shuffle` algorithm

The `shuffle` algorithm in the C++ Standard Library [20, §28.6.13] 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 to that of a range of type `value_type`. The signature now reads:

```
void shuffle(value_type* a, size_type n, unsigned short* rand);
```

The argument `rand` holds the state of a simple random number generator that is used in the implementation of `shuffle`.

Figure 7.55 illustrates an example run of `shuffle`. In this figure, the values 1, 2, 3, and 4 occur twice, once, once, and three times, respectively, both before and after the `shuffle` run. This expresses that the range has been reordered.

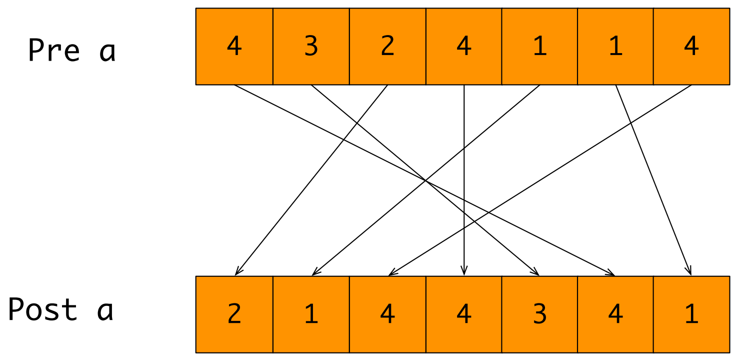


Figure 7.55.: Effects of `shuffle`

7.17.1. The predicate `MultisetUnchanged`

The `shuffle` algorithm is the first example in this document where we have to specify a *rearrangement* or *reordering* of the elements of a given range. We say that an an array has been reordered between two states if the the number of each element in the array remains unchanged. In other words, reordering leaves the *multiset*[24] of elements in the range unchanged.

We use the predicate `MultisetUnchanged`, defined in Listing 7.56, 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 4.36. We also formulate several lemma with basic properties of `MultisetUnchanged`. We will use these lemmas during the verification of various algorithms.

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

```
/*@
  axiomatic MultisetUnchanged
  {
    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);

    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 i, k, n;
        0 <= i <= k <= n                ==>
        MultisetUnchanged{L1,L2}(a, i, k)  ==>
        MultisetUnchanged{L1,L2}(a, k, n)  ==>
        MultisetUnchanged{L1,L2}(a, i, 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 7.56: The predicate `MultisetUnchanged`

7.17.2. Formal specification of `shuffle`

The ACSL specification of `shuffle` is shown in Listing 7.57. The `shuffle` algorithm expects that the range `a` is valid for reading and writing. We use the predicate `MultisetUnchanged` defined in Listing 7.56 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. The array `rand` contains a seed for the random number generator used to randomize the shuffle. By specifying that the function assigns to `rand` we capture that the function may return a different permutation every time.

```
/*@
  requires valid:   \valid(a + (0..n-1));
  requires valid:   \valid(seed + (0..2));
  requires sep:     \separated(a + (0..n-1), seed + (0..2));
  assigns           a[0..n-1];
  assigns           seed[0..2];
  ensures  reorder: MultisetUnchanged{Old,Here}(a,n);
*/
void
shuffle(value_type* a, size_type n, unsigned short* seed);
```

Listing 7.57: Formal specification of `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 `shuffle` is required to be random. The informal specification [20, §28.6.13] of `shuffle` states that *that each possible permutation of those elements has equal probability of appearance*. 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, we refer to a paper [22, p.6–8], which 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 [23].

7.17.3. Implementation of `shuffle`

Listing 7.58 shows an implementation of the `shuffle` function. It repeatedly calls the function `swap` from Section 7.3 to *transpose* (randomly) selected elements. For details of the hereby used function `random_number` we refer to Section 7.18.

```
void
shuffle(value_type* a, size_type n, unsigned short* seed)
{
  if (0u < n) {
    /*@
      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], seed[0..2];
      loop variant   n - i;
    */
    for (size_type i = 1u; i < n; ++i) {
      size_type k = random_number(seed, i) + 1u;

      //@ assert less: 0 <= k <= i;
      if (k < i) {
        swap(&a[k], &a[i]);
        //@ assert swapped: SwappedInside{LoopCurrent,Here}(a, k, i, n);
        //@ assert reorder: MultisetUnchanged{LoopCurrent,Here}(a, i+1);
        //@ assert reorder: MultisetUnchanged{Pre,Here}(a, i+1);
      }

      //@ assert reorder: MultisetUnchanged{Pre,Here}(a, i+1);
    }
  }
}
```

Listing 7.58: Implementation of `shuffle`

The loop invariants `reorder` and `unchanged` of `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 several auxiliary assertions `reorder` which use the the predefined label `LoopCurrent`, to guide the automatic verification the loop invariant `reorder`.

We introduce the predicate SwappedInside, shown in Listing 7.59 rather than the literal postcondition of swap, since this leads to better performance of the automatic provers.

```
/*@
  axiomatic SwappedInside
  {
    predicate
    SwappedInside{K,L}(value_type* a, integer i, integer k, integer n) =
      0 <= i < k < n              &&
      \at(a[i],K) == \at(a[k],L)  &&
      \at(a[k],K) == \at(a[i],L)  &&
      Unchanged{K,L}(a, 0,   i)   &&
      Unchanged{K,L}(a, i+1, k)   &&
      Unchanged{K,L}(a, k+1, n);

    lemma SwappedInsideReorder{K,L}:
      \forall value_type* a, integer i, k, n;
        SwappedInside{K,L}(a, i, k, n)  ==>
        MultisetUnchanged{K,L}(a, i, k+1);

    lemma SwappedInsidePreserve{K,L,M}:
      \forall value_type* a, integer i, k, n;
        MultisetUnchanged{K,L}(a, k)    ==>
        Unchanged{K,L}(a, k, n)         ==>
        SwappedInside{L,M}(a, i, k, n)  ==>
        MultisetUnchanged{K,M}(a, k+1);
  }
*/
```

Listing 7.59: The predicate SwappedInside

The lemma, SwappedInsideReorder states that swapping the elements a[i] and a[k] is a particular kind of reordering on the range a[i..k].

This lemma is extended to lemma SwappedInsidePreserve which additionally considers a left context a[0..k-1] and a right context a[k..n-1]; if the left and right context is reordered and kept untouched, respectively, and a[i] and a[k] are swapped as before, then this whole action is a reordering on the range a[0..k]. These two lemmas are useful for proving that the loop invariant reorder is preserved and can be applied in similar circumstances as well (cf. Section 11.2).

7.18. Verifying a random number generator

We describe in this section the implementation of a simple random-number generator named `random_number` which is specified in Listing 7.60. As in the case of `shuffle` itself, we do not formulate specific properties of randomness and only require its result to be in the specified range `[0..n-1]`. Again, the `assigns` clause to the array `state` models the dependency on an additional state.

Note that Listing 7.60 also provide the rather simple specification of the function `random_init` that is called to initialize the state of the random generator.

```
/*@
  requires  pos:    0 < n;
  requires  valid:  \valid(state + (0..2));
  assigns           state[0..2];
  ensures   result: 0 <= \result < n;
*/
size_type
random_number(unsigned short* state, size_type n);


/*@
  requires  \valid(state + (0..2));

  assigns   state[0..2];
*/
void
random_init(unsigned short* state);
```

Listing 7.60: Formal specification of `random_number`

The implementations of `random_number` and `random_init` are shown in Listing 7.61. Internally, we rely on a custom implementation of the POSIX.1 random number generator `lrand48()`[25] This random number generator is a linear congruence generator with a 48 bit state and the iteration procedure

$$x_{n+1} = ax_n + c \bmod 2^{48} \tag{7.6}$$

where $a = 25214903917$ and $c = 11$ are relatively prime integers.

As a part of the iteration procedure in Equation (7.6) an unsigned overflow may occur. This does not affect the result as we are only interested in its lowest 48 bits. However, as one of the options we use, `-warn-unsigned-overflow`, causes Frama-C/WP assert the absence of unsigned overflow this algorithm does not verify under the same options used for the other algorithms. As an exception, we have therefore decided to disable `-warn-unsigned-overflow` for this function as the unsigned overflow is both benign and well-defined (cf. [17, §6.2.5, 9]).

[25]See `http://pubs.opengroup.org/onlinepubs/9699919799/functions/lrand48.html`

```
// see IEEE 1003.1-2008, 2016 Edition for specification
/*@
  requires valid: \valid(seed + (0..2));
  assigns seed[0..2];
  ensures lower:  0 <= \result;
  ensures upper:  \result <= 0x7fffffff;
*/
static long
my_lrand48(unsigned short* seed)
{
  unsigned long long state = (unsigned long long)seed[0] << 32
                           | (unsigned long long)seed[1] << 16
                           | (unsigned long long)seed[2];
  state = (0x5deece66dull * state + 0xbull) % (1ull << 48);
  //@ assert lower: state < (1ull << 48);
  long result = state / (1ull << 17);
  //@ assert lower: 0 <= result;
  seed[0u] = state >> 32 & 0xffff;
  seed[1u] = state >> 16 & 0xffff;
  seed[2u] = state >>  8 & 0xffff;
  return result;
}

size_type
random_number(unsigned short* state, size_type n)
{
  return my_lrand48(state) % n;
}

void
random_init(unsigned short* state)
{
  state[0] = 0x243f;
  state[1] = 0x6a88;
  state[2] = 0x85a3;
}
```

Listing 7.61: Implementation of `random_number`

Note that we use the custom ACSL lemma `RandomNumberModulo` which we introduce in Listing 7.62 to support the verification of some assertions.

```
/*@
  axiomatic C_Bit
  {
    lemma RandomNumberModulo:
      \forall unsigned long long a;
        (a % (1ull << 48)) < (1ull << 48);
  }
*/
```

Listing 7.62: The lemma `RandomNumberModulo`

8. A closer look at `unique_copy`

In this chapter we take a closer look on testing and formal verification of the algorithm `unique_copy` from the C++ standard library. We start in Section 8.1 with a careful analysis of `unique_copy`'s informal requirements. We use the identified requirements to guide our testing and verification artefacts in the subsequent chapters.

Although testing is rightly considered easier than formal verification, *designing* both good test code and convincing test data is far from trivial. This is particularly true when one has to deal with testing properties that rely on *implicit* relationships between the input and output data of an algorithm. In the case of `unique_copy`, the main problem is to show that the first (and only the first) element of each consecutive range of equal elements is copied. We tackle this problem in Section 8.2 by exploiting the fact that we deal with a *generic* algorithm. This allows us to transport additional data through the algorithm under test and subsequently use this data to establish that the original data are processed according to the requirements.

In Section 8.3 we then proceed to formally verify a non-generic C version of `unique_copy` with the Frama-C [2] verification platform. In particular, we are writing formal function contract in Frama-C's specification language ACSL [9]. Here, we emphasize that there are different levels of how formal one wants to be. Thus, we consider both the formal verification of the *absence of undefined behavior* and the verification of the *functional correctness* of `unique_copy`.

8.1. Informal specification of `unique_copy`

This section deals with the informal specification of `unique_copy` and its behavior. In Section 8.1.1 we present the requirements of `unique_copy` as they are derived from the C++ standard library. In Section 8.1.2 we look at specific examples to provide a better understanding of `unique_copy`. To take it one step further we finally present in Section 8.1.3 a more formal analysis of `unique_copy`'s behavior.

8.1.1. What does the C++ standard says about `unique_copy`?

The `unique_copy` algorithms of the C++ standard library [20, §28.6.9], whose signature is shown in Listing 8.1, is a template function which copies certain values from a sequence given by the right-open interval of iterators `[first, last)` into a sequence that starts at the iterator `result`.

```
template<class InputIterator, class OutputIterator>
OutputIterator
unique_copy(InputIterator first, InputIterator last, OutputIterator result);
```

Listing 8.1: Signature of `unique_copy` from the C++ standard library

For the purposes of this report we do not consider the wide possibilities of ranges covered by this signature, rather we assume the two ranges to be arrays[26] `a[0..n-1]` and `b[0..n-1]` of length `n`. Listing 8.2

[26]We employ here and in the following the ACSL notation `a[0..n-1]` to denote an array of `n` elements.

shows thus the signature and implementation of a simplified yet still generic function `unique_copy`. Later in this document we will consider an even more specific version of `unique_copy` that is implemented in C.

```
template<typename T>
size_type unique_copy(const T* a, size_type n, T* b)
{
  auto result = std::unique_copy(a, a + n, b);
  return result - b;
}
```

Listing 8.2: A simplified version of `unique_copy`

Besides assuming the input and output ranges to be (generic) arrays, this version of `unique_copy` returns the number of copied elements instead of an iterator that indicates the last copied element in the output range.

Table 8.3 shows our interpretation of the requirements for `unique_copy` from the C++ standard[28.6.9] [20] for the signature of Listing 8.2.

Requirement	**Description**
Unique Copy Size	The output range must be able to store the same number of elements as the input range.
Unique Copy Separation	The input range and the output range do not overlap.
Unique Copy Consecutive	Only the first element from every consecutive group of equal elements of the input range is copied into the output range.
Unique Copy Return	The algorithm returns the number of copied elements.
Unique Copy Complexity	At most $n - 1$ comparisons of adjacent elements are performed.

Table 8.3.: Requirements of `unique_copy`

Requirement **Unique Copy Consecutive** captures the core functionality of the algorithms. The intention of this requirement, which is not explicitly mentioned in **Unique Copy Consecutive**, is that the copied elements do *not* contain adjacent equal elements. One goal of this report is to show how this requirement can be expressed in Frama-C's specification language ACSL.

Complexity requirements, as formulated in terms of the number of comparison operations in requirement **Unique Copy Complexity**, are essential for specifying efficient algorithms. However, since Frama-C does currently not provide sufficient support for specifying this kind of requirements, we will not consider them in the rest of this document.

8.1.2. Some examples for `unique_copy`

In this section we analyze the requirement **Unique Copy Consecutive** by looking at how `unique_copy` behaves on different inputs.

Figure 8.4 shows the result of `unique_copy` when applied to a short array of integers. The arrows indicate from which index in the array `a` the respective value in the array `b` originates. The gray portion in the target array indicates that in our example not all elements from `a` have been copied.

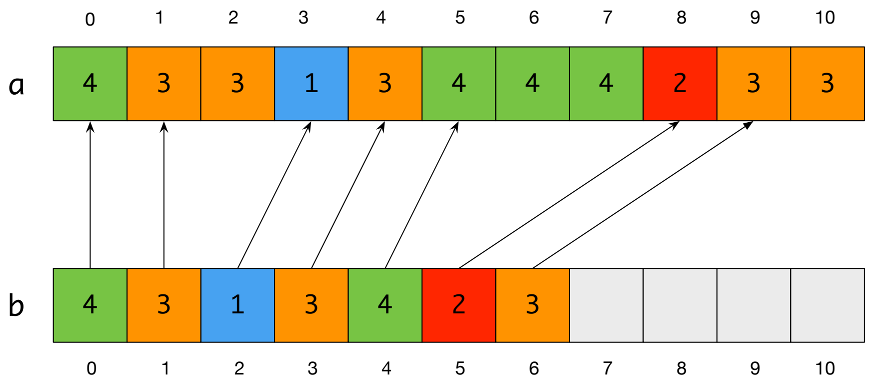


Figure 8.4.: Example of applying `unique_copy`

Requirement **Unique Copy Consecutive** also implies that if `unique_copy` is applied to sequence that contains no adjacent equal elements in the first place, then it behaves like an ordinary copy algorithms (Figure 8.5).

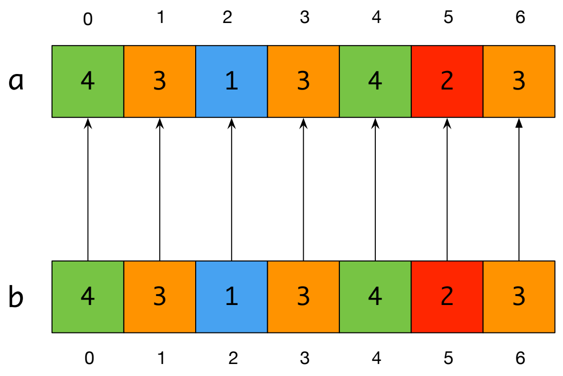


Figure 8.5.: Applying `unique_copy` to a sequence with no adjacent equal elements

Another, somewhat extreme, example is applying `unique_copy` to a sequence where all elements are equal to each other. In this case, the result of `unique_copy` will consists of a single value (Figure 8.6).

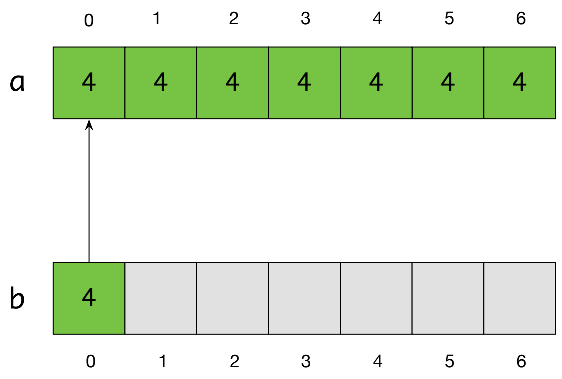


Figure 8.6.: Applying `unique_copy` to a sequence where all elements are equal

A typical use case of `unique_copy` is to apply it to a *sorted* sequence. In this case calling `unique_copy` ensures that each value of the input range occurs exactly once in the output range (Figure 8.7). This is of course known to `Unix` programmers who can use `sort FILE | uniq` to remove all duplicate lines from `FILE`.

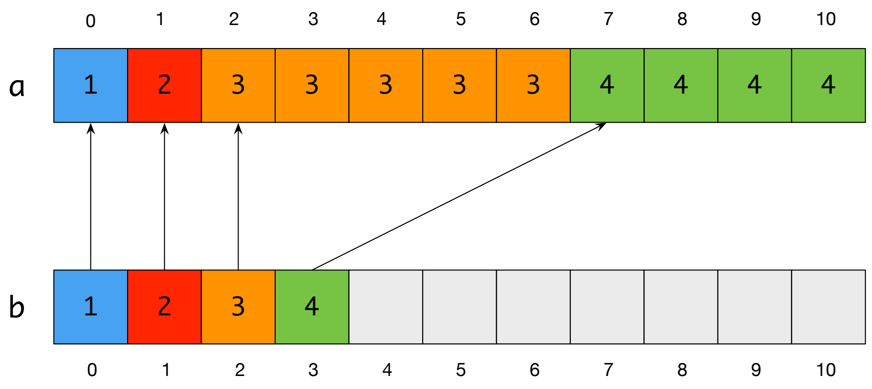


Figure 8.7.: Applying `unique_copy` to remove all duplicate elements from a sorted sequence

8.1.3. A first analysis of `unique_copy`

Figure 8.8 is a slight modification of Figure 8.4. We show here only the indices of the source array whose values are copied into the target array. In addition, we have added another (dashed) arrow to link the indices that correspond to the *one past the end* locations of the input and output ranges, respectively. We use this additional arrow in order to be able to describe all sub sequences of consecutive equal elements in the source array.

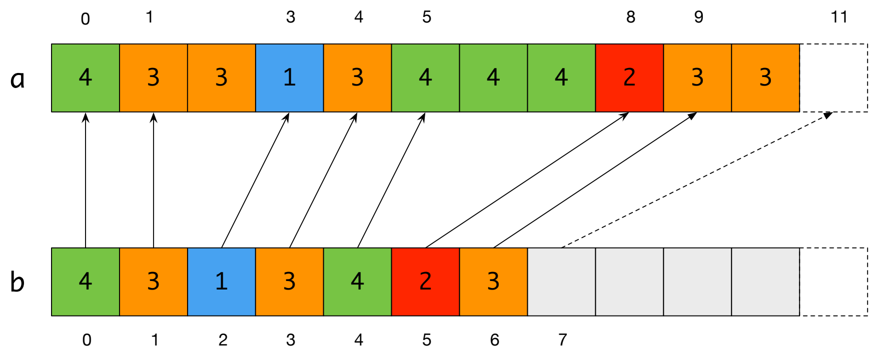


Figure 8.8.: Partitioning the input of `unique_copy`

These arrows between the indices of the array `b` and the array `a` define the following sequence p of eight indices where the index that points one past the end is underlined.

$$p = (0, 1, 3, 4, 5, 8, 9, \underline{11}) \qquad \text{for Figure 8.8}$$

For the other examples the corresponding index sequences are

$$p = (0, 1, 2, 3, 4, 5, 6, \underline{7}) \qquad \text{for Figure 8.5}$$
$$p = (0, \underline{7}) \qquad \text{for Figure 8.6}$$
$$p = (0, 1, 2, 7, \underline{11}) \qquad \text{for Figure 8.7}$$

Don't forget that the last three figures *do not show* the respective *one past the end* arrows.

More generally, we refer to the sequence p as *partitioning sequence* of `unique_copy` for the array `a[0..n-1]`. This sequence is characterized by the following properties: If $m + 1$ is the **length of a partitioning sequence**, then we observe that they are **strictly monotone increasing**

$$0 = p_0 < \ldots < p_{m+1} = n \tag{8.1}$$

and that the right-open index intervals

$$[p_i, p_{i+1}) \qquad \forall i : 0 \leq i < m$$

mark **consecutive ranges** of equal elements in the source array, that is,

$$\mathtt{a}[p_i] = \mathtt{a}[k] \qquad \forall k : p_i \leq k < p_{i+1} \tag{8.2}$$

We also have that the consecutive ranges are **maximal** in the following sense

$$\mathtt{a}[p_i] \neq \mathtt{a}[p_{i+1}] \qquad \forall i : 0 \leq i < m - 1 \tag{8.3}$$

and last but not least for the **result of `unique_copy`** it must hold

$$\mathtt{b}[i] = \mathtt{a}[p_i] \qquad \forall i : 0 \leq i < m \tag{8.4}$$

8.2. Unit tests for `unique_copy`

In this section we derive both *test data* and *test code* that shall establish that the implementation of `unique_copy` from Listing 8.2 satisfies the requirements listed in Table 8.3. We are mainly concerned with *unit tests* that capture the functionality of `unique_copy`. This is also the reason why we are not discussing the also important issue of *code coverage*.

We are dealing in this section with tests of a *generic* implementation, that is, with an implementation that is parameterized over the type of the elements stored in arrays. Our test code is therefore also as generic as suitable, see Section 8.2.1 for example. Of course, the actual test execution appears with both specific type parameters and concrete test data.

We start our presentation in Section 8.2.2 with a discussion of suitable test data for `unique_copy`. Thereby we also have a look at the test data for `unique_copy` in *libcxx* [24], an open source implementation of the C++ standard library.

Requirement **Unique Copy Consecutive** captures the core of `unique_copy`, and the design of our tests aims particularly at checking this requirement. This is, however, not an easy undertaking because it is not clear in the beginning how to convincingly demonstrate that the *first* (and only the first) element of a consecutive group of equal elements is copied. Our first tests `unique_copy` in Sections 8.2.3 and 8.2.4 (and the corresponding tests in *libcxx*) therefore only show that there are no adjacent equal elements in the output array.

There is, however, a not too complicated method to show the copying of the first element of the consecutive ranges. As we will show in Section 8.2.5 this method relies both on

1. our semi-formal analysis of the `unique_copy` in Section 8.1.3 and
2. the *generic* nature of the implementation of `unique_copy`.

The latter allows us to replace values of type `T` essentially by `std::pair<T,size_t>` where the `second` field of `std::pair` will hold the index of the copied element in the input sequence of `unique_copy`.

8.2.1. Preparation of test code

In order to facilitate the testing of `unique_copy` from Listing 8.2, we provide a wrapper implementation that uses the container `vector` from the C++ standard library.

```
template<typename T>
std::vector<T>
unique_copy(const std::vector<T>& a)
{
  std::vector<T> b(a.size());

  auto size = unique_copy(a.data(), a.size(), b.data());
  b.resize(size);

  return b;
}
```

Listing 8.9: `unique_copy` for `std::vector`

Using this generic auxiliary function from Listing 8.9 has several advantages for our tests.

- The `vector` container conveniently encapsulates both the memory and the number of elements of a C array.
- A `vector` hides many details of dynamic memory allocation from the user. Listing 8.2 shows that it is also easy to *resize* a `vector` object after it was created.
- The C++ standard ensures that different `vector` objects manage their own memory. Thus, using `vector`, it is easy to satisfy **Unique Copy Separation** which states that its two array arguments do not overlap.
- Also note that we initially declare the output `vector` to have the same size as the input `vector`. We are thus making sure that the requirement **Unique Copy Size** is satisfied.

8.2.2. Test data

Table 8.10 shows our initial test data for `unique_copy`. These are exactly the examples from Section 8.1.2 that have been used there to describe the behavior of `unique_copy`.

Input	Output	Reference
(4, 3, 3, 1, 3, 4, 4, 4, 2, 3, 3)	(4, 3, 1, 3, 4, 2, 3)	Figure 8.4
(4, 3, 1, 3, 4, 2, 3)	(4, 3, 1, 3, 4, 2, 3)	Figure 8.5
(4, 4, 4, 4, 4, 4, 4)	(4)	Figure 8.6
(1, 2, 3, 3, 3, 3, 3, 4, 4, 4)	(1, 2, 3, 4)	Figure 8.7

Table 8.10.: Initial test data

Boundary test data are data for extreme inputs of the specific algorithm. It can be argued that the example from Figure 8.6 where the elements of the input array equal *one* value represents boundary test data. Table 8.11 shows the expected behavior of `unique_copy` for input ranges of size 0 and size 1.

Input	Output	Reference
( )	( )	empty input range
(3)	(3)	one-element input range

Table 8.11.: Boundary test data

It is interesting to compare our test data with those from the functional tests for `unique_copy` in an open source implementation of the C++ standard library [24, `unique_copy.pass.cpp`]. We have listed these test data in Table 8.12.

Input	Output
(0, 1, 2, 2, 4)	(0, 1, 2, 4)
(0)	(0)
(0, 1)	(0, 1)
(0, 0)	(0)
(0, 0, 1)	(0, 1)
(0, 0, 1, 0)	(0, 1, 0)
(0, 0, 1, 1)	(0, 1)
(0, 0, 1)	(0, 1)
(0, 1, 1, 1, 2, 2, 2)	(0, 1, 2)

Table 8.12.: Test data for `unique_copy` from an open source test suite

Here the emphasis is on an arguably more systematic presentation of small input ranges of sizes. Interestingly, however, there is no test case for the empty range.

8.2.3. A basic test

When we present *test code* then we present code that shows that the function under test satisfies certain *properties* which in turn are justified by the requirements.

Listing 8.13, for example, contains code that tests that the elements copied by `unique_copy` contain no adjacent equal elements. This is, as we have explained in Section 8.1.2, a simple consequence of **Unique Copy Consecutive** from Table 8.3.

```
template<typename T>
std::vector<T>
unique_copy_basic_test(const std::vector<T>& input)
{
  auto result = unique_copy(input);
  assert(std::adjacent_find(result.begin(), result.end()) == result.end());

  //std::cout << "test " << __func__ << " succeeded " << std::endl;

  return result;
}
```

Listing 8.13: Testing the absence of adjacent equal elements

The key ingredient of the test in Listing 8.13 consists in calling the C++ standard library function `adjacent_find` which searches its input range for (the first) occurrence of two consecutive equal elements. If there is no such occurrence, `adjacent_find` returns the iterator that indicates the end of the range.

Listing 8.14 shows how the test from Listing 8.13 is executed.

```
int main(int argc, char** argv)
{
  assert(argc == 2);
  std::fstream file(argv[1]);
  std::vector<int> v;

  while (true) {
    file >> v;
    if (file) {
      // std::cout << v << std::endl;
      unique_copy_basic_test(v);
      v.clear();
    }
    else {
      break;
    }
  }

  std::cout << "\tsuccessful execution of " << argv[0] << "\n";

  return EXIT_SUCCESS;
}
```

Listing 8.14: Test execution code for the absence of adjacent equal elements

In our setting the test data step from the file in Listing 8.15 which contains the input data from Tables 8.10, 8.11, and 8.12.

```
(4, 3, 3, 1, 3, 4, 4, 4, 2, 3, 3)
(4, 3, 1, 3, 4, 2, 3)
(4, 4, 4, 4, 4, 4, 4)
(1, 2, 3, 3, 3, 3, 3, 4, 4, 4)

(3)
()

(0, 1, 2, 2, 4)
(0)
(0, 1)
(0, 0)
(0, 0, 1)
(0, 0, 1, 0)
(0, 0, 1, 1)
(0, 0, 1)
(0, 1, 1, 1, 2, 2, 2)
```

Listing 8.15: Test input data for `unique_copy`

8.2.4. Extending the basic test

This basic test can be extended to the slightly more elaborate test in Listing 8.16 that in addition to Listing 8.14 compares whether

1. the number of copied elements equals the size of the expected output range and
2. the copied elements actually equal the expected output range.

```
template<typename T>
void
unique_copy_compare_test(const std::vector<T>& input,
                         const std::vector<T>& expected)
{
  auto result = unique_copy_basic_test<T>(input);
  assert(result == expected);

  //std::cout << "test " << __func__ << " succeeded " << std::endl;
}
```

Listing 8.16: Comparing with expected result

Listing 8.17 shows how the test from Listing 8.16 is executed with the test data read from a file.

```
int main(int argc, char** argv)
{
  assert(argc == 2);
  std::fstream file(argv[1]);

  while (true) {
    std::vector<int> input, expected;
    file >> input;
    file >> expected;
    if (file) {
      // std::cout << input    << "\t" << expected << std::endl;
      unique_copy_compare_test(input, expected);
    }
    else {
      break;
    }
  }

  std::cout << "\tsuccessful execution of " << argv[0] << "\n";

  return EXIT_SUCCESS;
}
```

Listing 8.17: Test execution code for comparing with expected result

In this setting, the test data step from the file in Listing 8.18 which contains the input data and the expected output data from Tables 8.10, 8.11, and 8.12.

```
(4, 3, 3, 1, 3, 4, 4, 4, 2, 3, 3)  (4, 3, 1, 3, 4, 2, 3)
(4, 3, 1, 3, 4, 2, 3)              (4, 3, 1, 3, 4, 2, 3)
(4, 4, 4, 4, 4, 4, 4)              (4)
(1, 2, 3, 3, 3, 3, 3, 4, 4, 4      (1, 2, 3, 4)

()                                 ()
(3)                                (3)

(0, 1, 2, 2, 4)                    (0, 1, 2, 4)
(0)                                (0)
(0, 1)                             (0, 1)
(0, 0)                             (0)
(0, 0, 1)                          (0, 1)
(0, 0, 1, 0)                       (0, 1, 0)
(0, 0, 1, 1)                       (0, 1)
(0, 0, 1)                          (0, 1)
(0, 1, 1, 1, 2, 2, 2)              (0, 1, 2)
```

Listing 8.18: Test input and expected output for `unique_copy`

Executing the test from Listing 8.16 with the test data from Section 8.2.2 allows us to establish whether `unique_copy` satisfies the Requirement **Unique Copy Size** and the above mentioned consequence of Requirement **Unique Copy Consecutive**. However, these tests cannot establish that the *first* (and only the first) element of each consecutive group of equal elements is copied. In this sense, our test is as expressive as the tests for `unique_copy` from [24, `unique_copy.pass.cpp`]

8.2.5. Partition testing

In Section 8.1.3 we had, informally, argued that for each sequence $a = (a_0, \ldots, a_{n-1})$ there is a *partitioning sequence* $p = (p_0, \ldots, p_m)$ that satisfies the conditions (8.1), (8.2), and (8.3) and which, moreover, relates the input and output of `unique_copy` according to Equation (8.4). The term *partition testing* refers here to using these properties in the design of our tests.

The problem is that the partitioning sequence does not *explicitly* occur in `unique_copy`. There is, however, a relatively simple and natural way to make the partitioning sequence explicitly.

We explain the basic idea at hand of the input sequence

$$a = (4, 3, 3, 1, 3, 4, 4, 4, 2, 3, 3)$$

from Figure 8.8. We begin with creating a sequence of pairs (a_i, i) consisting of the original value a_i and its index i, in other words, we *zip* the sequence a with the sequence of its indices. In order to facilitate the readability of lists of pairs we also write $\begin{pmatrix} a_i \\ i \end{pmatrix}$ instead of (a_i, i).

For our example, the augmented sequence of pairs a' reads

$$a' = \left(\begin{pmatrix}4\\0\end{pmatrix}, \begin{pmatrix}3\\1\end{pmatrix}, \begin{pmatrix}3\\2\end{pmatrix}, \begin{pmatrix}1\\3\end{pmatrix}, \begin{pmatrix}3\\4\end{pmatrix}, \begin{pmatrix}4\\5\end{pmatrix}, \begin{pmatrix}4\\6\end{pmatrix}, \begin{pmatrix}4\\7\end{pmatrix}, \begin{pmatrix}2\\8\end{pmatrix}, \begin{pmatrix}3\\9\end{pmatrix}, \begin{pmatrix}3\\10\end{pmatrix}\right)$$

If we define the equality of two pairs (a_i, i) and (a_j, j) by the equality of its first components, then `unique_copy` *piggybacks* the original index of an element into the result. In other words, `unique_copy` applied to the sequence a' produces the following sequence

$$b' = \left(\begin{pmatrix}4\\0\end{pmatrix}, \begin{pmatrix}3\\1\end{pmatrix}, \begin{pmatrix}1\\3\end{pmatrix}, \begin{pmatrix}3\\4\end{pmatrix}, \begin{pmatrix}4\\5\end{pmatrix}, \begin{pmatrix}2\\8\end{pmatrix}, \begin{pmatrix}3\\9\end{pmatrix}\right)$$

where the second components indicate the index in the input sequence.

Finally, we extract the two lists of its first and second components from b' (that is we *unzip* b') and add a final element 11 (the number of elements of a) to the sequence of indices. We thus obtain the sequence

$$b = (4, 3, 1, 3, 4, 2, 3)$$

which is just the result of applying `unique_copy` to a, and the partitioning sequence

$$p = (0, 1, 3, 4, 5, 8, 9, 11)$$

Both sequences can, of course be found in Figure 8.8.

In the following subsections, we discuss the details of the implementation of our partitioning test.

Pairs of values and indices

Listing 8.19 shows our definition of a type for *indexed values* which consists of a pair of generic type `T` and `size_t` as index type. We implement this type with the generic class `std::pair` that has two public fields `first` and `second`, respectively.

The equality operation (**`operator`**`==`) of this new type has been defined in such a way that only the first component of the underlying pair type is evaluated. Note that we only provide a special implementation for the equality operation but not for copying and assigning a value `p` of type `Indexed<T>`. This is essential for keeping the values `p.first` and `p.second` unchanged while they are processed by our generic `unique_copy`.

```
template<typename T>
struct Indexed : public std::pair<T, size_t> {

  // inherit constructors of base class
  using std::pair<T, size_t>::pair;

};

template<typename T>
bool operator==(const Indexed<T>& a, const Indexed<T>& b)
{
  return a.first == b.first;
}
```

Listing 8.19: A type for indexed values

Creating the partition sequence

Listing 8.20 shows our implementation of computing the partitioning sequence of a given input sequence.

```
template<typename T>
std::pair<std::vector<T>, std::vector<size_t>>
    unique_copy_create_partition(const std::vector<T>& input)
{
  std::vector<Indexed<T>> zipped(input.size());
  for (size_t i = 0; i < input.size(); i++) {
    zipped[i] = Indexed<T>(input[i], i);
  }

  auto output = unique_copy(zipped);

  std::pair<std::vector<T>, std::vector<size_t>> unzipped;
  unzipped.first.resize(output.size());
  unzipped.second.resize(output.size() + 1);

  for (size_t i = 0; i < output.size(); ++i) {
    unzipped.first[i]  = output[i].first;
    unzipped.second[i] = output[i].second;
  }
  unzipped.second.back() = input.size();

  return unzipped;
}
```

Listing 8.20: Creating the partition underlying `unique_copy`

1. As explained at the beginning of this section we start with creating the sequence of pairs of values and their respective indices.

2. We then pass this sequence of pairs to our generic version of `unique_copy` for `vector` from Listing 8.9. The resulting sequence contains, thanks to our definition of equality of our pair type, the values with their indices in the original sequence.

3. Finally, we *unzip* the vector of pairs into a pair of vectors and add to the index vector the size of the input sequence as final element.

Testing the partition properties

Listing 8.21 shows our generic partition test. We have added comments to facilitate the tracing of our test code to the properties (8.1)–(8.4) from Section 8.1.3.

First `unique_copy` is called and the corresponding partition sequence is computed. Note that computing of this partition sequence naturally also leads to a second computation of the result of `unique_copy`. Thus, initially we check (for sanity) that both computations have the same result.

```
template<typename T>
void unique_copy_partition_test(const std::vector<T>& a)
{
  auto b = unique_copy(a);
  auto unzipped = unique_copy_create_partition(a);
  assert(unzipped.first == b);

  const std::vector<size_t>& p = unzipped.second;

  // partition sequence is one element longer than output array
  assert(p.size() == b.size() + 1);

  // monotonicity (first and last element only)
  assert(p.front() == 0);
  assert(p.back()  == a.size());

  for (size_t i = 0; i < b.size(); ++i) {
    // consider i-th segment of the partition
    auto begin = p[i];
    auto end   = p[i + 1];

    // monotonicity
    assert(begin < end);

    // consecutive range of equal elements
    for (size_t k = begin; k < end; ++k) {
      assert(a[begin] == a[k]);
    }

    // maximal consecutive range of equal elements
    if (i + 1 < b.size()) {
      assert(a[begin] != a[end]);
    }

    // result of unique_copy
    assert(b[i] == a[begin]);
  }
}
```

Listing 8.21: Partition testing of `unique_copy`

We then check whether the first element of partitioning sequence equals 0 and whether the last element equals the size of the input sequence. This is, of course, in accordance with the chain of (in)equalities from Relation 8.1. Finally, we check for each partition segment $[p_i, p_{i+1})$ the rest of the monotonicity conditions from Relation (8.1), and then proceed to verify the properties (8.2), (8.3), and (8.4).

Listing 8.22 shows the code for executing partition tests with test data read from a file. As test data we use again the inputs from Listing 8.15.

```
int main(int argc, char** argv)
{
  assert(argc == 2);
  std::fstream file(argv[1]);

  while (true) {
    std::vector<int> v;
    file >> v;
    if (file) {
      // std::cout << v << std::endl;
      unique_copy_partition_test(v);
    }
    else {
      break;
    }
  }

  std::cout << "\tsuccessful execution of " << argv[0] << "\n";

  return EXIT_SUCCESS;
}
```

Listing 8.22: Test execution code for partition tests

8.3. Formal verification of `unique_copy`

In this Section we discuss the formal verification of `unique_copy` with Frama-C/WP. The first issue is that while `std::unique_copy` is implemented in C++ and heavily relies on C++ templates, Frama-C/WP can only deal with C functions. For this reason we present in Section 8.3.1 an implementation of `unique_copy` in C.

We discuss our first and simplest version of an ACSL specification of `unique_copy` in Section 8.3.2. The main idea of a so-called *minimal contract* is that our formal specification is just strong enough to verify the absence of certain undefined behaviors, such as illegal memory accesses and integer overflows. More specifically this means that our minimal contract for `unique_copy` formalizes **Unique Copy Size** and **Unique Copy Separation** but only partially formalizes **Unique Copy Return**. The formalization of the core requirement **Unique Copy Consecutive** is not addressed at all. Still, the verification of a minimal contract is meaningful since the addressed undefined behaviors are often a cause for security vulnerabilities. The verification of the *absence* of these undesirable behaviors can play an important role for ensuring the robustness of software.

In Section 8.3.3, we extend the minimal contract from Section 8.3.2 by a postcondition that states that the output range of `unique_copy` will not contain any adjacent equal elements. In other words, the new contract also partially addresses **Unique Copy Consecutive**.

Finally, in Section 8.3.4 we present a further extension of our contract that also captures the missing aspects of **Unique Copy Return** and **Unique Copy Consecutive** in the specification of `unique_copy`. Here, we build on top of our analysis of the so-called *partitioning sequence* from Section 8.1.3.

In order to differentiate the those different variants of formal verification we number the functions. For example the functions `unique_copy2` and `unique_copy3` share basically the same implementation but have different formal annotations.

8.3.1. Reformulation of the algorithms in C

Our reformulation of `unique_copy` in the C programming language has a signature that can be considered as a specialisation of our simplified generic implementation of Listing 8.2. Instead of the generic type parameter `T`, however, we employ the integer type alias `value_type`. We also replace the standard unsigned integer type `size_t` by the type alias `size_type`. More information can be found in Section 2.3.

The basic idea of our C-implementation of `unique_copy` in Listing 8.23 is to traverse the input array `a[0...n-1]` and copy an element `a[i]` to the output array `b[0..n-1]` whenever it has been detected that it is different from its predecessor `a[i-1]`. Assuming a non-empty array, the implementation starts with copying `a[0]` to `b[0]`. In order to detect whether the current value `a[i]` is different from its predecessor we compare it with the most recently copied value `b[k]`.

```
size_type
unique_copy(const value_type* a, size_type n, value_type* b)
{
  if (0u < n) {

    size_type k = 0u;
    b[k] = a[0];

    for (size_type i = 1u; i < n; ++i) {
      const value_type val = a[i];

      if (b[k] != val) {
        b[++k] = val;
      }
    }

    return ++k;
  }
  else {
    return n;
  }
}
```

Listing 8.23: Re-implementation of `unique_copy` in C

Note that the test code in Section 8.2 has be designed in such a way that it can be applied also to a function with the signature in Listing 8.23.

8.3.2. A minimal contract for `unique_copy`

When we talk about a *minimal contract* of a function we mean a small contract that covers only basic properties. One might, for example, only be interested that during the execution of a function no runtime errors such as arithmetic overflows or invalid pointer accesses occur. Since many software security problems are caused by undetected runtime errors, minimal contracts can help to achieve a higher degree of quality assurance. This means that our minimal contract for `unique_copy` formalizes the requirements **Unique Copy Size** and **Unique Copy Separation** but only partially formalizes **Unique Copy Return**. The formalization of the core requirement **Unique Copy Consecutive** is not addressed at all.

Formal specification

Listing 8.24 shows the specification of our minimal contract. We have *labeled* the various preconditions and postconditions of our contract by names, e.g., we use the label `sep` in order to refer to our formal specification of **Unique Copy Separation**. Using these user-supplied labels simplifies the documentation of contracts and can also be helpful during the process of formal verification. In the following we often refer to the various formal properties in a contract by their labels.

```
/*@
  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 result:     0 <= \result <= n;
  ensures unchanged: Unchanged{Old,Here}(b, \result, n);
  ensures unchanged: Unchanged{Old,Here}(a, n);

*/
size_type
unique_copy2(const value_type* a, size_type n, value_type* b);
```

Listing 8.24: A “minimal“ contract for `unique_copy`

Using the built-in predicates `\valid` and `\valid_read`, the preconditions `valid` state that

1. the elements of the input range `a[0..n-1]` can be safely accessed for reading,
2. whereas the elements of the output range `b[0..n-1]` can be safely accessed both for reading and writing.

Note that in accordance with **Unique Copy Size** both ranges have the same size. The informal specification also states in **Unique Copy Separation** that the input and output ranges do not overlap. This precondition is expressed by our property `sep` which in turn uses the built-in predicate `\separated`.

The `assigns` clause of our minimal contract states that `unique_copy` can only modify the array `b [0..n-1]`. Note that this requirement on the side effects of `unique_copy` cannot be found in the requirements in Table 8.3. This can be attributed to the fact that little is known about internal side effects of the generic type parameter `T`. In our more specific situation with the concrete type `value_type` we can use the means of ACSL to restrict the side effects allowed by our contract.

The postcondition `result` describes the numerical range for the return value of `unique_copy`. In other words, our minimal contract only provides a rather coarse estimation of **Unique Copy Return** for the number of elements copied by `unique_copy`. Note the use the ACSL keyword `\result` in this postcondition to refer to the return value of function. Also note that we have employed a *chained inequality* instead of writing

```
0 <= \result && \result <= n
```

This is a nice, little feature that helps writing compact contracts. There is a second postcondition `unchanged` that is formulated using the *user defined* ACSL predicate `Unchanged` that we show in Listing 7.1. This predicate comes in the form of two overloaded versions. The first one is defined for an array section wheres the second one only requires the length of the array. The arguments `K` and `L` of the predicate are *labels* that represent *program states*. The predicate `Unchanged` says the respective elements of the array have the same value in state `K` and state `L`.

We use the predicate `Unchanged` in order to make the assigns clause a bit more precise. Using the predefined labels `Old`, which refers to the pre-state of the contract, and `Post`, which refers to the post-state of the contract, the postcondition `unchanged` says that element of the range `b[\result..n-1]` are the same before and after `unique_copy` has been called. All this would easier if we could use just the assigns clause

```
assigns b[0..\result-1];
```

since this would imply our postcondition

```
ensures unchanged: Unchanged{Old, Here}(b, \result, n);
```

We remark here that the ACSL documentation does not forbid the use of `\result` outside of `ensures` clauses [15, p. 30]. While Frama-C/WP does not reject it either, the corresponding proof obligations are, in any case, not verified.

Graphical presentation of the minimal contract

Figure 8.25 is an attempt to graphically represent the minimal contract from Listing 8.24.

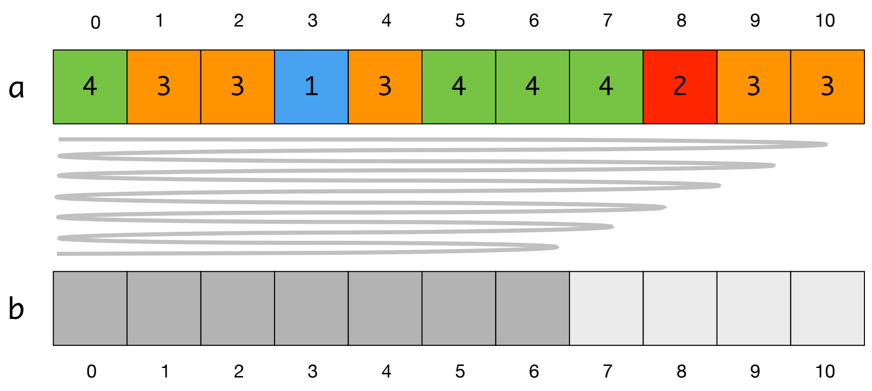


Figure 8.25.: Representation of a minimal contract for `unique_copy`

In contrast to Figure 8.4, it is not indicated which elements of the input array have been copied to which elements of the output array. This is because, it has not been specified, whether any element at all has been copied. On the other hand, the minimal contract ensures that the part of the output array, that is not needed to hold the result, is kept unchanged.

Annotating the implementation

The verification of the contract in Listing 8.24 requires that we add appropriate *loop annotations* to the implementation in Listing 8.26. Among these annotations are so-called *loop invariants*, which are formulas that must hold at the beginning of each loop iteration.

```
size_type
unique_copy2(const value_type* a, size_type n, value_type* b)
{
  if (0u < n) {
    size_type k = 0u;
    b[k] = a[0];

    /*@
      loop invariant bound:     0 <= k < i <= n;
      loop invariant unchanged: Unchanged{Pre,Here}(b, k+1, n);
      loop assigns i, k, b[0..n-1];
      loop variant n-i;
    */
    for (size_type i = 1u; i < n; ++i) {
      const value_type val = a[i];

      if (b[k] != val) {
        b[++k] = val;
      }
    }

    return ++k;
  }
  else {
    return n;
  }
}
```

Listing 8.26: Annotations for the minimal contract

We now have a closer look at the loop annotations.

- The loop invariant `bound` states that the index `k` is always less than `i` that both are limited from below and above by `0` and the array length `n`, respectively.
- Similar to the postcondition `unchanged` in Listing 8.24 the loop invariant `unchanged` states that in each iteration of the loop the values in the range `b[k+1..n]` will be the same as in the pre-state of `unique_copy`. Note the use of the predefined label `Pre` to denote the program state before the function was called.
- We also need a *loop assigns clause* which lists all memory locations that can be changed during the execution of the loop. These memory locations comprise not only the array `b[0..n-1]` but also the local variables `i` and `k`.
- Finally, we have a *loop variant* which must contain a positive value that is decreased in each loop iteration. Loop variants serve to verify the *termination* of loops.

Understanding the verification of minimal contracts

The WP plugin [25] of Frama-C (in short also Frama-C/WP) is activated by using the option `-wp`. However, before Frama-C/WP starts generating and discharging proof obligations, the Frama-C kernel produces a *normalized version* of the source code. Most Frama-C plugins, including Frama-C/WP, use this semantically equivalent presentation to conduct their respective analyses.

```
/*@ 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));
    ensures result: 0 <= \result <= \old(n);
    ensures unchanged: Unchanged{Old, Here}(\old(b), \result, \old(n));
    ensures unchanged: Unchanged{Old, Here}(\old(a), \old(n));
    assigns *(b + (0 .. n - 1));
 */
size_type unique_copy2(value_type const *a, size_type n, value_type *b)
{
  size_type __retres;
  if (0u < n) {
    size_type k = 0u;
    *(b + k) = *(a + 0);
    {
      size_type i = 1u;
      /*@ loop invariant bound: 0 <= k < i <= n;
          loop invariant unchanged: Unchanged{Pre, Here}(b, k + 1, n);
          loop assigns i, k, *(b + (0 .. n - 1));
          loop variant n - i;
      */
      while (i < n) {
        {
          value_type const val = *(a + i);
          if (*(b + k) != val) {
            k ++;
            *(b + k) = val;
          }
        }
        i ++;
      }
    }
    k ++;
    __retres = k;
    goto return_label;
  }
  else {
    __retres = n;
    goto return_label;
  }
  return_label: return __retres;
}
```

Listing 8.27: Normalized presentation of the minimal 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));
    ensures result: 0 <= \result <= \old(n);
    ensures unchanged: Unchanged{Old, Here}(\old(b), \result, \old(n));
    ensures unchanged: Unchanged{Old, Here}(\old(a), \old(n));
    assigns *(b + (0 .. n - 1));
 */
size_type unique_copy2(value_type const *a, size_type n, value_type *b)
{
  size_type __retres;
  if (0u < n) {
    size_type k = 0u;
    /*@ assert rte: mem_access: \valid(b + k); */
    /*@ assert rte: mem_access: \valid_read(a + 0); */
    *(b + k) = *(a + 0);
    {
      size_type i = 1u;
      /*@ loop invariant bound: 0 <= k < i <= n;
          loop invariant unchanged: Unchanged{Pre, Here}(b, k + 1, n);
          loop assigns i, k, *(b + (0 .. n - 1));
          loop variant n - i;
      */
      while (i < n) {
        {
          /*@ assert rte: mem_access: \valid_read(a + i); */
          value_type const val = *(a + i);
          /*@ assert rte: mem_access: \valid_read(b + k); */
          if (*(b + k) != val) {
            /*@ assert rte: unsigned_overflow: k + 1 <= 4294967295; */
            k ++;
            /*@ assert rte: mem_access: \valid(b + k); */
            *(b + k) = val;
          }
        }
        /*@ assert rte: unsigned_overflow: i + 1 <= 4294967295; */
        i ++;
      }
    }
    /*@ assert rte: unsigned_overflow: k + 1 <= 4294967295; */
    k ++;
    __retres = k;
    goto return_label;
  }
  else {
    __retres = n;
    goto return_label;
  }
  return_label: return __retres;
}
```

Listing 8.28: Normalized presentation of the minimal contract with RTE assertions

Listing 8.27 shows the normalized version of the formal specification from Listing 8.24 and the annotated implementation from Listing 8.26.

In order to extract the normalized version, which is also shown in Frama-C GUI, one can use the option `-print`. One of the most visible differences between the original and the normalized form is that *for loops* are represented as *while loops*. For more details on the normalization process we refer to the respective section of the Frama-C manual [26, §5.3].

Using the option `-wp-rte`, the Frama-C/WP plugin allows to generate additional assertions that are placed as guards before potentially dangerous C constructs, such as pointer dereferencing of integer operations that might overflow. Listing 8.28 shows these additional assertions in the normalized version of `unique_copy` when using the options `-wp-rte -warn-unsigned-overflow -warn-unsigned-downcast`. For more details how to customize the generation of RTE (runtime error) guards we refer to the respective manuals [25, 27].

Verifying the minimal contract with the additional run time error assertions essentially shows that a large class of undefined behaviors cannot occur *if* the preconditions of the contract are satisfied. Since undefined behaviors often represent security vulnerabilities, even the verification of the minimal contract can, thus, provide significant evidence that the execution of a function such as `unique_copy` cannot cause security weaknesses.

8.3.3. A more elaborate contract for `unique_copy`

After using the minimal contract to prove the absence of undefined behavior we now show that there are no equal neighbors in the output array `b[0..\result-1]`. This property reflects an important consequence of **Unique Copy Consecutive**. It does, however, neither express the fact that only the first element of every consecutive range within the input array is copied nor does it state that the elements in the output range are at all related to those from the input range.

In order to formalize this new property we use the new predicate `HasEqualNeighbors` which we show in Listing 4.16. The predicate states that there exists an element in the range `a[0..n-1]` which is equal to its direct successor.

Formal specification

Listing 8.29 is an extension of our minimal contract. We keep all properties but also add the new postcondition `solitary` to our 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));
  assigns             b[0..n-1];
  ensures result:     0 <= \result <= n;
  ensures solitary:   !HasEqualNeighbors (b, \result);
  ensures unchanged: Unchanged{Old,Here}(b, \result, n);
  ensures unchanged: Unchanged{Old,Here}(a, n);
 */
size_type
unique_copy3(const value_type* a, size_type n, value_type* b);
```

Listing 8.29: A more elaborate contract for `unique_copy`

In order to formally express the new postcondition we use the negation of `HasEqualNeighbors` from Listing 4.16.

Graphical presentation of the more elaborate contract

Figure 8.30 is an attempt to graphically represent the more elaborate specification from Listing 8.29. Compared to Figure 8.25 our new figure highlights that neighbouring elements of the output array are not equal. Our figure, however, still not indicates which elements of the input array have been copied to which elements of the output array.

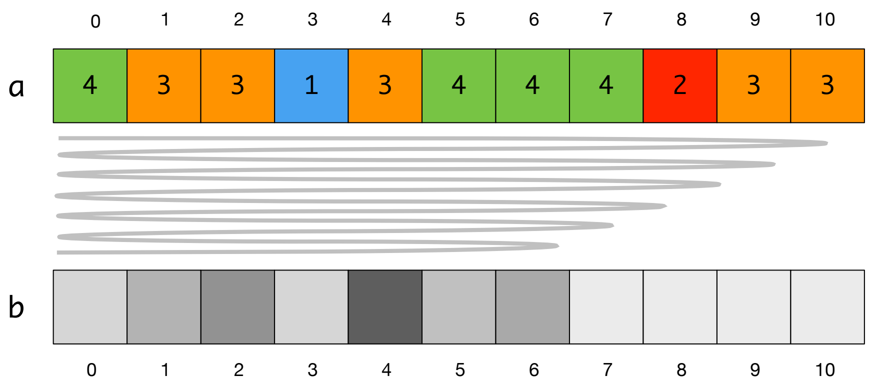


Figure 8.30.: Representation of a more elaborate contract for `unique_copy`

Annotating the implementation

In order to support the verification of the postcondition `unique` from Listing 8.29 we add the loop invariant `solitary` to our implementation 8.31. This loop invariant states that in each loop iteration there are no adjacent equal elements in the range `b[0..k]` of already copied elements. We use the predicate `HasEqualNeighbors` to formally describe this property.

```
size_type
unique_copy3(const value_type* a, size_type n, value_type* b)
{
  if (0u < n) {
    size_type k = 0u;
    b[k] = a[0];

    /*@
      loop invariant bound:     0 <= k < i <= n;
      loop invariant solitary:  !HasEqualNeighbors(b, k+1);
      loop invariant unchanged: Unchanged{Pre,Here}(b, k+1, n);
      loop assigns i, k, b[0..n-1];
      loop variant n-i;
    */
    for (size_type i = 1; i < n; ++i) {
      const value_type val = a[i];

      if (b[k] != val) {
        b[++k] = val;
      }
    }

    return ++k;
  }
  else {
    return n;
  }
}
```

Listing 8.31: Annotations for a more elaborate contract of `unique_copy`

8.3.4. A complete contract for `unique_copy`

In this section we finally tackle the issue of formalizing the requirements of **Unique Copy Consecutive** in ACSL. The main idea is that we

1. capture in ACSL the properties of the partitioning sequence from Section 8.1.3
2. use these properties to specify and verify `unique_copy`

Formalizing the number of elements copied by `unique_copy`

The logic function `UniqueSize` in Listing 8.32 computes the number of elements that are to be copied by `unique_copy` from an array `a[0..n-1]` In other words, `UniqueSize` represents the number *m* in the inequalities (8.1) of consecutive sub-ranges of equal elements. The Listing 8.32 contains explicit *recursive* definition of `UniqueSize` and the corresponding implicit definition through the lemmas.

```
/*@
  axiomatic UniqueSize
  {
    logic integer
    UniqueSize(value_type* a, integer n) =
      n <= 0 ? 0 : (n == 1 ? 1 : UniqueSize(a, n-1) + (a[n-1] == a[n-2] ? 0 : 1));

    lemma UniqueSizeEmpty:
      \forall value_type *a, integer n;
        n <= 0  ==>  UniqueSize(a, n) == 0;

    lemma UniqueSizeOne:
      \forall value_type *a;
        UniqueSize(a, 1) == 1;

    lemma UniqueSizeEqual:
      \forall value_type *a, integer n;
        0 < n  ==>  a[n-1] == a[n]  ==>  UniqueSize(a, n+1) == UniqueSize(a, n);

    lemma UniqueSizeDiffer:
      \forall value_type *a, integer n;
        0 < n  ==>  a[n-1] != a[n]  ==>  UniqueSize(a, n+1) == UniqueSize(a, n) + 1;

    lemma UniqueSizeRead{K,L}:
      \forall value_type *a, integer n, i;
        Unchanged{K,L}(a, n)  ==>  UniqueSize{K}(a, n) == UniqueSize{L}(a, n);

    lemma UniqueSizeBound:
      \forall value_type *a, integer n;
        0 <= n  ==>  0 <= UniqueSize(a, n) <= n;
  }
*/
```

Listing 8.32: The logic function `UniqueSize`

Note that the definition of `UniqueSize` also covers arrays of negative size and in general ignore whether the involved pointers can be dereferenced. The issue here is that the function `UniqueSize` must be defined as a *total function* regardless of the fact whether the involved function arguments make sense in C-code. For more details we refer to the rules for logic definitions in the description of ACSL [15, §2.2.2].

The acsl lemmas `UniqueSizeBound` from Listing 8.32 formulates some simple bounds on the number of copied elements. As trivial as these inequalities might look like, their not too complicated proofs rely on mathematical induction.

Formalizing the properties of the partitions of `unique_copy`

The function `UniquePartition`, whose axiomatic definition is given in Listing 8.33, defines the partitioning sequence *p* from Section 8.1.3. Before we begin to relate the axioms from Listing 8.33 to the formulas from Section 8.1.3 we want to remind the reader that logic functions (and predicates) must be total that is they must be defined for all possible argument values.

- The monotonicity conditions (8.1) are described by the axioms `UniquePartitionEmpty`, `UniquePartitionLeft`, `UniquePartitionRight` and `UniquePartitionMonotone`.
- Equation (8.2) is represented by the axiom `UniquePartitionSegment`.
- Inequality (8.3) is described by axiom `UniquePartitionMaximal`.
- Axiom `UniquePartitionEqual` expresses that the value of `UniquePartition(a, n, i)` does not depend on the size of the array.
- Axiom `UniquePartitionRead`, finally states that `UniquePartition` is independent from the particular programme state in which it is used—as long as the respective array elements are equal in both states.

With the definitions of the logic functions `UniqueSize` and `UniquePartition` we can now formulate the ACSL predicate `Unique` from Listing 8.34. This predicate reflects Equation (8.4) and therefore will serve a prominent role in our complete contract of `unique_copy`.

```
/*@
  axiomatic UniquePartition
  {
    logic integer
    UniquePartition(value_type* a, integer n, integer i) reads a[0..n-1];

    axiom UniquePartitionEmpty:
      \forall value_type *a, integer n, i;
        n <= 0  ==>  UniquePartition(a, n, i) == 0;

    axiom UniquePartitionLeft:
      \forall value_type *a, integer n, i;
        0 < n  ==>  i <= 0  ==>  UniquePartition(a, n, i) == 0;

    axiom UniquePartitionRight:
      \forall value_type *a, integer n, i;
        0 < n  ==>  UniqueSize(a, n) <= i  ==>  UniquePartition(a, n, i) == n;

    axiom UniquePartitionMonotone:
      \forall value_type *a, integer n, i, j;
        0 <= i < j <= UniqueSize(a, n)  ==>
        UniquePartition(a, n, i) < UniquePartition(a, n, j);

    axiom UniquePartitionSegment:
      \forall value_type *a, integer n, i, k;
        0 <= i < UniqueSize(a, n)  ==>
        AllEqual(a, UniquePartition(a, n, i), UniquePartition(a, n, i+1));

    axiom UniquePartitionMaximal:
      \forall value_type *a, integer n, i;
        0 <= i < UniqueSize(a, n) - 1  ==>
        a[UniquePartition(a, n, i)] != a[UniquePartition(a, n, i+1)];

    axiom UniquePartitionEqual:
      \forall value_type *a, integer n, m, i;
        n < m  ==>  0 <= i < UniqueSize(a, n)  ==>
        UniquePartition(a, n, i) == UniquePartition(a, m, i);

    axiom UniquePartitionRead{K,L}:
      \forall value_type *a, integer n, i;
        Unchanged{K,L}(a, n)  ==>
          UniquePartition{K}(a, n, i) == UniquePartition{L}(a, n, i);

    lemma UniquePartitionZero:
      \forall value_type *a, integer n;
        UniquePartition(a, n, 0) == 0;

    lemma UniquePartitionLowerBound:
      \forall value_type *a, integer n, i;
        0 < n  ==> 0 <= i < UniqueSize(a, n)  ==> 0 <= UniquePartition(a, n, i);

    lemma UniquePartitionUpperBound:
      \forall value_type *a, integer n, i;
        0 < n  ==> 0 <= i < UniqueSize(a, n)  ==>  UniquePartition(a, n, i) < n;

    lemma UniquePartitionDiffer:
      \forall value_type *a, integer i, k, n;
        UniquePartition(a, n, k-1) < i <= UniquePartition(a, n, k)  ==>
        a[i-1] != a[i]  ==>  i == UniquePartition(a, n, k);
  }
*/
```

Listing 8.33: Axiomatic description of the function `UniquePartition`

```
/*@
  axiomatic Unique
  {
    predicate
    Unique(value_type* a, integer n, value_type* b) =
      \forall integer k; 0 <= k < UniqueSize(a, n)  ==>
        b[k] == a[UniquePartition(a, n, k)];

    lemma UniqueImpliesNoEqualNeighbors:
      \forall value_type *a, *b, integer n;
         Unique(a, n, b)  ==>  !HasEqualNeighbors(b, UniqueSize(a, n));

    lemma UniquePreserve{K,L}:
      \forall value_type *a, *b, integer i, k, n;
        0 <= k <= i < n           ==>
        k == UniqueSize{L}(a, i)  ==>
        Unique{K}(a, i, b)        ==>
        Unchanged{K,L}(b, k)      ==>
        Unchanged{K,L}(a, n)      ==>
        Unique{L}(a, i, b);
  }
*/
```

Listing 8.34: The predicate Unique

Formal specification

Listing 8.35 shows how we use the predicate Unique in the postcondition unique in order to formally specify **Unique Copy Consecutive** for unique_copy.

Note that we use the Lemma UniqueImpliesNoEqualNeighbors from Listing 8.34 in order to establish the postcondition solitary from the postcondition unique. Thanks to this lemma we will be able to omit the dedicated loop annotation solitary in Listing 8.36.

```
/*@
  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 == UniqueSize(a, n);
  ensures unique:    Unique(a, n, b);
  ensures solitary:  !HasEqualNeighbors (b, \result);
  ensures unchanged: Unchanged{Old,Here}(a, n);
  ensures unchanged: Unchanged{Old,Here}(b, \result, n);
 */
size_type
unique_copy4(const value_type* a, size_type n, value_type* b);
```

Listing 8.35: A complete contract for unique_copy

We mention here also lemma UniquePreserve from Listing 8.34. This lemma indicates that between two memory states K and L, the predicate Unique holds for the later state L if it already held in the first state K and the value of k equals the value of UniqueSize(a, i) in state L. Also the values in the ranges b[0..k] and a[0..n] must still be unchanged in both states.

Annotating the implementation

Listing 8.36 shows that we need considerably more annotation in order to verify the contract from Listing 8.35. As mentioned before, however, we can omit a dedicated loop invariant to verify the postcondition solitary.

```
size_type
unique_copy4(const value_type* a, size_type n, value_type* b)
{
  if (0u < n) {
    size_type k = 0u;
    b[k] = a[0];
    //@ assert mapping:   0 == UniquePartition(a, n, k);

    /*@
      loop invariant bound:     0 <= k < i <= n;
      loop invariant size:      k+1 == UniqueSize(a, i);
      loop invariant copy:      b[k] == a[i-1];
      loop invariant mapping:   UniquePartition(a, n, k) < i;
      loop invariant mapping:   i <= UniquePartition(a, n, k+1);
      loop invariant unique:    Unique(a, i, b);
      loop invariant unchanged: Unchanged{Pre,Here}(a, n);
      loop invariant unchanged: Unchanged{Pre,Here}(b, k+1, n);
      loop assigns i, k, b[0..n-1];
      loop variant n-i;
    */
    for (size_type i = 1u; i < n; ++i) {
      const value_type val = a[i];

      if (b[k] != val) {
        //@ assert distinct:  a[i-1] != a[i];
        b[++k] = val;
        //@ assert unchanged: Unchanged{LoopCurrent,Here}(b, k);
        //@ assert unchanged: Unchanged{LoopCurrent,Here}(a, n);
        //@ assert unchanged: Unchanged{LoopCurrent,Here}(a, i);
        //@ assert mapping:   i == UniquePartition(a, n, k);
        //@ assert size:      k == UniqueSize(a, i);
        //@ assert unique:    Unique(a, i, b);
      }

      //@ assert mapping:   i <= UniquePartition(a, n, k+1);
      //@ assert unchanged: Unchanged{Pre,Here}(b, k+1, n);
    }

    return ++k;
  }
  else {
    return n;
  }
}
```

Listing 8.36: Annotations for the complete contract of unique_copy

9. 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 [20, §29.8] that use arithmetic operations on `value_type` to combine the elements of sequences.

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

Listing 9.1: Limits of `value_type`

In order to refer to potential arithmetic overflows we introduce the two constants shown in Listing 9.1 which refer to the numeric limits of `value_type` (see also Section 2.3).

We consider the following algorithms.

- `iota` writes sequentially increasing values into a range (Section 9.1 on Page 166)
- `accumulate` computes the sum of the elements in a range (Section 9.2 on Page 168)
- `inner_product` computes the inner product of two ranges (Section 9.3 on Page 171)
- `partial_sum` computes the sequence of partial sums of a range (Section 9.4 on Page 174)
- `adjacent_difference` computes the differences of adjacent elements in a range (Section 9.5 on Page 176)
- Finally, In Section9.6 we show that under appropriate preconditions the algorithms `partial_sum` and `adjacent_difference` are inverse to each other.

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

9.1. The iota algorithm

The iota algorithm in the C++ Standard Library [20, §29.8.12] assigns sequentially increasing values to a range, where the initial value is user-defined. Our version of the original signature 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.

9.1.1. Formal specification of iota

The specification of iota relies on the logic function IotaGenerate that is defined in Listing 9.2.

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

Listing 9.2: The logic function IotaGenerate

The ACSL specification of iota is shown in Listing 9.3. It uses the logic function IotaGenerate 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: IotaGenerate(a, n, val);
*/
void
iota(value_type* a, size_type n, value_type val);
```

Listing 9.3: 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 9.1). In order to avoid integer overflows the sum val+n must not be greater than the constant VALUE_TYPE_MAX.

9.1.2. Implementation of `iota`

Listing 9.4 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: IotaGenerate(a, i, \at(val, Pre));

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

Listing 9.4: 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`.

9.2. The `accumulate` algorithm

The `accumulate` algorithm in the C++ Standard Library [20, §29.8.2] computes the sum of an given initial value and the elements in a range. Our version of the original signature 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.

9.2.1. The logic function `Accumulate`

As in the case of `count` (see Section 4.10) we specify `accumulate` by first defining a *logic function* `Accumulate` that formally defines the summation of elements in an array.

```
/*@
  axiomatic Accumulate
  {
    logic integer
    Accumulate{L}(value_type* a, integer n, integer init) =
      n <= 0 ? init : Accumulate(a, n-1, init) + a[n-1];

    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;

    lemma AccumulateRead{K,L}:
      \forall value_type *a, integer n, init;
        Unchanged{K,L}(a, n)  ==>
        Accumulate{K}(a, n, init) == Accumulate{L}(a, n, init);
  }
*/
```

Listing 9.5: The logic function `Accumulate`

With this definition the following equation holds for $n \geq 0$

$$\texttt{Accumulate}(\texttt{a}, \texttt{n}, \texttt{init}) = \texttt{init} + \sum_{i=0}^{\texttt{n}-1} \texttt{a}[i] \tag{9.1}$$

The lemma `AccumulateRead` in Listing 9.5 express that the result of the `Accumulate` function only depends on the values of `a[0..n-1]`.

Listing 9.5 also contains the predicate `AccumulateBounds` that we will subsequently use in order to compactly express requirements that exclude numeric overflows while accumulating value. This predicate expresses that for $0 \le i < n$ the *partial sums*

$$\texttt{init} + \sum_{k=0}^{\texttt{i}} \texttt{a}[k] \tag{9.2}$$

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

9.2.2. `AccumulateDefault` — a variant of `Accumulate`

Listing 9.6 shows another version of `Accumulate`, called `AccumulateDefault`, that uses `a[0 ]` as default value of `init`. Thus, for `AccumulateDefault` we have

$$\texttt{AccumulateDefault}(\texttt{a},\texttt{n}) = \sum_{i=0}^{\texttt{n}-1} \texttt{a}[i] \tag{9.3}$$

We will use this version for the specification of the algorithm `partial_sum` (see Section 9.4).

```
/*@
  axiomatic AccumulateDefault
  {
    logic integer
    AccumulateDefault{L}(value_type* a, integer n) =
      Accumulate(a+1, n, (value_type)(a[0]));

    predicate
    AccumulateDefaultBounds{L}(value_type* a, integer n) =
      \forall integer i; 0 <= i < n  ==>
        VALUE_TYPE_MIN <= AccumulateDefault(a, i) <= VALUE_TYPE_MAX;

    lemma AccumulateDefaultRead{K,L}:
      \forall value_type *a, integer n;
        0 <= n               ==>
        Unchanged{K,L}(a, n+1)  ==>
        AccumulateDefault{K}(a, n) == AccumulateDefault{L}(a, n);

    lemma AccumulateDefault_Zero{L}:
      \forall value_type* a; AccumulateDefault(a, 0) == a[0];

    lemma AccumulateDefault_One{L}:
      \forall value_type* a; AccumulateDefault(a, 1) == a[0] + a[1];

    lemma AccumulateDefault_Next{L}:
      \forall value_type* a, integer n;
        0 <= n  ==>
        AccumulateDefault(a, n+1) == AccumulateDefault(a, n) + a[n+1];
  }
*/
```

Listing 9.6: The predicate `AccumulateDefault`

This listing also includes additional properties of observable `AccumulateDefault` behavior, here given as a lemmas. These lemmas are proved automatically based on the definitions from Listing 9.5. This listing also contains the predicate `AccumulateDefaultBounds` with corresponding numeric limits for the predicate `AccumulateDefault`.

9.2.3. 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 9.7.

```
/*@
  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 9.7: Formal specification of `accumulate`

9.2.4. Implementation of accumulate

Listing 9.8 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 = 0u; i < n; ++i) {
    //@ assert rte_help: init + a[i] == Accumulate(a, i+1, \at(init,Pre));
    init = init + a[i];
  }

  return init;
}
```

Listing 9.8: Implementation of `accumulate`

Note that loop invariant `partial` claims that in the *i*-th iteration step `result` equals the accumulated value of Equation (9.2). This depends on the property `bounds` in Listing 9.7 which expresses that there is no numeric overflow when updating the variable `init`.

9.3. The `inner_product` algorithm

The `inner_product` algorithm in the C++ Standard Library [20, §29.8.4] computes the *inner product*[27] of two ranges. Our version of the original signature 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

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

thus, `inner_product` will return `init` for empty ranges.

9.3.1. The logic function `InnerProduct`

As in the case of `accumulate` (see Section 9.2) we specify `inner_product` by first defining a *logic function* `InnerProduct` that formally defines the summation of the element-wise product of two arrays. Lemma `InnerProductRead` express that the result of the `InnerProduct` only depends on the values of `a[0..n-1]` and `b[0..n-1]`.

```
/*@
  axiomatic InnerProduct
  {
    logic integer
    InnerProduct{L}(value_type* a, value_type* b, integer n,
                    value_type init) =
      n <= 0 ? init : InnerProduct(a, b, n-1, init) + (a[n-1] * b[n-1]);

    lemma InnerProductRead{K,L}:
      \forall value_type *a, *b, init, integer n;
        Unchanged{K,L}(a, n)  ==>
        Unchanged{K,L}(b, n)  ==>
        InnerProduct{K}(a, b, n, init) == InnerProduct{L}(a, b, n, init);

    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 9.9: The logic function `InnerProduct`

Before we present our formal specification of `inner_product` we shortly discuss two more predicates from Listing 9.9. We will use them subsequently in order to compactly express requirements that exclude

[27] Also referred to as *dot product*, see `http://en.wikipedia.org/wiki/Dot_product`

numeric overflows while computing the inner product. Predicate `ProductBounds` expresses that for $0 \le i < n$ the products

$$\mathsf{a}[i] \cdot \mathsf{b}[i] \tag{9.4}$$

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

$$\texttt{init} + \sum_{k=0}^{\texttt{i}} \mathsf{a}[k] \cdot \mathsf{b}[k] \tag{9.5}$$

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

9.3.2. Formal specification of `inner_product`

Using the logic function `InnerProduct`, we specify `inner_product` as shown in Listing 9.10. 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{Old,Here}(a, n);
  ensures unchanged: Unchanged{Old,Here}(b, n);
*/
value_type
inner_product(const value_type* a, const value_type* b, size_type n,
              value_type init);
```

Listing 9.10: Formal specification of `inner_product`

9.3.3. Implementation of inner_product

Listing 9.11 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 = 0u; i < n; ++i) {
    /*@
      assert rte_help: init + a[i] * b[i] ==
                       InnerProduct(a, b, i+1, \at(init,Pre));
    */
    init = init + a[i] * b[i];
  }

  return init;
}
```

Listing 9.11: 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 (9.5). This depends of course on the properties `bounds` in Listing 9.10, which express that there is no arithmetic overflow when computing the updates of the variable `init`.

9.4. The `partial_sum` algorithm

The `partial_sum` algorithm in the C++ Standard Library [20, §29.8.6] computes the sum of a given initial value and the elements in a range. Our version of the original signature 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

$$\mathsf{b}[i] = \sum_{k=0}^{\mathtt{i}} \mathsf{a}[k] \tag{9.6}$$

Equations (9.6) and (9.3) suggest that we define the ACSL predicate `PartialSum` in Listing 9.12 by using the logic function `AccumulateDefault` from Listing 9.6.

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

    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;
        0 <= n                            ==>
        PartialSum(a, n, b)               ==>
        b[n] == AccumulateDefault(a, n)   ==>
        PartialSum(a, n+1, b);
  }
*/
```

Listing 9.12: The predicate `PartialSum`

9.4.1. Formal specification of `partial_sum`

Using the predicates `PartialSum` and `AccumulateDefaultBounds` (Listing 9.6), we specify `partial_sum` as shown in Listing 9.13.

```
/*@
  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 bounds:    AccumulateDefaultBounds(a, n);
  assigns             b[0..n-1];
  ensures result:     \result == n;
  ensures partialsum: PartialSum(a, n, b);
  ensures unchanged:  Unchanged{Old,Here}(a, n);
*/
size_type
partial_sum(const value_type* a, size_type n, value_type* b);
```

Listing 9.13: 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*. In order to simplify verification of `partial_sum`, we provide in Listing 9.12 several lemmas.

9.4.2. Implementation of `partial_sum`

Listing 9.14 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 (0u < n) {
    b[0u] = a[0u];

    /*@
       loop invariant bound:      1 <= i <= n;
       loop invariant unchanged:  Unchanged{Pre,Here}(a, n);
       loop invariant accumulate: b[i-1] == AccumulateDefault(a, i-1);
       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) {
      b[i] = b[i - 1u] + a[i];
      //@ assert unchanged:  a[i] == \at(a[i],LoopCurrent);
      //@ assert unchanged:  Unchanged{LoopCurrent,Here}(a, i);
      //@ assert unchanged:  Unchanged{LoopCurrent,Here}(b, i);
    }
  }

  return n;
}
```

Listing 9.14: Implementation of `partial_sum`

9.5. The `adjacent_difference` algorithm

The `adjacent_difference` algorithm in the C++ Standard Library [20, §29.8.11] computes the differences of adjacent elements in a range. Our version of the original signature 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

$$\begin{aligned}
\mathsf{b}[0] &= \mathsf{a}[0] \\
\mathsf{b}[1] &= \mathsf{a}[1] - \mathsf{a}[0] \\
&\vdots \\
\mathsf{b}[n-1] &= \mathsf{a}[n-1] - \mathsf{a}[n-2]
\end{aligned} \tag{9.7}$$

9.5.1. The predicate `AdjacentDifference`

We define the predicate `AdjacentDifference` in Listing 9.16 with the help of logic function `Difference` whose definition is shown in Listing 9.15.

```
/*@
  axiomatic Difference
  {
    logic integer
    Difference{L}(value_type* a, integer n) =
      n <= 0 ? a[0] : a[n] - a[n-1];

    lemma Difference_Zero{L}:
      \forall value_type *a; Difference(a, 0) == a[0];

    lemma Difference_Next{L}:
      \forall value_type *a, integer n;
        0 < n  ==>  Difference(a, n) == a[n] - a[n-1];

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

Listing 9.15: The logic function `Difference`

We introduce Listing 9.16 also the predicate `AdjacentDifferenceBounds` that captures conditions that prevent numeric overflows while computing difference of the form `a[i] - a[i-1]`. The lemmas shown in Listing 9.16 are also needed for the verification of the algorithm in Section 9.6.

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

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

    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, n+1, 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 9.16: The predicate AdjacentDifference

9.5.2. Formal specification of adjacent_difference

Using the predicates AdjacentDifference and AdjacentDifferenceBounds we can provide a concise formal specification of adjacent_difference (Listing 9.17). 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 sep:        \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{Old,Here}(a, n);
*/
size_type
adjacent_difference(const value_type* a, size_type n, value_type* b);
```

Listing 9.17: Formal specification of adjacent_difference

9.5.3. Implementation of `adjacent_difference`

Listing 9.18 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 9.16
- 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 (0u < n) {
    b[0u] = a[0u];

    /*@
       loop invariant index:      1 <= i <= n;
       loop invariant unchanged:  Unchanged{Pre,Here}(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(a, i+1, b);
    }
  }

  return n;
}
```

Listing 9.18: Implementation of `adjacent_difference`

9.6. Inverting `partial_sum` and `adjacent_difference`

In this section we show that under appropriate preconditions the algorithms `partial_sum` and `adjacent_difference` are inverse to each other.

9.6.1. Inverting `partial_sum`

Let `a[0..n-1]` and `b[0..n-1]` be the respective input and output of `partial_sum`. We have in other words

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

If we apply now the algorithm `adjacent_difference` to `b[0..n-1]`, we find for its output `a'[0..n-1]`

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

In other words the algorithms `partial_sum` and `adjacent_difference` are inverse to each other. In this current section, we are going to verify this claims with the help of Frama-C, viz. that applying `adjacent_difference` to the output of `partial_sum` produces an array that is equal to the original array. Lemma `PartialSumInverse` from Listing 9.19 expresses our claim as an ACSL lemma.

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

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

Listing 9.19: The lemmas `PartialSumInverse` and `AdjacentDifferenceInverse`

Since the lemma does not deal with arithmetic overflows or potential aliasing of data, we introduce an auxiliary C function `partial_sum_inv` which takes these issues into account. In particular, the C function uses the predicate `DefaultBounds` from Listing 9.20 in order to express that the values in the input (and output!) array `a[0..n-1]` do not overflow.

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

Listing 9.20: The predicate `DefaultBounds`

Listing 9.21 now shows the contract and implementation of the auxiliary C function `partial_sum_inv`. This function calls first `partial_sum` and then `adjacent_difference`.

```
/*@
  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));
  requires bounds:    AccumulateDefaultBounds(a, n);
  requires bounds:    DefaultBounds(a, n);
  assigns             a[0..n-1], b[0..n-1];
  ensures unchanged:  Unchanged{Pre,Here}(a, n);
*/
void
partial_sum_inv(value_type* a, size_type n, value_type* b)
{
  partial_sum(a, n, b);
  adjacent_difference(b, n, a);
}
```

Listing 9.21: `partial_sum` and then `adjacent_difference`

The contract of this function formulates preconditions that shall guarantee that during the computation neither arithmetic overflows (property `bounds`) nor unintended aliasing of arrays (property `sep`) occur. Under these precondition, Frama-C shall verify that the final `adjacent_difference` call just restores the original contents of `a[0..n-1]` used for the initial `partial_sum` call.

9.6.2. Inverting `adjacent_difference`

After executing the function `adjacent_difference` on the input array `a[0..n-1]` the output 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 call now partial_sum with the array b[0..n-1] as input, then we obtain for its output a'[0..n-1]

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

which means that applying partial_sum on the output of adjacent_difference produces the original input of adjacent_difference. Lemma AdjacentDifferenceInverse from Listing 9.19 expresses this property as an ACSL lemma.

As in the case discussed of partial_sum_inv, we give a corresponding C function in order to account for possible arithmetic overflows and potential aliasing of data. Function adjacent_difference_inv, shown in Listing 9.22, 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 sep) occur.

```
/*@
  requires size:      0 <= n;
  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));
  requires bounds:    DefaultBounds(a, n);
  requires bounds:    AdjacentDifferenceBounds(a, n);
  assigns             a[0..n-1], b[0..n-1];
  ensures unchanged:  Unchanged{Old,Here}(a, n);
*/
void
adjacent_difference_inv(value_type* a, size_type n, value_type* b)
{
  adjacent_difference(a, n, b);
  partial_sum(b, n, a);
}
```

Listing 9.22: adjacent_difference and then partial_sum

In order to improve the automatic verification rate of the function adjacent_difference_inv we also use lemma Unchanged_Transitive from Listing 7.1.

Part IV.

Sorting algorithms

10. Heap Algorithms

The heap algorithms of the C++ Standard Library [20, 28.7.7] 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 [28].

The C++ standard[28] 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 an increasing sequence.

Figure 10.1 gives an overview on the five heap algorithms by means of an example. Algorithms, which in a typical implementation are in a caller-callee relation, have the same color.

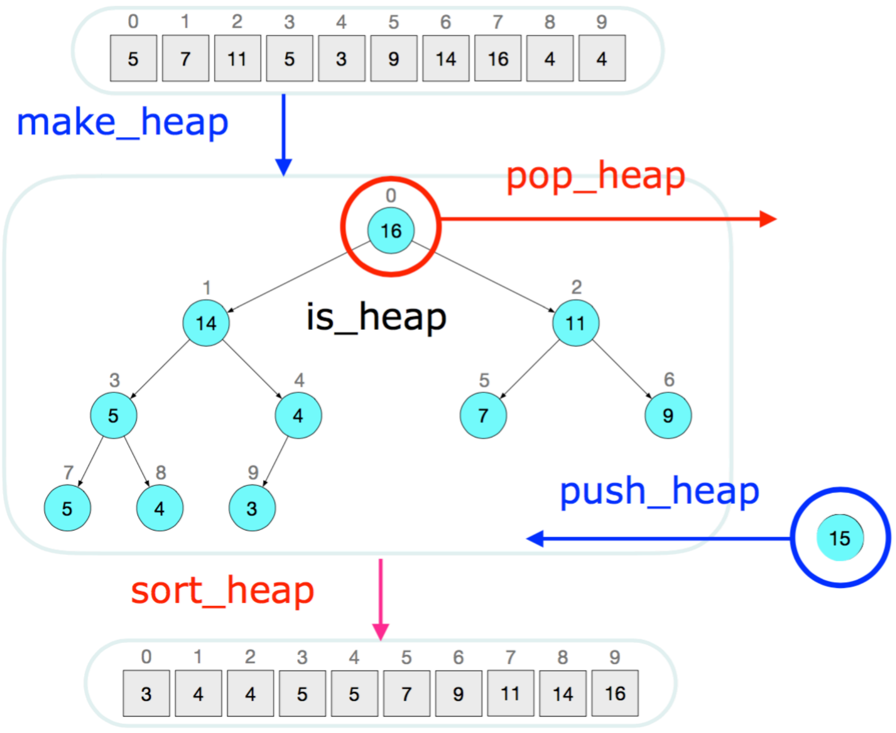


Figure 10.1.: Overview on heap algorithms

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

Roughly speaking, the algorithms from Figure 10.1 have the following behavior.

- In Section 10.1 we briefly recapitulate basic heap concepts.
- In Section 10.2 we show how these heap concepts can be described in ACSL.
- In Section 10.3 we verify two auxiliary heap functions.
- The algorithms `is_heap_until` and `is_heap` from Sections 10.4 and 10.5 allow to test at run time whether a given array is arranged as a heap
- The algorithm `push_heap` from Section 10.6 *adds* an element to a given heap in such a way that resulting array is again a heap
- The algorithm `pop_heap` from Section 10.7, on the other hand, *removes* an element from a given heap in such a way that the resulting array is again a heap
- The algorithm `make_heap` from Section 10.8 rearranges a given array into a heap.
- Finally, the algorithm `sort_heap` from Section 10.9 sorts a heap into an increasing range.

In Section 10.1 we present in more detail how heaps are defined. The ACSL logic functions and predicate that formalize the basic heap properties of heaps are introduced in Section 10.2.

```
#define SIZE_TYPE_MAX  UINT_MAX
```

Listing 10.2: Upper limits of `size_type`

In order to admit maximally large heaps, we had to catch border cases in ACSL as well as in C, cf. e.g. Listing 10.7 and 10.11. To this end, we introduced the constant from Listing 10.2. to refer to the upper bound of `size_type`. We don't need a corresponding constant `SIZE_TYPE_MIN` for the lower bound, since it is trivial.

10.1. Basic heap concepts

The 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:[29]

> 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:

$$\text{left child for index } i \qquad \text{child}_\text{l} : i \mapsto 2i + 1 \tag{10.1}$$
$$\text{right child for index } i \qquad \text{child}_\text{r} : i \mapsto 2i + 2 \tag{10.2}$$

and

$$\text{parent index for index } i \qquad \text{parent} : i \mapsto \frac{i-1}{2} \tag{10.3}$$

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. [15, §2.2.4]).

$$\text{parent}(\text{child}_\text{l}(i)) = i \tag{10.4}$$
$$\text{parent}(\text{child}_\text{r}(i)) = i \tag{10.5}$$

In order to given an example for the usefulness of heaps we consider the following multiset of integers X.

$$X = \{2, 3, 3, 3, 6, 7, 8, 8, 9, 11, 13, 14\} \tag{10.6}$$

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

Figure 10.3 shows how the multiset from Equation (10.6) can, according to the parent-child relations of a heap, be represented as a tree.

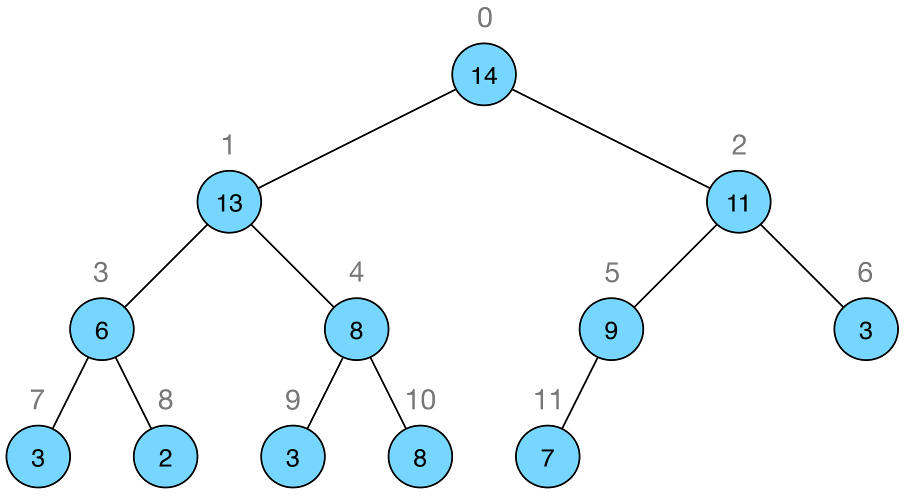


Figure 10.3.: Tree representation of the multiset X

The numbers outside the nodes in Figure 10.3 are the indices at which the respective node value is stored in the underlying array of a heap (cf. Figure 10.4).

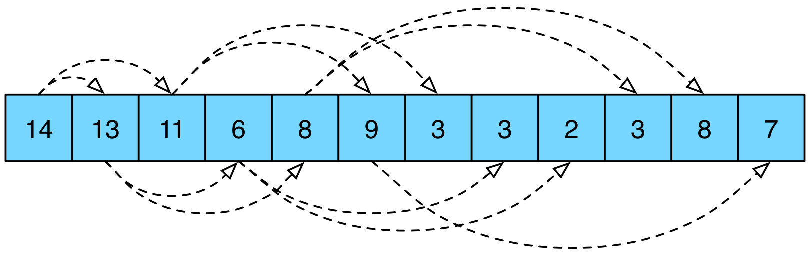


Figure 10.4.: Underlying array of a heap

It is important to understand that there can be various representations of a multiset as a heap. Figure 10.5, for example, arranges the elements of the multiset X as a heap in a different tree.

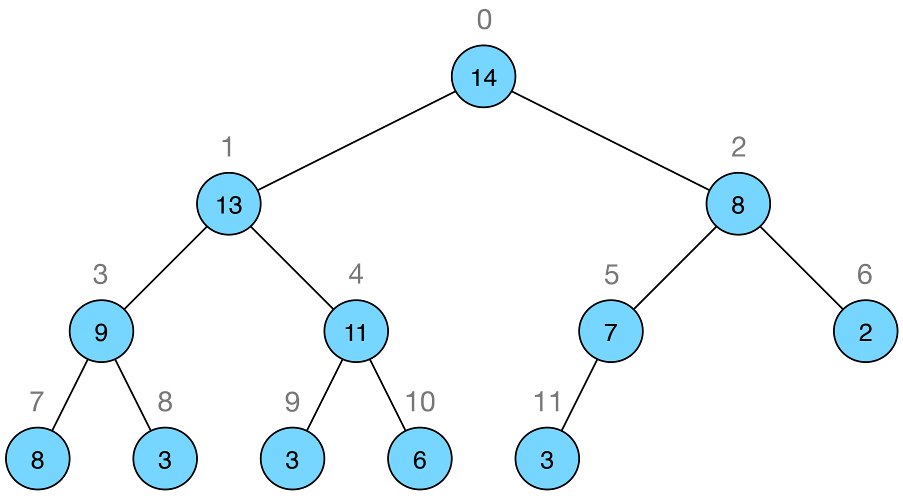


Figure 10.5.: An alternative representation of the multiset X

Figure 10.6 then shows the underlying array that corresponds to the tree in Figure 10.5.

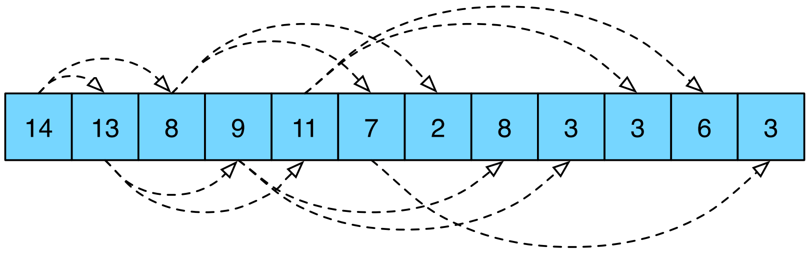


Figure 10.6.: Underlying array of the alternative representation

10.2. Presentation of heap concepts in ACSL

Listing 10.7 shows three logic functions `HeapLeft`, `HeapRight` and `HeapParent` that correspond to the definitions (10.1), (10.2) and (10.3), respectively. The predicate `HeapChildMax` describes the child of a heap node with the the largest index.

```
/*@
  axiomatic HeapBasics
  {
    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 p; 0 <= p  ==>  HeapParent(HeapLeft(p)) == p;

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

    lemma HeapParentChild:
      \forall integer c, p;
        0 < c              ==>
        HeapParent(c) == p  ==>
        (c == HeapLeft(p) || c == HeapRight(p));

    lemma HeapChilds:
      \forall integer a, b;
        0 < a  ==>  0 < b              ==>
        HeapParent(a) == HeapParent(b)  ==>
        (a == b || a+1 == b || a == b+1);

    lemma HeapParentBounds:
      \forall integer c; 0 < c  ==>  0 <= HeapParent(c) < c;

    lemma HeapChildBounds:
      \forall integer p;
        0 <= p  ==>  p < HeapLeft(p) < HeapRight(p);

    predicate
    HeapChildMax{L}(value_type* a, integer n, integer p, integer c) =
      0 <= p < n-1                                          &&
      (p < (SIZE_TYPE_MAX-1)/2  ==>  p == HeapParent(c))      &&
      (HeapLeft(p)  < n-1       ==>  a[HeapLeft(p)]  <= a[c]) &&
      (HeapRight(p) < n-1       ==>  a[HeapRight(p)] <= a[c]);
  }
*/
```

Listing 10.7: Basic ACSL description of heaps

Listing 10.7 also contains a number of ACSL lemma that state among other things that

- the HeapParent function satisfies the equations (10.4) and (10.5) and
- the function HeapParent is the *left inverse* to the HeapLeft and HeapRight functions.[30]

On top of these basic definitions we introduce in Listing 10.8 the predicate Heap.

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

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

Listing 10.8: The predicate Heap

The root of a heap, that is the element at index 0, is always the largest element of the heap. Lemma HeapMaximum from Listing 10.8 expresses this property using the predicate MaxElement from Listing 5.2. We use the following fact about division in C in the proof of lemma HeapMaximum.

```
/*@
  axiomatic C_Division
  {
    lemma C_Division_Two:
      \forall integer a; 0 <= a  ==>  0 <= a/2 <= a;
  }
*/
```

Listing 10.9: The lemma C_Division_Two

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

10.3. The auxiliary functions heap_parent and heap_child_max

This section features the two auxiliary heap functions We start with the function heap_parent from Listing 10.10 which is the C counterpart of the ACSL function HeapParent function in Listing 10.7.

```
/*@
   requires bound:  0 < child;
   assigns          \nothing;
   ensures  parent: \result == HeapParent(child);
 */
static inline size_type
heap_parent(size_type child)
{
  return (child - 1u) / 2u;
}
```

Listing 10.10: The auxiliary heap function heap_parent

The second auxiliary function is heap_child_max from Listing 10.11. It computes the child of an heap node with the the largest index. This function will be used in the implementation of pop_heap 10.7. On the logic level this notion is expressed in ACSL by the predicate HeapChildMax from Listing 10.7. Note that it explicitly handles the case that the child index computation would overflow; in which case it returns n.

```
/*@
   requires bound: 2 <= n;
   requires bound: 0 <= parent < n - 1;
   requires valid: \valid(a + (0..n-1));
   requires heap:  Heap(a, n);
   assigns         \nothing;
   ensures heap:   Heap(a, n);
   ensures max:    HeapChildMax(a, n, parent, \result);
   ensures less:   parent < \result;
   ensures less:   \result < n - 1  ==>  parent == HeapParent(\result);
*/
static inline size_type
heap_child_max(const value_type* a, size_type n, size_type parent)
{
  if (parent < (SIZE_TYPE_MAX - 1u) / 2u) {
    const size_type right = 2u * parent + 2u;
    const size_type left  = right - 1u;

    if (right < n - 1u) {
      // case of two children: select child with maximum value
      return a[left] >= a[right] ? left : right;
    }
    else {
      // at most one child that comes before n-1 can exist
      return left;
    }
  }
  else {
    return n;
  }
}
```

Listing 10.11: The auxiliary function heap_child_max

10.4. The is_heap_until algorithm

The is_heap_until algorithm of the C++ Standard Library [20, §28.7.7.5] works on generic sequences. For our purposes we have modified the generic implementation to that of an array of type value_type. The signature now reads:

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

The algorithm is_heap_until returns the largest range of an array, beginning at the first position, where it still satisfies the heap properties we have semi-formally described in the beginning of this chapter. In particular, is_heap_until will return the size of the array, called with the array argument from Figure 10.4.

10.4.1. Formal specification of is_heap_until

The ACSL specification of is_heap_until is shown in Listing 10.12. The function returns an index i if and only if its array a[0..i-1] satisfies the predicate Heap introduced in Section 10.2. In addition the postcondition last states, that for all indices greater than or equal to i the predicate Heap is not satisfied.

```
/*@
   requires valid: \valid_read(a + (0..n-1));
   assigns         \nothing;
   ensures bound:  0 <= \result <= n;
   ensures heap:   Heap(a, \result);
   ensures last:   \forall integer i; \result < i <= n ==> !Heap(a, i);
*/
size_type
is_heap_until(const value_type* a, size_type n);
```

Listing 10.12: Function Contract of is_heap_until

10.4.2. Implementation of is_heap_until

Listing 10.13 shows one way to implement the function is_heap_until. 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. If the value at a parent index is smaller than the value at a (child) index, is_heap_until returns the (child) index. Otherwise, if the algorithm iterates through the whole array, the size of the array is returned.

```
size_type
is_heap_until(const value_type* a, size_type n)
{
  size_type parent = 0u;

  /*@
    loop invariant bound:    0 <= parent < child <= n+1;
    loop invariant parent:   parent == HeapParent(child);
    loop invariant heap:     Heap(a, child);
    loop invariant not_heap: a[parent] < a[child] ==> \forall integer i; child < i <=
         n ==> !Heap(a, i);

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

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

  return n;
}
```

Listing 10.13: Implementation of is_heap_until

10.5. The is_heap algorithm

The is_heap algorithm of the C++ Standard Library [20, §28.7.7.5] works on generic sequences. For our purposes we have modified the generic implementation 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 10.4.

10.5.1. Formal specification of is_heap

The ACSL specification of is_heap is shown in Listing 10.14. The function returns **true** if and only if its arguments satisfy the predicate Heap introduced in Section 10.2.

```
/*@
   requires valid: \valid_read(a +(0..n-1));
   assigns         \nothing;
   ensures heap:   \result <==> Heap(a, n);
*/
bool
is_heap(const value_type* a, size_type n);
```

Listing 10.14: Function Contract of is_heap

10.5.2. Implementation of is_heap

Listing 10.15 shows one way to implement the function is_heap by utilizing the function is_heap_until from Section 10.4.

The function is_heap returns **true** if its whole given range satisfies the heap properties. So basically, is_heap returns **true** if the function is_heap_until returns the size of the input array.

```
bool
is_heap(const value_type* a, size_type n)
{
  return is_heap_until(a, n) == n;
}
```

Listing 10.15: Implementation of is_heap

10.6. The push_heap algorithm

Whereas in the C++ Standard Library [20, §28.7.7.1] push_heap works on a range of random access iterators, 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 n is greater or equal than 1. It also assumes that the array a[0..n-2] forms a heap. The algorithms then *rearranges* the array a[0..n-1] such that the resulting array is a heap. In this sense the algorithm *pushes* an element on a heap.

10.6.1. Formal specification of push_heap

Listing 10.16 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.

```
/*@
   requires nonempty:   0 < n;
   requires valid:      \valid(a + (0..n-1));
   requires heap:       Heap(a, n-1);
   assigns              a[0..n-1];
   ensures heap:        Heap(a, n);
   ensures reorder:     MultisetUnchanged{Old,Here}(a, n);
*/
void
push_heap(value_type* a, size_type n);
```

Listing 10.16: Formal specification of push_heap

Pushing an element on a heap usually *rearranges* several elements of the array (cf. Figures 10.17 and 10.18). 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 7.56, to formally describe this property.

10.6.2. 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 10.17 shows the heap from Figure 10.3 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.

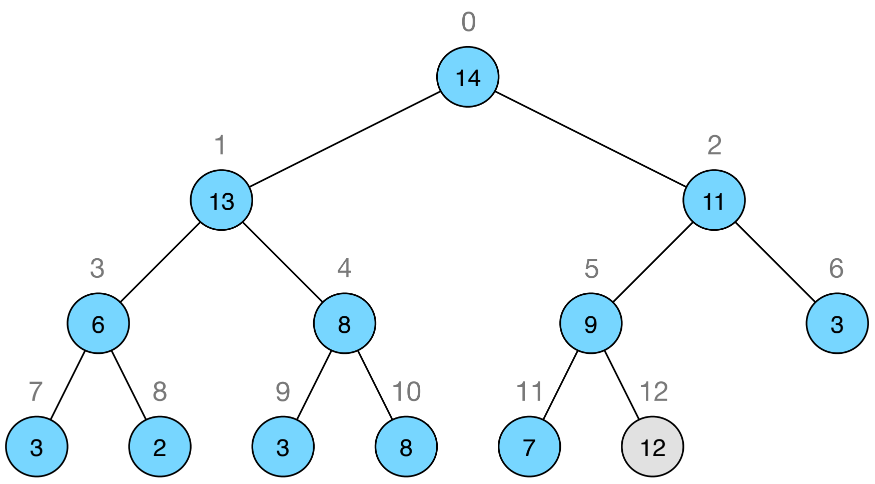


Figure 10.17.: Heap before the call of `push_heap`

Figure 10.18 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.

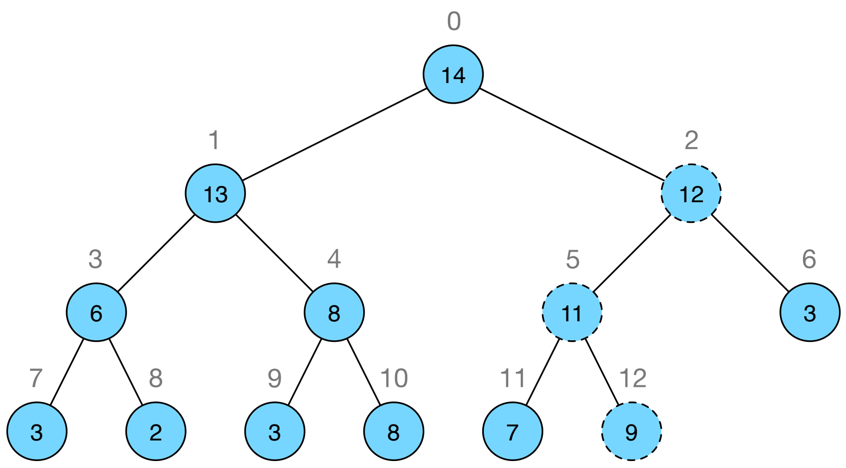


Figure 10.18.: Heap after the call of `push_heap`

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 as

- *prologue* (see Section 10.6.2)
- *main act* (see Section 10.6.2)
- *epilogue* (see Section 10.6.2)

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 10.17.

Verifying `push_heap` is a non-trivial undertaking, and we will proceed, roughly speaking, as follows:

> We can establish the `heap` property of Listing 10.16 already in the prologue. However, 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.

Prologue

Our prologue initializes some important variables, checks whether the initial heap is nonempty, *and* also tries to move the new element upwards within the heap. In other implementations, the latter step is usually performed as part of `push_heap`'s main loop. In order to better understand our implementation decision we can look at Figure 10.19 which shows exemplarily its effects.

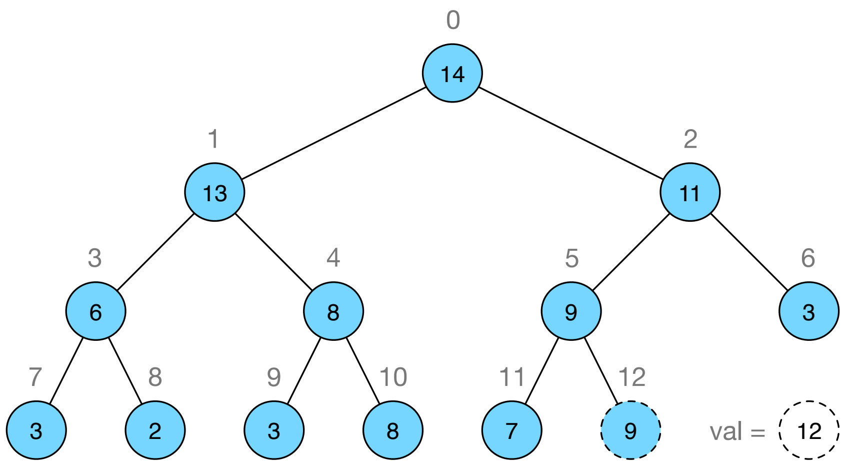


Figure 10.19.: Heap after the prologue of `push_heap`

If we compare this tree to the tree in Figure 10.17 we notice that the element in the node with the dashed outlining changed its 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. The increased number of occurrences of `9` and the decreased number of elements `12` means that at this stage the postcondition `reorder` is violated. On the other hand, the modified array is now a heap. We express this by coloring all elements blue (cf. Figure 10.17).

More generally speaking, the following properties hold after the prologue:

1. The modified array is now a heap.
2. The "parent value" `a[parent]` now occurs one time more often.
3. The value `a[n-1]`, on the other hand, now occurs one time fewer.
4. No other value changed its number of occurrences.

Listing 10.20 shows the implementation of the prologue. It starts with the listing of an auxiliary function `heap_parent` that computes the parent index of its argument. The prologue also deals with the trivial cases that

- the array contains only one element or
- if `a[n-1]` is less or equal than its parent element

At the end of the prologue we have added four assertions that formally express the properties we have just enumerated above. As we will see (in Listing 10.23), these properties will occur as loop invariants in the main act. Adding these assertions makes the purpose of the prologue more explicit and thus supports the long-term maintenance of the annotated code. The auxiliary predicates that are used in the assertions are:

- predicate `MultisetAdd` (Listing 10.21)
- predicate `MultisetMinus` (Listing 10.21)
- various overloaded versions of predicate `MultisetRetain` and `MultisetRetainRest` (Listing 10.22)

```
void
push_heap(value_type* a, size_type n)
{
  // start of prologue
  if (1u < n) { // otherwise nothings needs to be done
    const value_type v   = a[n - 1u];
    size_type hole       = heap_parent(n - 1u);

    if (a[hole] < v) {
      a[n - 1u] = a[hole];

      //@ assert heap:   Heap(a, n);
      //@ assert add:    MultisetAdd{Pre,Here}(a, n, a[hole]);
      //@ assert minus:  MultisetMinus{Pre,Here}(a, n, v);
      //@ assert retain: MultisetRetainRest{Pre,Here}(a, n, v, a[hole]);
      // end of prologue
```

Listing 10.20: Prologue of `push_heap` implementation

These auxiliary predicates are discussed in the following subsections.

The predicates `MultisetAdd` and `MultisetMinus`

The predicate `MultisetAdd` in Listing 10.21 expresses that the number of occurrences of a specific element in an array has increased by one between between two program points `K` and `L`. The predicate `MultisetMinus` in Listing 10.21, on the other hand, expresses that the number of occurrences of a specific element in an array has decreased by one between two program points `K` and `L`.

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

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

    lemma MultisetAddDistinct{K,L}:
      \forall value_type *a, v, integer i, n;
        0 <= i < n                         ==>
        \at(a[i],K) != v                   ==>
        \at(a[i],L) == v                   ==>
        MultisetUnchanged{K,L}(a, 0, i)    ==>
        MultisetUnchanged{K,L}(a, i+1, n)  ==>
        MultisetAdd{K,L}(a, n, v);

    lemma MultisetMinusDistinct{K,L}:
      \forall value_type *a, v, integer i, n;
        0 <= i < n                         ==>
        \at(a[i],K) == v                   ==>
        \at(a[i],L) != v                   ==>
        MultisetUnchanged{K,L}(a, 0, i)    ==>
        MultisetUnchanged{K,L}(a, i+1, n)  ==>
        MultisetMinus{K,L}(a, n, v);
  }
*/
```

Listing 10.21: The predicates `MultisetAdd` and `MultisetMinus`

Note that we could have defined `MultisetMinus` also by calling `MultisetAdd` with the labels reversed.

```
  predicate
  MultisetMinus{K,L}(value_type* a, integer n, value_type val) =
    MultisetAdd{L,K}(a, n, val);
```

It is a often only a matter of taste how to decide which of several ways to define a predicate is more appropriate. However, one also has to take into account which definition can be handled more easily by Frama-C/WP and its associated theorem provers.

In order to guide the automatic provers, we also provide in Listing 10.21 the lemmas `MultisetAddDistinct` and `MultisetMinusDistinct`. These lemmas formalize conditions under which the respective predicates `MultisetAdd` and `MultisetMinus`.

The predicates `MultisetRetain` and `MultisetRetainRest`

In order to achieve a concise specification we introduce the overloaded predicate `MultisetRetainRest` (see Listing 10.22). The expression `MultisetRetainRest{K,L}(a, m, b, n, v)` is true if the range `a[0..n-1]` at time `K` contains the same elements as `b[0..m-1]` at time `L`, except possibly for occurrences of `v`; the elements' order may differ in `a` and `b`. There is also a more general version of `MultisetRetainRest` that is defined over array segments.

```
/*@
  axiomatic MultisetRetain
  {
    predicate
    MultisetRetain{K,L}(value_type* a, integer n, value_type v) =
      Count{K}(a, n, v) == Count{L}(a, n, v);

    predicate
    MultisetRetainRest{K,L}(value_type* a, integer m1, integer m2,
                            value_type* b, integer n1, integer n2, value_type  v) =
      \forall value_type x;
        x != v  ==>  Count{K}(a, m1, m2, x) == Count{L}(b, n1, n2, x);

    predicate
    MultisetRetainRest{K,L}(value_type* a, integer m,
                            value_type* b, integer n, value_type  v) =
      MultisetRetainRest{K,L}(a, 0, m, b, 0, n, v);

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

Listing 10.22: The predicate `MultisetRetain`

For `push_heap` another overloaded version of `MultisetRetainRest` is particularly useful. The new version holds if the number of occurrences for all elements, except the two given ones, remains unchanged between two program points.

Main act

The goal of the main act is to locate the array index to which the new element can be assigned. Listing 10.23 shows its implementation.

is true for the array throughout the main act. Instead of an invariant `reorder` that reflects the postcondition with the same name, we now consider the invariants `add`, `minus`, and `retain`.

It is important to understand the use of the variable `hole` in these loop invariants. Before each loop iteration, `hole` stores the index of the node whose value was assigned to one of its children in the previous iteration or in the prologue (for the first loop run). Therefore, the value `a[hole]` appears in the loop invariants `add` and `retain`.

Verifying the various loop invariants and assertions has been far from being straightforward, and required additional assertions and the predefined label `LoopCurrent`. The following remarks highlight some of the issues.

```
    // start of main act
    if (0u < hole) {
      size_type parent = heap_parent(hole);

      /*@
        loop invariant bound:  0 <= hole < n-1;
        loop invariant heap:   Heap(a, n);
        loop invariant heap:   parent == HeapParent(hole);
        loop invariant less:   a[hole] < v;
        loop invariant add:    MultisetAdd{Pre,Here}(a, n, a[hole]);
        loop invariant minus:  MultisetMinus{Pre,Here}(a, n, v);
        loop invariant retain: MultisetRetainRest{Pre,Here}(a, n, v, a[hole]);
        loop assigns           hole, parent, a[0..n-1];
        loop variant           hole;
      */
      while ((0u < hole) && (a[parent] < v)) {
        if (a[hole] < a[parent]) {
          a[hole] = a[parent];
          //@ assert less:    \at(a[hole],LoopCurrent) < v;
          //@ assert less:    a[hole] < v;
          //@ assert retain:  MultisetUnchanged{LoopCurrent,Here}(a, 0, hole);
          //@ assert retain:  MultisetUnchanged{LoopCurrent,Here}(a, hole + 1, n);
          //@ assert minus:   MultisetMinus{LoopCurrent,Here}(a, n, \at(a[hole],
              LoopCurrent));
          //@ assert add:     MultisetAdd{LoopCurrent,Here}(a, n, a[hole]);
          //@ assert retain:  MultisetRetain{LoopCurrent,Here}(a, n, v);
          //@ assert retain:  MultisetRetain{Pre,Here}(a, n, \at(a[hole],
              LoopCurrent));
          //@ assert retain:  MultisetRetainRest{Pre,Here}(a, n, v, a[hole]);
        }

        hole = parent;

        if (0u < hole) {
          parent = heap_parent(hole);
        }
      }
    }

    // end of main act
```

Listing 10.23: Main act of `push_heap` implementation

The `heap` property implies that `a[hole] <= a[parent]` always holds. Thus, the assignment `a[ hole] = a[parent]` might be redundant. We have not check whether guarding this assignment with the condition `a[hole] < a[parent]` is more efficient. Important for us is the following: the guard allows us to put additional assertions where they really matter and where they can more easily be verified.

In order to guide the automatic provers, we have also provided the lemmas `MultisetPushHeapRetain` and `MultisetPushHeapClosure` (Listing 10.24). Note that `MultisetPushHeapClosure` is needed in the in the verification of `push_heap`'s epilogue in Listing 10.26.

```
/*@
  axiomatic MultisetPushHeap
  {
    lemma MultisetPushHeapRetain{K,L,M}:
     \forall value_type *a, ap, ah, v, integer h, p, n;
       0 <= p < h < n-1                       ==>
       ah < ap < v                            ==>
       \at(a[h],L)   ==  ah                   ==>
       \at(a[p],L)   ==  ap                   ==>
       \at(a[h],M)   ==  ap                   ==>
       MultisetMinus{K,L}(a, n, v)            ==>
       MultisetAdd{K,L}(a, n, ah)             ==>
       MultisetRetainRest{K,L}(a, n, v, ah)   ==>
       MultisetUnchanged{L,M}(a, 0, h)        ==>
       MultisetUnchanged{L,M}(a, h+1, n)      ==>
       MultisetRetainRest{K,M}(a, n, v, ap);

   lemma MultisetPushHeapClosure{K,L,M}:
     \forall value_type *a, u, v, integer i, n;
       0 <= i < n-1                          ==>
       u != v                                ==>
       \at(a[i],M)   == v                    ==>
       MultisetAdd{K,L}(a, n, u)             ==>
       MultisetMinus{K,L}(a, n, v)           ==>
       MultisetRetainRest{K,L}(a, n, v, u)   ==>
       MultisetUnchanged{L,M}(a, 0, i)       ==>
       MultisetUnchanged{L,M}(a, i+1, n)     ==>
       MultisetAdd{L,M}(a, n, v)             ==>
       MultisetMinus{L,M}(a, n, u)           ==>
       MultisetUnchanged{K,M}(a, n);
  }
*/
```

Listing 10.24: The lemmas `MultisetPushHeapRetain` and `MultisetPushHeapClosure`

Figure 10.25 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.

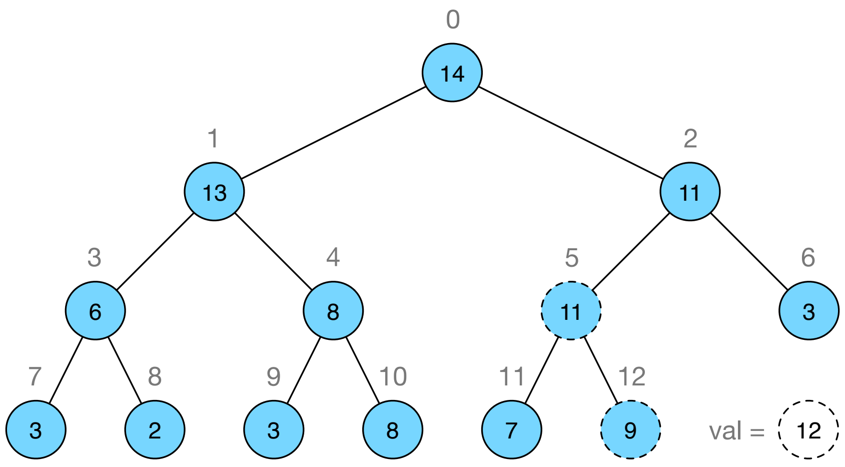


Figure 10.25.: Heap after the main act of `push_heap`

In our example, the loop performs just one assignment, viz. a[5] = a[2], and then stops with hole being 2. 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.

Epilogue

The last part of the implementation is the epilogue, shown in Listing 10.26. It consists of exactly one assignment which re-establishes the reorder property while maintaining the heap property already established.

```
      // start of epilogue
      /*@
        requires value:      a[hole] != v;
        assigns              a[hole];
        ensures  value:      a[hole] == v;
        ensures  unchanged:  MultisetUnchanged{Old,Here}(a, 0, hole);
        ensures  unchanged:  MultisetUnchanged{Old,Here}(a, hole+1, n);
      */
      a[hole] = v;
      //@ assert reorder:    MultisetUnchanged{Pre,Here}(a, n);
    }
  }

  // end of epilogue
}
```

Listing 10.26: Epilogue of push_heap implementation

We use here a simple statement contract together with the lemma MultisetPushHeapClosure from Listing 10.24 in order to guide the automatic theorem provers to verify the final assertion reorder.

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 (cf. Figure 10.25). The heap in Figure 10.27, on the other hand, shows the tree after the epilogue has assigned the value 12 to the node with the index 2, which contained the value 11. Hence the reorder property is re-established and the function can return. Moreover, since the heap property has been inferred, all nodes are now colored blue.

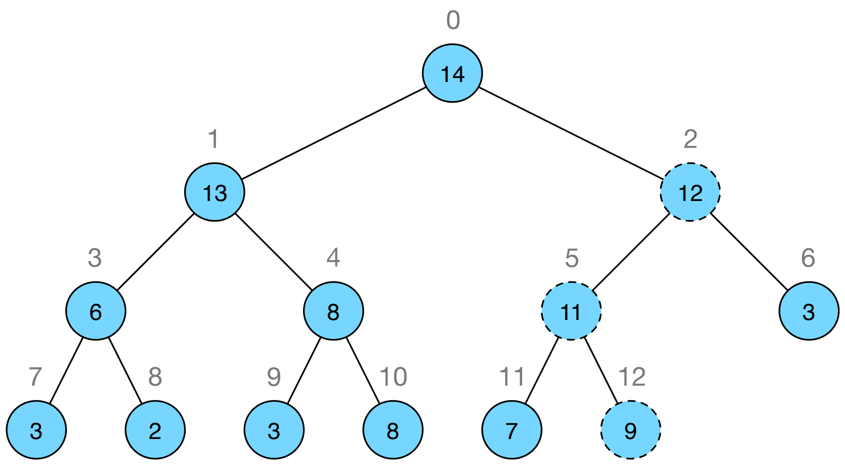


Figure 10.27.: Heap after the epilogue of push_heap

10.7. The pop_heap algorithm

Whereas in the C++ Standard Library [20, §28.7.7.2] pop_heap works on a range of random access iterators, our version operates on an array of value_type. We therefore use the following signature for pop_heap

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

The pop_heap algorithm expects that n is greater or equal than 1 and that the array a[0..n-1] forms a heap. The algorithms then *rearranges* the array a[0..n-1] such that the resulting array satisfies the following properties.

- a[n-1] = \old(a[0]), that is, the largest element[31] of the original heap is transferred to the end of the array
- the subarray a[0..n-2] is a heap

In this sense the algorithm *pops* the largest element from a heap.

10.7.1. Formal specification of pop_heap

The ACSL function contract of pop_heap is given in Listing 10.28.

```
/*@
   requires bounds:     0 < n;
   requires valid:      \valid(a + (0..n-1));
   requires heap:       Heap(a, n);
   assigns              a[0..n-1];
   ensures heap:        Heap(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 10.28: Formal specification of pop_heap

10.7.2. Implementation of pop_heap

Listing 10.29 shows our implementation of pop_heap together with ACSL annotations. Note that in this version the postcondition reorder of pop_heap, which states that the algorithm only *rearranges* the elements of the array, is not verified by Frama-C/WP. Verifying this postcondition would require more elaborate loop invariants which we will supply in a later version of this document.

[31]See Lemma HeapMaximum in Listing 10.8.

```
void
pop_heap(value_type* a, size_type n)
{
  if (1u < n) { // otherwise nothings needs to be done
    const value_type v = a[0u];

    //@ assert max:  MaxElement(a, n, 0);
    if (a[n - 1u] < v) { // otherwise nothings needs to be done
      //@ assert bounds: 2 <= n;
      size_type hole = 0u;
      size_type child = heap_child_max(a, n, hole);

      //@ assert heap: child < n - 1  ==>  hole == HeapParent(child);
      /*@
          loop invariant bounds: 0 <= hole < n-1;
          loop invariant bounds: hole < child;
          loop invariant heap:   Heap(a, n);
          loop invariant heap:   a[n-1] < a[HeapParent(hole)];
          loop invariant heap:   child < n - 1  ==>  hole == HeapParent(child);
          loop invariant child:  HeapChildMax(a, n, hole, child);
          loop invariant max:    UpperBound(a, 0, n, v);
          loop assigns           hole, child, a[0..n-2];
          loop variant           n - hole;
       */
      while ((child < n - 1u) && (a[n - 1u] < a[child])) {
        a[hole] = a[child];
        hole    = child;
        //@ assert heap: Heap(a, n);
        child = heap_child_max(a, n, hole);
      }

      //@ assert child: child < n-1 ==> a[n-1] >= a[child];
      //@ assert child: HeapChildMax(a, n, hole, child);
      //@ assert heap:  Heap(a, n);
      //@ assert heap:  a[n-1] < a[HeapParent(hole)];
      a[hole]   = a[n - 1u];
      //@ assert heap:  Heap(a, n-1);
      a[n - 1u] = v;
      //@ assert heap:  Heap(a, n-1);
    }
  }
}
```

Listing 10.29: Implementation of pop_heap

Our implementation relies on the auxiliary function heap_child_max from Listing 10.11 (Section 10.3). As discussed there this helper function computes the child of an heap node with the the largest index. On the logic level this notion is expressed in ACSL by the predicate HeapChildMax from Listing 10.7.

10.8. The `make_heap` algorithm

Whereas in the C++ Standard Library [20, §28.7.7.3] `make_heap` works on a pair of generic random access iterators, 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.

As an examples we look at the array in Figure 10.30. The elements of this array do not form a heap, as indicated by the grey colouring. Executing the `make_heap` algorithm on this array rearranges its elements so that they form a heap as shown in Figure 10.4.

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

Figure 10.30.: Array before the call of `make_heap`

10.8.1. Formal specification of `make_heap`

Listing 10.31 shows the ACSL specification of `make_heap`.

```
/*@
   requires valid:  \valid(a + (0..n-1));
   assigns          a[0..n-1];
   ensures heap:    Heap(a, n);
   ensures reorder: MultisetUnchanged{Old,Here}(a, n);
*/
void
make_heap(value_type* a, size_type n);
```

Listing 10.31: Formal 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 7.56) to ensure `make_heap` only rearranges the array elements.

10.8.2. Implementation of make_heap

The implementation of make_heap, shown in Listing 10.32, is straightforward. From low to high the array's elements are pushed to the growing heap. We used i < n as loop condition, rather than the more tempting i <= n, in order to admit also n == SIZE_TYPE_MAX; as a consequence, we had to call push_heap with i+1. The iteration starts at i+1 == 2, because an array with length one is a heap already.

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

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

Listing 10.32: 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 10.6.1).

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

10.9. The `sort_heap` algorithm

Whereas in the C++ Standard Library [20, §28.7.7.4] `sort_heap` works on a range of random access iterators, 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]` in increasing order. Thus, applying `sort_heap` to the heap in Figure 10.4 produces the increasing array in Figure 10.33.

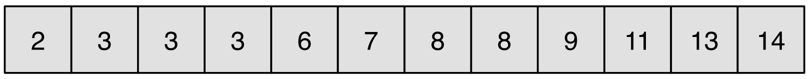


Figure 10.33.: Array after the call of `sort_heap`

10.9.1. Formal specification of `sort_heap`

Listing 10.34 shows our ACSL specification of `sort_heap`. The formal specification of `sort_heap` must ensure that the resulting array is increasing. Furthermore the multiset contained by the array must be the same as in the pre-state of the function.

```
/*@
   requires valid:       \valid(a + (0..n-1));
   requires heap:        Heap(a, n);
   assigns               a[0..n-1];
   ensures reorder:      MultisetUnchanged{Old,Here}(a, n);
   ensures increasing:   Increasing(a, n);
*/
void
sort_heap(value_type* a, size_type n);
```

Listing 10.34: Formal specification of `sort_heap`

The postconditions `increasing` and `reorder` express these properties, respectively. The specification effort is relatively simple because we can reuse the previously defined predicates `MultisetUnchanged` (Listing 7.56) and `Increasing` (Listing 6.1).

10.9.2. Implementation of `sort_heap`

The implementation of `sort_heap` (Listing 10.35) is relatively simple because it relies on the `pop_heap` algorithm performing essential work.

```
void
sort_heap(value_type* a, size_type n)
{
  /*@
     loop invariant bound:        0 <= i <= n;
     loop invariant heap:         Heap(a, i);
     loop invariant lower:        LowerBound(a, i, n, a[0]);
     loop invariant reorder:      MultisetUnchanged{Pre,Here}(a, 0, n);
     loop invariant increasing:   Increasing(a, i, n);
     loop assigns i, a[0..n-1];
     loop variant i;
  */
  for (size_type i = n; i > 1u; --i) {
    /*@
        requires heap:     Heap(a, i);
        assigns  a[0..i-1];
        ensures  heap:     Heap(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 10.35: Implementation of `sort_heap`

Our implementation of `sort_heap` repeatedly calls `pop_heap` to extract the maximum of the shrinking heap and adding it to the part of the array that is already in increasing order. 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 arranged in increasing order.

In order to facilitate the automatic verification of the property `increasing` we rely among others on the properties `lower` and `max` and the lemma `IncreasingUpperBound` from Listing 10.36.

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

Listing 10.36: The lemma `IncreasingUpperBound`

To verify the property `reorder` we rely on the lemmas from Listing 7.56 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)`.

11. Sorting Algorithms

Many issues in computer science can be exemplified in the field of sorting algorithms; see e.g. [29] for a famous textbook. Therefore we arrange some of the most common classic sorting algorithms. In this chapter, we present algorithms of the C++ Standard Library [20, §28.7.1] that are related to the task of sorting a linear array.

Following [30], we have also used (C rephrasings of) functions from the C++ Standard Library as far as possible to implement the different algorithmic approaches.

- Section 11.1 shows an algorithm to check if a given array is already in increasing order.
- The algorithm in Section 11.2 partitions a given array, and sorts only the resulting lower part.
- `bubble_sort` in Section 11.3 describes a simple, well-known and sorting algorithm.[32]
- `selection_sort` in Section 11.4 presents the classic *selection sort* algorithm.[33]
- `insertion_sort` in Section 11.5 the also well-known *insertion sort* algorithm.[34]
- `merge` in Section 11.7 the *merge* algorithm from *merge sort*.[35]
- `heap_sort` in Section 11.6 describes the quite efficient *heap sort*, which relies on the algorithms presented in Chapter 10.[36]

These algorithms essentially share the following contract; it is their implementations that differ fundamentally.

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

   assigns  a[0..n-1];

   ensures increasing:  Increasing(a, n);
   ensures reorder:     MultisetUnchanged{Old, Here}(a, n);
*/
void xxx_sort(value_type* a, size_type n);
```

While `heap_sort` achieves a run-time complexity upper bound of $O(n \cdot \log(n))$ due to the efficiency of the heap data structure, both `selection_sort` and `insertion_sort` need $O(n^2)$ in the average case, and also in the worst case.

Note that the `sort` algorithm from the C++ Standard Library is not handled here because it typically relies on *introspection sort* which is sophisticated mix of various classic algorithms.[37] In future releases we plan to handle the more algorithms related sorting.

[32]See `https://en.wikipedia.org/wiki/Bubble_sort`
[33]See `https://en.wikipedia.org/wiki/Selection_sort`
[34]See `https://en.wikipedia.org/wiki/Insertion_sort`
[35]See `https://en.wikipedia.org/wiki/Merge_sort`
[36]See `https://en.wikipedia.org/wiki/Heapsort`
[37]See `https://en.wikipedia.org/wiki/Introsort`

11.1. The `is_sorted` algorithm

Our version of the `is_sorted` algorithm compared to the C++ Standard Library [20, §28.7.1.5] has the signature

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

It returns `true` if the given array is in increasing order, and `false` otherwise.

11.1.1. Formal specification of `is_sorted`

Listing 11.1 shows the ACSL specification of `is_sorted`. In the contract, we use the predicate `Increasing`, (see Listing 6.1) which states that any array element is always less or equal to any other element right of it. We'll use an easier-to-handle predicate in the implementation, see below.

```
/*@
  requires valid: \valid_read(a + (0..n-1));
  assigns         \nothing;
  ensures result: \result <==> Increasing(a, n);
*/
bool
is_sorted(const value_type* a, size_type n);
```

Listing 11.1: Formal specification of `is_sorted`

11.1.2. Implementation of `is_sorted`

The implementation of `is_sorted` is shown in Listing 11.2. As usual, it doesn't compare every array element to all that are right to it, but only to the immediately adjacent one, which is of course more efficient. For this reason, we use the predicate `WeaklyIncreasing` from Listing 6.1 in the loop invariant of the implementation.

```
bool
is_sorted(const value_type* a, size_type n)
{
  if (0u < n) {
    /*@
        loop invariant increasing: WeaklyIncreasing(a, i+1);
        loop assigns i;
        loop variant n - i;
    */
    for (size_type i = 0u; i < n - 1u; ++i) {
      if (a[i] > a[i + 1u]) {
        return false;
      }
    }
  }

  return true;
}
```

Listing 11.2: The implementation of `is_sorted`

Users inexperienced in formal verification often have a blind spot at the difference between Increasing and WeaklyIncreasing. Both versions are logically equivalent, and proving that Increasing implies WeaklyIncreasing is even trivial. However, proving the converse direction is not, and requires an induction on the array size n, employing the transitivity of <= in the induction step. Humans are trained to perform such inductions unnoticed, but none of the automated provers supported by Frama-C is able to perform induction at all.

Since our implementation uses WeaklyIncreasing in its loop invariant, and follows the same principle in its code, its verification is straight-forward — except for the final reasoning that WeaklyIncreasing (a,n) implies Increasing(a,n). We have an own lemma for that step, shown in Listing 11.3, which needs to be proven manually with Coq. The converse lemma (Listing 11.3) is proven automatically, but isn't actually needed to verify our is_sorted implementation.

Alternatively, we could have dragged the predicate Increasing along the loop, which happens to cause no particular problems in this case.

```
/*@
  axiomatic IncreasingLemmas
  {
    lemma WeaklyIncreasingAddElement{L}:
      \forall value_type *a, integer m;
        1 < m && WeaklyIncreasing(a, m-1) && a[m-2] <= a[m-1]  ==>  WeaklyIncreasing(
            a, m);

    lemma WeaklyIncreasingShift{L}:
      \forall value_type *a, integer n, m;
        WeaklyIncreasing(a + n, 0, m)  <==>  WeaklyIncreasing(a, n, m + n);

    lemma EqualRangesWeaklyIncreasing{L}:
      \forall value_type *a, *b, integer n, m;
        WeaklyIncreasing(a, n, m) && EqualRanges{L,L}(a, n, m, b)  ==>
        WeaklyIncreasing(b, n, m);

    lemma WeaklyIncreasingJoin{L}:
      \forall value_type *a, integer n, m;
        0 < n < m                 &&
        WeaklyIncreasing(a, n)    &&
        WeaklyIncreasing(a, n, m) &&
        a[n-1] <= a[n]            ==>
        WeaklyIncreasing(a, m);

    lemma IncreasingImpliesWeaklyIncreasing{L}:
      \forall value_type* a, integer m, n;
        0 <= m <= n                 ==>
        Increasing(a, m, n)         ==>
        WeaklyIncreasing(a, m, n);

    lemma WeaklyIncreasingImpliesIncreasing{L}:
      \forall value_type* a, integer m, n;
        0 <= m <= n                ==>
        WeaklyIncreasing(a, m, n)  ==>
        Increasing(a, m, n);
  }
*/
```

Listing 11.3: Some lemmas about Increasing and WeaklyIncreasing

11.2. The `partial_sort` algorithm

Our version of the `partial_sort` algorithm compared to the C++ Standard Library [20, §28.7.1.3] has the signature

```
void partial_sort(value_type* a, size_type m, size_type n);
```

The algorithm *reorders* the given array `a` in such a way that it represents a *partition*: each member of the left part `a[0..m-1]` is less or equal to each member of the right part `a[m..n-1]`. Moreover, the algorithm *sorts* the left part in increasing order. The order of elements in the right part, however, is *unspecified*. Figure 11.4 uses a bar chart to depict a typical result of a call `partial_sort(a, m, n)`. In the post-state, the left and the right part is colored in green and orange, respectively.

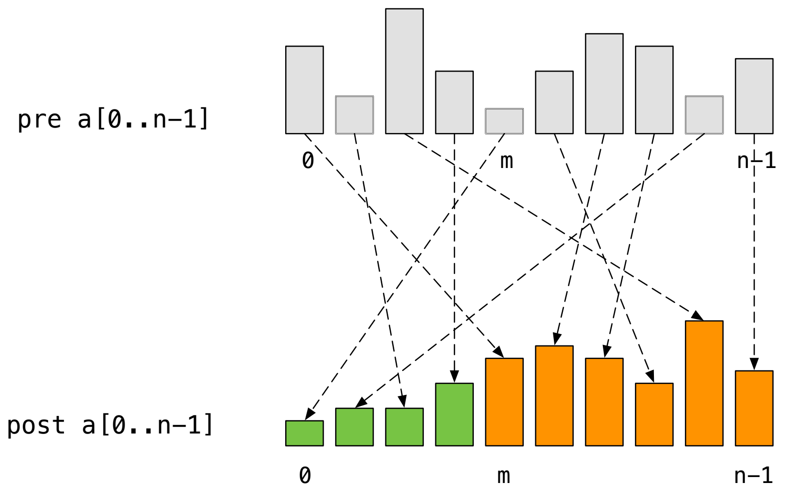


Figure 11.4.: Effects of `partial_sort`

11.2.1. Formal specification of `partial_sort`

We start this section by introducing in Listing 11.5 the new predicate `Partition` which formalizes the partitioning property.

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

Listing 11.5: The predicate `Partition`

The formal specification of the `partial_sort` function is shown in Listing 11.6. It uses the just introduced predicate `Partition` and reuses the previously defined predicates `Increasing` (shown in Listing 6.1) and `MultisetUnchanged` (Listing 7.56).

```
/*@
  requires valid:        \valid(a + (0..n-1));
  requires split:        0 <= m <= n;
  assigns                a[0..n-1];
  ensures reorder:       MultisetUnchanged{Old,Here}(a, n);
  ensures partition:     Partition(a, m, n);
  ensures increasing:    Increasing(a, m);
*/
void
partial_sort(value_type* a, size_type m, size_type n);
```

Listing 11.6: Formal specification of `partial_sort`

11.2.2. Implementation of `partial_sort`

Our implementation is shown in Listing 11.8 and 11.9. It initially calls `make_heap` (Section 10.8) to rearrange the left part `a[0..m-1]` into a heap. After that, it scans the right part, from left to right, for elements that are too small; each such element is exchanged for the left part's maximum, by applying `pop_heap` (10.7) and `push_heap` (10.6) appropriately. When the scan is done, the smallest elements are collected in the left part. We finally convert it from a heap into an increasingly ordered range, by `sort_heap` (10.9).

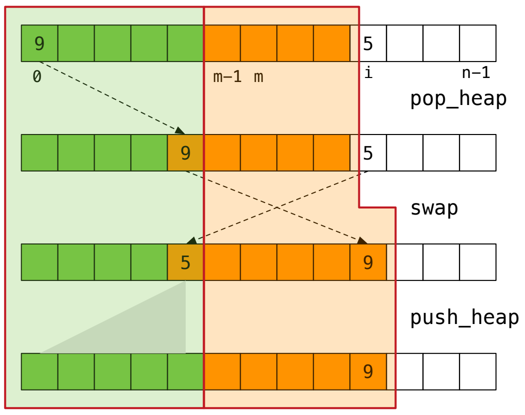


Figure 11.7.: An iteration of `partial_sort`

In the scan loop, we maintain as invariants

- that the left part is a heap (invariant `heap`);
- that its maximal element, `a[0]`, is a "separating element" between the left part `a[0..m-1]` and the right part `a[m..i-1]`, i.e., an upper bound of the left (invariant `upper`) and a lower bound of the right part (invariant `lower`), respectively;
- that `a[i..m-1]` is yet unchanged (invariant `unchanged`); and
- that only permutation operations have been applied to `a[0..i-1]` (invariant `reorder`).

In order to preserve the loop invariants after `i` is incremented, nothing has to be done if `a[0]` happens to be also a lower bound for `a[i]`. Otherwise, let us follow the algorithm through the `then` part code, depicting the intermediate states in Figure 11.7. The elements considered so far are shown colored similar to Figure 11.4; in particular the heap part is shown in green.

```
void
partial_sort(value_type* a, size_type m, size_type n)
{
  if (m > 0u) {
    make_heap(a, m);

    //@ assert reorder: Unchanged{Pre,Here}(a, m, n);
    /*@
      loop invariant bound:     m <= i <= n;
      loop invariant heap:      Heap(a, m);
      loop invariant upper:     UpperBound(a, 0, m, a[0]);
      loop invariant lower:     LowerBound(a, m, i, a[0]);
      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 = m; i < n; ++i)
      if (a[i] < a[0u]) {
        /*@
          assigns               a[0..m-1];
          ensures heap:         Heap(a, m-1);
          ensures max:          a[m-1] == \old(a[0]);
          ensures max:          MaxElement(a, m, m-1);
          ensures reorder:      MultisetUnchanged{Old,Here}(a, m);
          ensures unchanged:    Unchanged{Old,Here}(a, m, i);
          ensures unchanged:    Unchanged{Old,Here}(a, m, n);
        */
        pop_heap(a, m);
        //@ assert lower:       a[0] <= a[m-1];
        //@ assert lower:       a[i] <  a[m-1];
        //@ assert lower:       LowerBound(a, m, i, a[m-1]);
        //@ assert upper:       UpperBound(a, 0, m-1, a[0]);
        //@ assert upper:       UpperBound(a, 0, m,   a[m-1]);
        //@ assert partition:   Partition(a, m, i);
        //@ assert reorder:     MultisetUnchanged{Pre,Here}(a, i);
```

Listing 11.8: Implementation of `partial_sort` (1)

The overlaid transparent red shape indicates the ranges to which `Partition` applies, in each state. The figure assumes the initial contents of `a[0]` and `a[i]` to be 9 and 5, for sake of generality, let us call them p and q, respectively.

After `pop_heap` and `swap`, we have p at `a[i]`, and q at `a[m-1]`. At that point we know

1. $q < p \leq$ `a[k]` for each $m \leq k < i$, since p was a lower bound for `a[m..i-1]`;
2. $q < p =$ `a[i]`;
3. `a[j]` $\leq p \leq$ `a[k]` for each $0 \leq j < m-1$ and each $m \leq k < i$, since this held on loop entry, and we didn't more than reordering inside the parts; and
4. `a[j]` $\leq p =$ `a[i]` since p was the heap maximum on loop entry.

Altogether, we have `a[j]` $\leq p \leq$ `a[k]` for each $0 \leq j < m$ and each $m \leq k < i+1$. That is, `Partition (a,m,i+1)` holds, although we cannot name a separating element of `a` here.

After calling `push_heap`, which just performs some more reorderings of the left part,[38] this property is

[38]We can't and we needn't tell which position q is moved to; the former is indicated in Figure 11.4 by the vague grey triangle.

```
        //@ ghost BeforeSwap:
        swap(a + m - 1u, a + i);
        //@ assert swapped:    SwappedInside{BeforeSwap,Here}(a, m-1, i, n);
        //@ assert reorder:    MultisetUnchanged{BeforeSwap,Here}(a, i+1);
        //@ assert reorder:    MultisetUnchanged{Pre,Here}(a, i+1);
        //@ assert unchanged:  Unchanged{Pre,Here}(a, i+1, n);
        //@ assert lower:      a[m-1] < a[i];
        //@ assert lower:      \forall integer k; 0 <= k < m ==> LowerBound(a, m, i
            +1, a[k]);
        //@ assert upper:      UpperBound(a, 0, m-1, a[0]);

        /*@
          assigns              a[0..m-1];
          ensures heap:        Heap(a, m);
          ensures reorder:     MultisetUnchanged{Old,Here}(a, m);
          ensures unchanged:   Unchanged{Old,Here}(a, m, i+1);
          ensures unchanged:   Unchanged{Old,Here}(a, i+1, n);
        */
        push_heap(a, m);
        //@ assert upper:      UpperBound(a, 0, m,   a[0]);
        //@ assert lower:      LowerBound(a, m, i+1, a[0]);
      }

    //@ assert partition: Partition(a, m, n);
    /*@
      assigns                a[0..m-1];
      ensures reorder:       MultisetUnchanged{Old,Here}(a, m);
      ensures reorder:       MultisetUnchanged{Old,Here}(a, m, n);
      ensures increasing:    Increasing(a, m);
    */
    sort_heap(a, m);
    //@ assert reorder:   MultisetUnchanged{Pre,Here}(a, n);
    //@ assert partition: Partition(a, m, n);
  }
}
```

Listing 11.9: The Implementation of `partial_sort` (2)

preserved. Moreover, we now know again that `a[0]` has become an upper bound of the left part, and hence a separating element between `a[0..m-1]` and `a[m..i]`; that is, the loop invariants `upper` and `lower` have been re-established. These two invariants together are eventually used to prove the property `partition` of the contract.

Compared to its size, the algorithm makes a lot of procedure calls; in this respect it is closer to real-life software than most other algorithms of this tutorial. Therefore, we use it to illustrate a methodical point: For almost every procedure call, we give the callee's contract, tailored to its actual parameters, as a statement contract of the call. For example, everything we know from the `pop_heap` contract, instantiated to the particular situation, is documented in the first statement contract (Listing 11.8). In contrast, we use `assert` clauses to indicate intermediate reasoning to obtain subsequently needed properties.

Our implementation has a worst-case time complexity of $O((n + m) \cdot \log m)$. On the other hand, an implementation that ignores `m` and just sorts `a[0..n-1]` also satisfies the contract in Listing 11.6, and may have $O(n \cdot \log n)$ complexity. Some arithmetic shows that `partial_sort` performs better than plain sort if, and only if, $\log m < \frac{n}{m} \cdot \log\left(\frac{n}{m}\right)$, that is, if n is sufficiently larger than m.

Lemmas used during verification

The lemmas in Listing 11.10 are used in proofs of properties and annotations related to the loop invariants `upper` and `lower`.

```
/*@
  axiomatic PartitionLemmas
  {
    lemma Reorder_Match{K,L}:
      \forall value_type *a, integer n, i;
        0 < n                        ==>
        0 <= i < n                   ==>
        MultisetUnchanged{K,L}(a, n)  ==>
        SomeEqual{K}(a, n, \at(a[i],L));

    lemma Reorder_LowerBound{K,L}:
      \forall value_type* a, integer n, value_type v;
        0 <= n                       ==>
        MultisetUnchanged{K,L}(a, n)  ==>
        LowerBound{K}(a, n, v)        ==>
        LowerBound{L}(a, n, v);

    lemma Reorder_LowerBounds{K,L}:
      \forall value_type* a, integer m, n;
        0 < m <= n                                ==>

        (\forall integer k; 0 <= k < m  ==>
          LowerBound{K}(a, m, n, \at(a[k],K)))   ==>
        MultisetUnchanged{K,L}(a, 0, m)          ==>
        Unchanged{K,L}(a, m, n)                  ==>
        LowerBound{L}(a, m, n, \at(a[0],L));

    lemma Reorder_UpperBound{K,L}:
      \forall value_type* a, integer n, value_type v;
        0 <= n                       ==>
        MultisetUnchanged{K,L}(a, n)  ==>
        UpperBound{K}(a, n, v)        ==>
        UpperBound{L}(a, n, v);
  }
*/
```

Listing 11.10: Some lemma to support `partial_sort`

- Lemma `Reorder_Match` states that a value `a[i]` taken from a range `a[0..n-1]` after some reordering must have been in that range already before reordering. It is used to prove the lemmas above.
- Lemma `Reorder_LowerBound` informally says that a lower bound `v` of a range `a[0..n-1]` keeps its property even after the range is reordered.
- Dually, lemma `Reorder_UpperBound` says that reordering a range doesn't affect any of its upper bounds.
- Lemma `Reorder_LowerBounds` describes a more particular situation: if each element in `a[0..m-1]` is known to be a lower bound of `a[m..n-1]`, and the former range is reordered while the latter is kept untouched, then `a[0]` will still be a lower bound of `a[m..n-1]`. We employ this lemma to infer that, after `push_heap` was called, the new heap maximum `a[0]`, is a lower bound of `a[m..i]`,

The proof of Reorder_Match relies on an equivalence between the predicates SomeEqual and Count which we express with the lemmas SomeEqual_Count and Count_SomeEqual from Listing 11.11

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

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

Listing 11.11: The lemmas SomeEqual_Count and Count_SomeEqual

We also rely on the lemmas, SwappedInsideReorder and SwappedInsidePreserve from Listing 7.59 in order to verify that the loop invariant reorder is preserved.

11.3. The bubble_sort algorithm

The bubble_sort algorithm traverses the given array a[0..n-1] from left to right, maintaining a right-adjusted, constantly growing range a[n-i..n-1] that is already in increasing order. We achieve this range by iterating through the array and swapping two adjacent elements, if their respective value are in the wrong order.

11.3.1. Formal specification of bubble_sort

The Listing 11.12 shows our (generic sorting) contract for bubble_sort.

```
/*@
  requires valid:       \valid(a + (0..n-1));
  assigns               a[0..n-1];
  ensures increasing:   Increasing(a, n);
  ensures reorder:      MultisetUnchanged{Old,Here}(a, n);
 */
void
bubble_sort(value_type* a, size_type n);
```

Listing 11.12: Formal specification of bubble_sort

11.3.2. Implementation of bubble_sort

Our implementation of bubble_sort is shown in Listing 11.13. As it is typical for bubble_sort, the implementation uses two nested loops.

We first discuss the verification of the fact that bubble_sort produces an increasing array. For this we introduce for the *outer loop* the invariant increasing. This loop annotation states that the subrange a[n-i+1..n-1] is in increasing order. An important ingredient on the verification of the increasing property is the claim that that the first element a[n-i+1] of the already sorted subrange is an upper bound of *all* elements left of it. This claim is encoded in the loop invariant upper of the outer bound. In order to support this claim up we exploit the fact that the index j of the inner loop points to the maximum element of the subrange a[0..j]. We formalize this last property in the loop invariant max.

Note that the loop invariants increasing and upper occur also in the inner loop. This shall “assure” the outer loop that the inner loop really preserves these properties.

```
void
bubble_sort(value_type* a, size_type n)
{
  if (0 < n) {
    /*@
      loop invariant bound:           1 <= i <= n;
      loop invariant increasing:      Increasing(a, n-i+1, n);
      loop invariant upper:           1 < i ==> UpperBound(a, n-i+1, a[n-i+1]);
      loop invariant reorder:         MultisetUnchanged{Pre,Here}(a, n);
      loop assigns i, a[0..n-1];
      loop variant n-i;
    */
    for (size_type i = 1u; i < n; ++i) {
      /*@
        loop invariant bound:         0 <= j <= n-i;
        loop invariant increasing:    Increasing(a, n-i+1, n);
        loop invariant upper:         1 < i ==> UpperBound(a, n-i+1, a[n-i+1]);
        loop invariant max:           MaxElement(a, j+1, j);
        loop invariant reorder:       MultisetUnchanged{LoopEntry,Here}(a, j+1);
        loop invariant reorder:       Unchanged{LoopEntry,Here}(a, j+1, n);
        loop assigns                  j, a[0..n-1];
        loop variant n-j;
      */
      for (size_type j = 0u; j < n - i; ++j) {
        if (a[j] > a[j + 1]) {
          swap(&a[j], &a[j + 1]);
          //@ assert swapped: SwappedInside{LoopCurrent,Here}(a, j, j+1, n);
        }
      }
    }
  }
}
```

Listing 11.13: Implementation of `bubble_sort`

We now discuss briefly the verification of the postcondition `reorder`. In each iteration of the outer loop various elements of the not yet sorted subrange `a[0..n-1]` are swapped with their respective neighbour. More specifically, we know for the iteration `j` of the *inner loop* that while subrange `a[0..j]` has been been rearranged, the subrange `a[j+1..n-1]` has not been modified yet. Together this ensures that the loop invariant `reorder` holds for the *outer loop*.

Note the assertion `swapped` that uses the predicate `SwappedInside` from Listing 7.59 in order to describe the effects of the calls to `swap` inside an array. In order to derive from this description the properties `reorder` of the inner loop we also rely here on the lemma `SwappedInsidePreserve` from Listing 7.59.

11.4. The `selection_sort` algorithm

Our version of the `selection_sort` algorithm has the signature

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

The `selection_sort` algorithm sorts an array in increasing order, left to right, by selecting in each step the minimum element of the remaining segment and *swaps* it with its first element. This implies that each member of the increasingly ordered initial segment is less or equal than each member of the remaining segment.

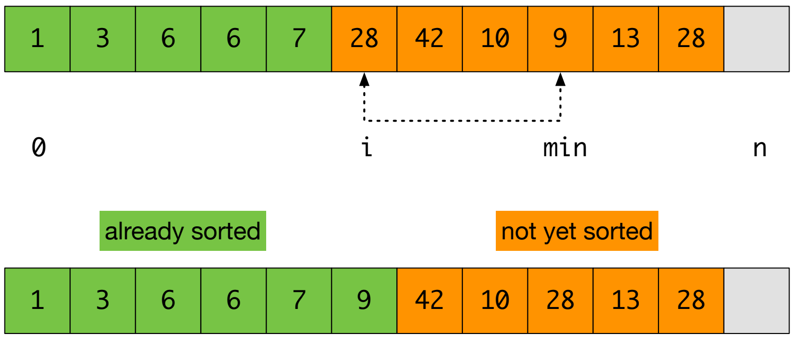


Figure 11.14.: An iteration of `selection_sort`

Figure 11.14 shows a typical situation in an example run. The algorithm will swap the `28` at position `i` with the `9` at position `min` to extend the increasingly ordered initial segment one field to the right.

11.4.1. Formal specification of `selection_sort`

Listing 11.15 shows the ACSL specification of `selection_sort`.

```
/*@
   requires valid:       \valid(a + (0..n-1));
   assigns               a[0..n-1];
   ensures reorder:      MultisetUnchanged{Old,Here}(a, n);
   ensures increasing:   Increasing(a, n);
*/
void
selection_sort(value_type* a, size_type n);
```

Listing 11.15: Formal specification of `selection_sort`

11.4.2. Implementation of selection_sort

The implementation of selection_sort is shown in Listing 11.16. We use min_element from Section 5.8 to find the minimum element of the remaining array segment.

```
void
selection_sort(value_type* a, size_type n)
{
  /*@
    loop invariant bound:       0 <= i <= n;
    loop invariant reorder:     MultisetUnchanged{Pre,Here}(a, n);
    loop invariant increasing:  Increasing(a, i);
    loop invariant increasing:  0 < i  ==> LowerBound(a, i, n, a[i-1]);
    loop assigns   i, a[0..n-1];
    loop variant   n - i;
  */
  for (size_type i = 0u; i < n; ++i) {
    const size_type sel = i + min_element(a + i, n - i);

    if (i < sel) {
      /*@
         assigns          a[sel], a[i];
         ensures swapped: SwappedInside{Old,Here}(a, i, sel, n);
      */
      swap(a + sel, a + i);
    }

    //@ assert reorder: MultisetUnchanged{LoopCurrent,Here}(a, n);
    //@ assert reorder: MultisetUnchanged{Pre,Here}(a, n);
  }
}
```

Listing 11.16: Implementation of selection_sort

The loop invariants increasing and lower establish that the initial segment a[0..i-1] is in increasing order and, respectively, state that a[i-1] is a lower bound of the remaining segment a[i..n-1]. Since the min_element call uses an address offset, we had to employ again the *shift lemmas* from Listing 6.12.

The loop invariant reorder, on the other hand, states that the multiset of values in the array a are only *rearranged* during the algorithm. While this is intuitively most obvious (as the call to the swap routine, from Section 7.3, is the only code that modifies a), it took considerable effort to prove it formally; including a statement contract that captures the effects of calling swap.

The main reason for introducing the statement contract is that it *transforms* the postcondition of the call to swap from Listing 7.5 into the hypotheses for the lemma SwappedInsideReorder in Listing 7.59. This lemma, which relies on the lemmas from Listing 7.56, captures the fact that *swapping two elements of an array* is a *reordering*.

11.5. The insertion_sort algorithm

Like selection_sort, the algorithm insertion_sort traverses the given array a[0..n-1] left to right, maintaining a left-adjusted, constantly increasing range a[0..i-1] that is already in increasing order.

Unlike selection_sort, however, insertion_sort adds a[i] to the initial segment in the ith step (see Figure 11.17). It determines the (rightmost) appropriate position to insert a[i] by a call to upper_bound (Section 6.2), and then uses rotate (Section 7.10) to perform a *circular shift* to establish the insertion.

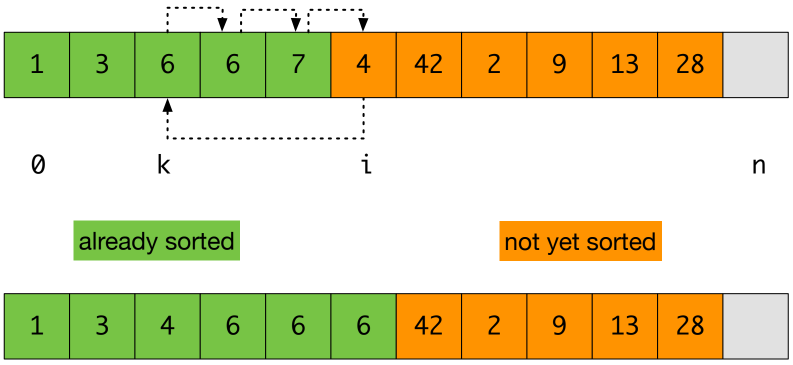


Figure 11.17.: An iteration of insertion_sort

11.5.1. Formal specification of insertion_sort

Listing 11.18 shows our (generic sorting) contract for insertion_sort.

```
/*@
   requires valid:       \valid(a + (0..n-1));
   assigns               a[0..n-1];
   ensures reorder:      MultisetUnchanged{Old,Here}(a, n);
   ensures increasing:   Increasing(a, n);
*/
void
insertion_sort(value_type* a, size_type n);
```

Listing 11.18: Formal specification of insertion_sort

11.5.2. Implementation of insertion_sort

The implementation of insertion_sort is shown in Listing 11.19. We used an ACSL statement contract to specify those aspects of the rotate contract that are needed here. Properties related to the result of insertion_sort being in increasing order are labelled increasing. Properties related to the rearrangement of elements are labelled reorder and, whenever their order isn't changed, unchanged.

```
void
insertion_sort(value_type* a, size_type n)
{
  /*@
    loop invariant bound:         0 <= i <= n;
    loop invariant reorder:       MultisetUnchanged{Pre,Here}(a, 0, i);
    loop invariant unchanged:     Unchanged{Pre,Here}(a, i, n);
    loop invariant increasing:    Increasing(a, i);
    loop assigns   i, a[0..n-1];
    loop variant   n - i;
  */
  for (size_type i = 0u; i < n; ++i) {
    const size_type k = upper_bound(a, i, a[i]);
    //@ assert bound:   0 <= k <= i;
    /*@
       requires increasing: UpperBound(a, k, a[i]);
       requires increasing: StrictLowerBound(a, k, i, a[i]);
       requires increasing: Increasing(a, k, i);
       assigns              a[k..i];
       ensures unchanged:   Unchanged{Old,Here}(a, 0, k);
       ensures unchanged:   Unchanged{Old,Here}(a, i+1, n);
       ensures reorder:     MultisetUnchanged{Old,Here}(a, 0, k);
       ensures reorder:     EqualRanges{Old,Here}(a, k, i, k+1);
       ensures reorder:     EqualRanges{Old,Here}(a, i, i+1, k);
       ensures increasing:  Increasing(a, 0, k);
       ensures increasing:  UpperBound(a, k, a[k]);
    */
    rotate(a + k, i - k, i - k + 1u);
    //@ assert increasing:  StrictLowerBound(a, k+1, i+1, a[k]);
    //@ assert increasing:  Increasing(a, k+1, i+1);
    //@ assert increasing:  Increasing(a, i+1);
    //@ assert reorder:     MultisetUnchanged{Pre,Here}(a, i+1);
  }
}
```

Listing 11.19: Implementation of insertion_sort

When we originally implemented and verified rotate, we hadn't yet in mind to use that function inside of insertion_sort. Consequently, the properties needed for the latter aren't directly provided by the former. One approach to solve this problem is to add the new properties to rotate's contract (Listing 7.26) and repeat its verification proof.[39]

However, if rotate is assumed to be part of a pre-verified library, this approach isn't feasible, since rotate's implementation may not be available for re-verification. Therefore, we used another approach, viz. to prove that rotate's original specification *implies* all the properties we need in insertion_sort. This is another use of the Hoare calculus' implication rule (Section 3.3). We used several lemmas, shown below, to make the necessary implications explicit, and to help the provers to establish them. Some of them

[39]ACSL allows to declare a function several times with different contracts; they are merged into a single one. Alternatively, non-disjoint behaviors, with empty assumes clauses, allow contract merging and provide finer control over the set of hypotheses generated from e.g. an assert.

needed manual proofs by induction.

Lemma `EqualRangesIncreasing` (Listing 11.20) assumes an ordered range `a[m..n-1]` and claims that every (elementwise) equal range range `a[m+p..n+p-1]` is ordered, too. It is needed to establish that the `rotate` call preserves the order of those elements that are shifted upwards (cf. Figure 11.17).

```
/*@
  axiomatic EqualRangeLemmas
  {
    lemma EqualRangesIncreasing{K,L}:
      \forall value_type* a, integer m, n, p;
        Increasing{K}(a, m, n)          ==>
        EqualRanges{K,L}(a, m, n, m+p)  ==>
        Increasing{L}(a, m+p, n+p);

    lemma EqualRangesCount{K,L}:
      \forall value_type *a, v, integer m, n, p;
        0 <= m <= n                   ==>
        EqualRanges{K,L}(a, m, n, p)  ==>
        Count{K}(a, m, n, v) == Count{L}(a, p, p + (n-m), v);
  }
*/
```

Listing 11.20: The lemmas `EqualRangesIncreasing`

Similarly, lemma `EqualRangesCount` says that two elementwise equal ranges `a[m..n-1]` and `a[p..p+n-m-1]` will result in the same occurrence count, for each value `v`. It is useful in the proof of the previous lemma, `CircularShift_MultisetUnchanged`, since the predicate `MultisetUnchanged` (Listing 7.56) is defined via the `Count` function (Listing 4.36).

Lemma `CircularShift_StrictLowerBound` (Listing 11.21) is used to prove that the range `a[k..i-1]` having `a[i]` as strict lower bound before our `rotate` call ensures that it has `a[k]` as such a bound after the call. Note that this lemma reflects that `rotate` is uses as a *circular shift* at the call site. Similarly, Lemma `CircularShift_MultisetUnchanged` establishes that a circular shift just reorders the range it is applied to.

```
/*@
   axiomatic CircularShiftLemmas
   {
     lemma CircularShift_StrictLowerBound{K,L}:
       \forall value_type* a, integer m, n;
         StrictLowerBound{K}(a, m, n, \at(a[n],K))  ==>
         EqualRanges{K,L}(a, m, n, m+1)             ==>
         EqualRanges{K,L}(a, n, n+1, m)             ==>
         StrictLowerBound{L}(a, m+1, n+1, \at(a[m],L));

     lemma CircularShift_MultisetUnchanged{K,L}:
       \forall value_type* a, integer m, n;
         0 <= m <= n                     ==>
         EqualRanges{K,L}(a, m, n, m+1)  ==>
         EqualRanges{K,L}(a, n, n+1, m)  ==>
         MultisetUnchanged{K,L}(a, m, n+1);
   }
*/
```

Listing 11.21: Some lemmas on circular shifts

11.6. The heap_sort algorithm

The heap_sort algorithm has the signature

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

It relies upon the heap data structure discussed in Chapter 10 to efficiently transform the array into increasing order.

11.6.1. Formal specification of heap_sort

Listing 11.22 shows the ACSL specification of heap_sort.

```
/*@
   requires valid:       \valid(a + (0..n-1));
   assigns               a[0..n-1];
   ensures reorder:      MultisetUnchanged{Old,Here}(a, n);
   ensures increasing:   Increasing(a, n);
*/
void
heap_sort(value_type* a, size_type n);
```

Listing 11.22: Formal specification of heap_sort

11.6.2. Implementation of heap_sort

The implementation of heap_sort, shown in Listing 11.23, is straightforward. Given the input array a, we use make_heap to arrange it into a heap; after that, we use sort_heap to turn this heap into increasing order.

```
void
heap_sort(value_type* a, size_type n)
{
  make_heap(a, n);
  sort_heap(a, n);
}
```

Listing 11.23: Implementation of heap_sort

11.7. The `merge` algorithm

Our version of the `merge` algorithm from the C++ standard library [20, 28.7.5] has the following signature.

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

The `merge` algorithm is a part of the *merge sort* algorithm. It operates on the second step to merge two increasingly ordered sub-arrays into a new one. The algorithm merges two increasingly ordered arrays `a[0..n-1]` and `b[0..m-1]`, respectively. The merged values are stored in the output array that starts at `result` which must be able to hold $m + n$ values of both input arrays.

11.7.1. Formal specification of `merge`

Listing 11.24 shows the ACSL specification of `merge`. The specification expects the input arrays of the proper size and in increasing order and the output array of enough size to contain all the input elements. The input arrays should not overlap with the output array. In the current edition of this guide, we prove only that the resulting array is in increasing order. Future editions will contain additional postconditions stating that the result array consists of reordered input elements and the stability of the algorithm, i.e., the same elements of the input arrays preserve their order in the output array.

```
/*@
  requires bound:        n + m <= SIZE_TYPE_MAX;
  requires valid:        \valid_read(a + (0..n-1));
  requires valid:        \valid_read(b + (0..m-1));
  requires valid:        \valid(result + (0..n+m-1));
  requires sep:          \separated(a + (0..n-1), result + (0..n+m-1));
  requires sep:          \separated(b + (0..m-1), result + (0..n+m-1));
  requires increasing:   WeaklyIncreasing(a, n);
  requires increasing:   WeaklyIncreasing(b, m);
  assigns                result[0 .. n+m-1];
  ensures  increasing:   Increasing(result, n + m);
 */
void
merge(const value_type* a, size_type n,
      const value_type* b, size_type m,
      value_type* result);
```

Listing 11.24: The specification of `merge`

11.7.2. More Lemmas for `WeaklyIncreasing`

We discuss here more lemmas from Listing 11.3.

- Lemma `WeaklyIncreasingAddElement` defines the way a weakly increasing array can be constructed.
- Lemma `WeaklyIncreasingShift` states the equality with respect to the `WeaklyIncreasing` property of the elements of the array `a[0..n]` and the elements of the array section `a[n..m+n]`.

- Lemma `EqualRangesWeaklyIncreasing` states that if an array is weakly increasing, then another array, whose elements are in a one-to-one correspondence with the original array, is also weakly increasing.
- Lemma `WeaklyIncreasingJoin` defines the conditions that two consequent weakly increasing ranges can be viewed as merged weakly increasing range.

Lemma `WeaklyIncreasingShift` requires a manual proof with Coq. The other lemmas can be proved automatically. Note that we also rely on the other lemma from Listing 11.3 to verify the postcondition.

11.7.3. Implementation of `merge`

The implementation of `merge`, shown in Listings 11.25 and 11.26, is straightforward. The algorithm operates by traversing both input arrays. On each iteration it writes the smaller of both elements into the result array, thus constructing an increasingly ordered array. If the algorithm reaches the end of one of the input arrays, it just copies the rest elements of the other array to the result array.

```
void
merge(const value_type* a, size_type n,
      const value_type* b, size_type m,
      value_type* result)
{
  size_type i = 0;
  size_type j = 0;
  size_type x = 0;

  if (0 < n || 0 < m) {
    /*@ loop invariant 0 <= i <= n;
        loop invariant 0 <= j <= m;
        loop invariant x == i + j;
        loop invariant 0 <= x <= n + m - 1;
        loop invariant order:  \forall integer k; 0 <= k < x && i < n ==>
                                   result[k] <= a[i];
        loop invariant order:  \forall integer k; 0 <= k < x && j < m ==>
                                   result[k] <= b[j];
        loop invariant sorted: WeaklyIncreasing(result, x);
        loop assigns i, j, x, result[0 .. n+m-1];
        loop variant (n + m) - (i + j);
     */
    while (i < n && j < m) {
      if (a[i] < b[j]) {
        result[x++] = a[i++];
      }
      else {
        result[x++] = b[j++];
      }
    }
```

Listing 11.25: Implementation of `merge` (1)

The listing contains a number of assertions to trigger an application of lemmas by the provers. The **`while`** loop traverses the input arrays and constructs, in accordance with Lemma `WeaklyIncreasingAddElement`, the resulting weakly increasing array. After the loop, the algorithm copies the remaining elements to the resulting array.

```
    //@ assert i == n ^^ j == m;
    //@ assert i < n ^^ j < m;
    //@ assert WeaklyIncreasing(result, 0, x);

    if (i < n) {
      //@ assert 0 < x ==> result[x-1] <= a[i];
      //@ assert WeaklyIncreasing(a + i, 0, n - i);
      copy(a + i, n - i, result + x);
      //@ assert result[x] == a[i];
      /*@ assert WeaklyIncreasing(a + i, 0, n - i) &&
                 EqualRanges{Here,Here}(a + i, 0, n - i, result + x) ==>
                   WeaklyIncreasing(result + x, 0, n - i);
       */
      //@ assert n - i + x == n + m;
    }
    else {
      //@ assert 0 < x ==> result[x-1] <= b[j];
      //@ assert WeaklyIncreasing(b + j, 0, m - j);
      copy(b + j, m - j, result + x);
      //@ assert result[x] == b[j];
      /*@ assert WeaklyIncreasing(b + j, 0, m - j) &&
                 EqualRanges{Here,Here}(b + j, 0, m - j, result + x) ==>
                   WeaklyIncreasing(result + x, 0, m - j);
       */
      //@ assert m - j + x == n + m;
    }

    //@ assert WeaklyIncreasing(result, x, n + m);
    //@ assert x > 0 ==> result[x-1] <= result[x];
    //@ assert WeaklyIncreasing(result, 0, n + m);
  }
  else {
    return;
  }
}
```

Listing 11.26: The Implementation of merge (2)

We also use the following lemmas from Listings 11.3 to support the verification of several properties.

- Lemma EqualRangesWeaklyIncreasing is used to show that the copied elements from one of the input arrays preserve the WeaklyIncreasing property.
- Lemma WeaklyIncreasingJoin is used to extend the WeaklyIncreasing property of the two sub-ranges of the resulting array over the whole range. Lemma WeaklyIncreasingShift is used in-between for array offset arithmetic.
- Finally, Lemma WeaklyIncreasingImpliesIncreasing is used to prove the output array is in increasing order.

Part V.

Verification of data structures

12. The `stack` data type

So far we have used the ACSL specification language for 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 only interested in the properties of a single function but also in properties describing how several function play together.

The C++ Standard Library provides a generic container adaptor `stack` [20, §26.6.6] whose signature and behavior we try to follow as far as our C implementation it allows. For a more detailed discussion of our approach to the formal verification of `stack` we refer to Kim Völlinger's thesis [31].

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

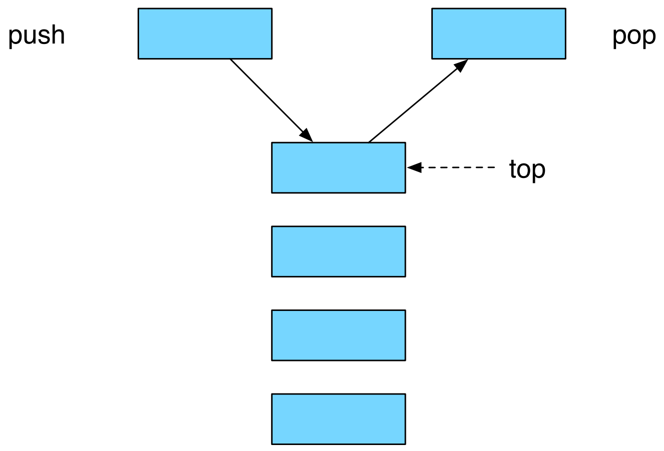
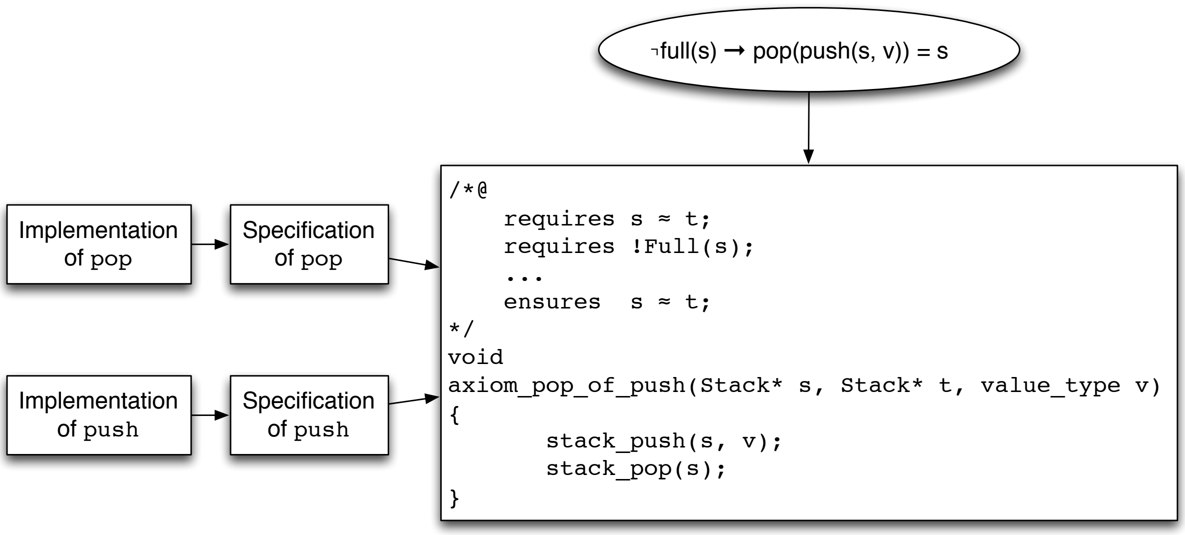


Figure 12.1.: Push and pop on a stack

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.

12.1. Methodology overview

Figure 12.2 gives an overview of our methodology to specify and verify abstract data types (verification of one axiom shown only).



```
/*@
    requires s ≈ t;
    requires !Full(s);
    ...
    ensures  s ≈ t;
*/
void
axiom_pop_of_push(Stack* s, Stack* t, value_type v)
{
       stack_push(s, v);
       stack_pop(s);
}
```

Figure 12.2.: Methodology Overview

What we will basically do is:

1. specify axioms about how the stack functions should interact with each other (Section 12.2),
2. define a basic implementation of C data structures (only one in our example, viz. **struct** Stack; see Section 12.3) and some invariants the instances of them have to obey (Section 12.4),
3. provide for each stack function an ACSL contract and a C implementation (Section 12.6),
4. verify each function against its contract (Section 12.6),
5. transform the axioms into ACSL-annotated C code (Section 12.7), and
6. verify that code, using access function contracts and data-type invariants as necessary (Section 12.7).

Section 12.5 provides an ACSL-predicate deciding whether two instances of a **struct** Stack are considered to be equal (indication by "≈" in Figure 12.2), while Section 12.6.1 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.

12.2. Stack axioms

To specify the interplay of the stack access functions, we use a set of axioms[40], 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*.

12.2.1. Stack initialization

After a stack has been initialized its size is 0.

$$\text{size}(\text{init}(s)) = 0. \tag{12.1}$$

The auxiliary functions empty and full are defined as follows

$$\text{empty}(s), \qquad \text{iff} \qquad \text{size(s)} = 0, \tag{12.2}$$
$$\text{full}(s), \qquad \text{iff} \qquad \text{size(s)} = c. \tag{12.3}$$

We expect that for every stack s the following condition holds

$$0 \le \text{size}(s) \le c. \tag{12.4}$$

[40]There is an analogy in geometry: Euclid (e.g. [32]) invented the use of axioms there, but still kept definitions of *point*, *line*, *plane*, etc. Hilbert [33] recognized that the latter are not only unformalizable, but also unnecessary, and dropped them, keeping only the formal descriptions of relations between them.

12.2.2. 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

$$\text{size}(\text{push}(s, v)) = \text{size}(s) + 1, \qquad \text{if} \quad \neg\text{full}(s). \tag{12.5}$$

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

$$\text{top}(\text{push}(s, v)) = v, \qquad \text{if} \quad \neg\text{full}(s). \tag{12.6}$$

12.2.3. 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

$$\text{size}(\text{pop}(s)) = \text{size}(s) - 1, \qquad \text{if} \quad \neg\text{empty}(s). \tag{12.7}$$

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

$$\text{pop}(\text{push}(s, v)) = s, \qquad \text{if} \quad \neg\text{full}(s). \tag{12.8}$$

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.

$$\text{push}(\text{pop}(s), \text{top}(s)) = s, \qquad \text{if} \quad \neg\text{empty}(s). \tag{12.9}$$

12.2.4. 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

$$\text{push}(s, \text{err}) = s. \tag{12.10}$$

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

$$\begin{aligned}
\text{size}(s) &= \text{size}(\text{pop}(\text{push}(s, \text{err}))) && \text{by 12.8} \\
&= \text{size}(\text{pop}(s)) && \text{as assumed in 12.10} \\
&= \text{size}(s) - 1 && \text{by 12.7}
\end{aligned}$$

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

12.3. 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 12.3.

```
struct Stack
{
  value_type* obj;

  size_type   capacity;

  size_type   size;
};

typedef struct Stack Stack;
```

Listing 12.3: 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 12.4 which attempts to interpret this definition according to Figure 12.1.

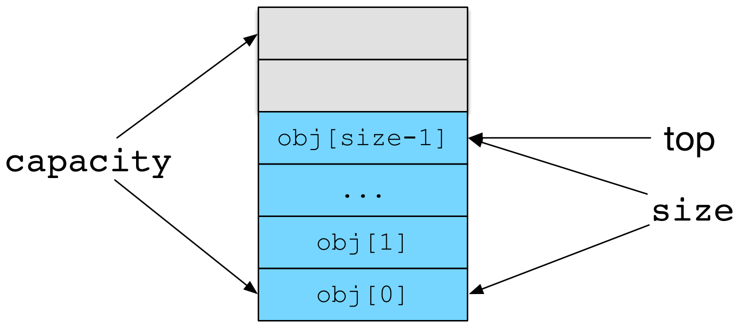


Figure 12.4.: Interpreting the data structure stack

Based on the stack functions from Section 12.2, we declare in Listing 12.5 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 12.5: Declaration of functions of type stack

Most of these functions directly correspond to methods of the C++ std::stack template class [20, §26.6.6.1]. 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.

12.4. Stack invariants

Not every possible instance of type stack is considered a valid one, e.g., with our definition of stack in Listing 12.3, Stack s = {{0,0,0,0},4,5} is not. In Listing 12.6, we present basic logic functions and predicates that we will use throughout this chapter In particular, we define the predicate StackInvariant that discriminates valid and invalid instances.

```
/*@
  axiomatic StackInvariant
  {
    logic integer
    StackCapacity{L}(Stack* s) = s->capacity;

    logic integer
    StackSize{L}(Stack* s) = s->size;

    logic value_type*
    StackStorage{L}(Stack* s) = s->obj;

    logic integer
    StackTop{L}(Stack* s) = s->obj[s->size-1];

    predicate
    StackEmpty{L}(Stack* s) =  StackSize(s) == 0;

    predicate
    StackFull{L}(Stack* s)  =  StackSize(s) == StackCapacity(s);

    predicate
    StackInvariant{L}(Stack* s) =
      0 < StackCapacity(s) &&
      0 <= StackSize(s) <= StackCapacity(s) &&
      \valid(StackStorage(s) + (0..StackCapacity(s)-1)) &&
      \separated(s, StackStorage(s) + (0..StackCapacity(s)-1));
  }
*/
```

Listing 12.6: Logic description of stack

We start, with the auxiliary logic function StackCapacity, StackSize and StackStorage which we can use in specifications to refer to the fields capacity, size and obj of stack, respectively. This listing also contains the logic function StackTop 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. We intentionally use here integer as a return value of these logic functions. Inaccurate use of logic functions with bounded types in axioms with arithmetic operations may lead to inconsistencies.

We also introduce in Listing 12.6 the predicates StackEmpty and StackFull that express the concepts of empty and full stacks by referring to a stack's size and capacity (see Equations (12.2) and (12.3)).

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

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

These invariants are formalized in the predicate `StackInvariant` of Listing 12.6.

Note how the use of the previously defined logic functions and predicates allows us to define the stack invariant without directly referring to the fields of `stack`.

We sometimes wish to express that there is no *memory aliasing* between two stacks. If there were aliasing, then 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 12.7 the predicate `StackSeparated`.

```
/*@
  axiomatic StackUtility
  {
    predicate
    StackSeparated(Stack* s, Stack* t) =
      \separated(s, s->obj + (0..s->capacity-1),
                 t, t->obj + (0..t->capacity-1));

    predicate
    StackUnchanged{K,L}(Stack* s) =
      StackSize{K}(s)     == StackSize{L}(s)      &&
      StackStorage{K}(s)  == StackStorage{L}(s)   &&
      StackCapacity{K}(s) == StackCapacity{L}(s)  &&
      Unchanged{K,L}(StackStorage{K}(s), StackSize{K}(s));
  }
*/
```

Listing 12.7: The predicate `StackSeparated`

Listing 12.7 also contains the predicate `StackUnchanged` that we will use to describe cases that the contents of a stack hasn't changed.

12.5. 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* [34, 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 `StackEqual` as in Listing 12.8.

```
/*@
  axiomatic StackEquality
  {
    predicate
    StackEqual{S,T}(Stack* s, Stack* t) =
      StackSize{S}(s) == StackSize{T}(t) &&
      EqualRanges{S,T}(StackStorage{S}(s), StackSize{S}(s), StackStorage{T}(t));

    lemma StackEqualReflexive{S} :
      \forall Stack* s; StackEqual{S,S}(s, s);

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

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

Listing 12.8: Equality of stacks

Our use of labels in Listing 12.8 makes the specification somewhat hard to read (in particular in the last line where we reuse the predicate `EqualRanges` from Page 44). However, this definition of `StackEqual` will allow us later to compare the same stack object at different points of a program. The logical expression `StackEqual{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.[41]

If equal stacks can have different capacities then, according to our definition of the predicate `StackFull` in Listing 12.6, 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 12.9 are considered equal although there is are obvious differences in their internal arrays.

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

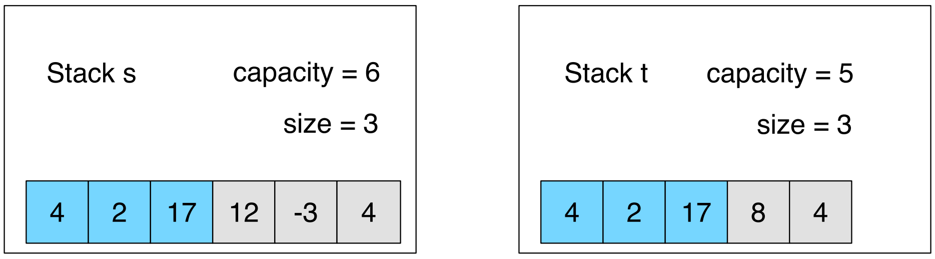


Figure 12.9.: Example of two equal stacks

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.

$$\text{reflexivity} \qquad \forall s \in S : s = s, \tag{12.11a}$$

$$\text{symmetry} \qquad \forall s, t \in S : s = t \implies t = s, \tag{12.11b}$$

$$\text{transitivity} \qquad \forall s, t, u \in S : s = t \wedge t = u \implies s = u. \tag{12.11c}$$

Any relation that satisfies the conditions (12.11) 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 12.8 shows a formalization of these three rules for the relation `StackEqual`; it can be automatically verified that they are a consequence of the definition of `StackEqual` in Listing 12.8.

The two stacks in Figure 12.9 show that an equivalence class of `StackEqual` can contain more than one element.[42] The stacks `s` and `t` in Figure 12.9 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*.[43] We ask, in other words, whether the function is *well-defined* on the set of all equivalence classes of the relation `StackEqual`.[44] The question of well-definition will play an important role when verifying the functions of the `stack` (see Section 12.6).

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

[43] This is why mathematicians know that $\frac{1}{2} + \frac{3}{5}$ equals $\frac{7}{14} + \frac{3}{5}$.

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

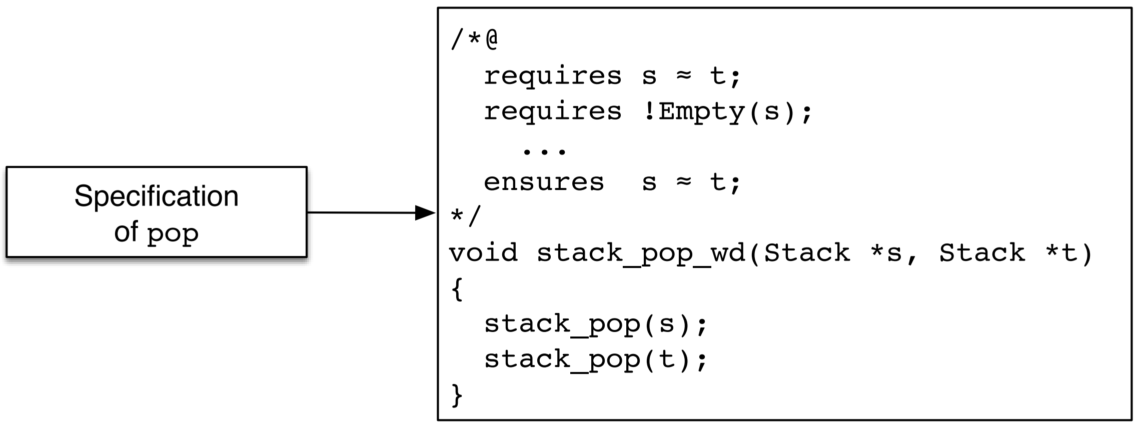
12.6. Verification of stack functions

In this section we verify the functions

- `stack_equal` (Section 12.6.1
- `stack_init` (Section 12.6.2
- `stack_size` (Section 12.6.3)
- `stack_full` (Section 12.6.4)
- `stack_empty` (Section 12.6.5)
- `stack_top` (Section 12.6.6)
- `stack_push` (Section 12.6.7)
- `stack_pop` (Section 12.6.8)

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`[45] accompanied by a an ACSL contract that expresses that the implementation of `stack_foo` is well-defined. Figure 12.10 shows our methodology for the verification of well-definition in the `pop` example, (≈) again indicating the user-defined `stack` equality.



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

Figure 12.10.: 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 12.7 we will show that the *interplay* of these functions satisfy the stack axioms from Section 12.2.

[45]The suffix `_wd` stands for *well definition*

12.6.1. The function `stack_equal`

The function `stack_equal` is the runtime counterpart for the `StackEqual` predicate. The specification of `stack_equal` is shown in Listing 12.11. Note that this specifications explicitly refers to valid stacks.

```
/*@
    requires valid:      \valid(s) && StackInvariant(s);
    requires valid:      \valid(t) && StackInvariant(t);
    assigns              \nothing;
    ensures  equal:      \result == 1  <==>  StackEqual{Here,Here}(s, t);
    ensures  not_equal: \result == 0  <==> !StackEqual{Here,Here}(s, t);
*/
bool
stack_equal(const Stack* s, const Stack* t);
```

Listing 12.11: Specification of `stack_equal`

The implementation of `stack_equal` in Listing 12.12 compares two stacks according to the same rules of predicate `StackEqual`.

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

Listing 12.12: Implementation of `stack_equal`

12.6.2. The function `stack_init`

Listing 12.13 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:     \valid(s);
  requires capacity:  0 < capacity;
  requires storage:   \valid(storage + (0..capacity-1));
  requires sep:       \separated(s, storage + (0..capacity-1));
  assigns             s->obj, s->capacity, s->size;
  ensures  valid:     \valid(s);
  ensures  capacity:  StackCapacity(s) == capacity;
  ensures  storage:   StackStorage(s) == storage;
  ensures  invariant: StackInvariant(s);
  ensures  empty:     StackEmpty(s);
*/
void
stack_init(Stack* s, value_type* storage, size_type capacity);
```

Listing 12.13: Specification of `stack_init`

Listing 12.13 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     = 0u;
}
```

Listing 12.14: Implementation of `stack_init`

12.6.3. The function stack_size

The function stack_size is the runtime version of the logic function StackSize from Listing 12.6 on Page 240. The specification of stack_size in Listing 12.15 simply states that stack_size produces the same result as StackSize.

```
/*@
    requires valid: \valid(s) && StackInvariant(s);
    assigns         \nothing;
    ensures   size:  \result == StackSize(s);
*/
size_type
stack_size(const Stack* s);
```

Listing 12.15: Specification of stack_size

As in the definition of the logic function StackSize the implementation of stack_size in Figure 12.16 simply returns the field size.

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

Listing 12.16: Implementation of stack_size

Listing 12.17 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:  \valid(s) && StackInvariant(s);
  requires valid:  \valid(t) && StackInvariant(t);
  requires equal:  StackEqual{Here,Here}(s, t);
  assigns          \nothing;
  ensures  equal:  \result;
*/
bool
stack_size_wd(const Stack* s, const Stack* t)
{
  return stack_size(s) == stack_size(t);
}
```

Listing 12.17: Well-definition of stack_size

12.6.4. The function `stack_full`

The function `stack_full` is the runtime version of the predicate `StackFull` from Listing 12.6.

```
/*@
    requires valid:    \valid(s) && StackInvariant(s);
    assigns            \nothing;
    ensures  full:     \result == 1  <==>  StackFull(s);
    ensures  not_full: \result == 0  <==> !StackFull(s);
*/
bool
stack_full(const Stack* s);
```

Listing 12.18: Specification of `stack_full`

As in the definition of the predicate `StackFull` the implementation of `stack_full` in Figure 12.19 simply checks whether the size of the stack equals its capacity.

```
bool
stack_full(const Stack* s)
{
  return stack_size(s) == s->capacity;
}
```

Listing 12.19: Implementation of `stack_full`

Note that with our definition of stack equality (Section 12.5) there can be equal stack with different capacities. As a consequence, there can are equal stacks where one is full while the other is not. In other words, `StackFull` is not well-defined.

12.6.5. The function stack_empty

The function stack_empty is the runtime version of the predicate StackEmpty from Listing 12.6.

```
/*@
    requires valid:    \valid(s) && StackInvariant(s);
    assigns            \nothing;
    ensures empty:     \result == 1  <==>  StackEmpty(s);
    ensures not_empty: \result == 0  <==> !StackEmpty(s);
*/
bool
stack_empty(const Stack* s);
```

Listing 12.20: Specification of stack_empty

As in the definition of the predicate StackEmpty the implementation of stack_empty in Figure 12.21 simply checks whether the size of the stack is zero.

```
bool
stack_empty(const Stack* s)
{
  return stack_size(s) == 0u;
}
```

Listing 12.21: Implementation of stack_empty

Listing 12.22 shows our check whether stack_empty is well-defined.

```
/*@
  requires valid:  \valid(s) && StackInvariant(s);
  requires valid:  \valid(t) && StackInvariant(t);
  requires equal:  StackEqual{Here,Here}(s, t);
  assigns          \nothing;
  ensures  equal:  \result;
*/
bool
stack_empty_wd(const Stack* s, const Stack* t)
{
  return stack_empty(s) == stack_empty(t);
}
```

Listing 12.22: Well-definition of stack_empty

12.6.6. The function stack_top

The function stack_top is the runtime version of the logic function StackTop from Listing 12.6. The specification of stack_top in Listing 12.23 simply states that for non-empty stacks stack_top produces the same result as StackTop which in turn just returns the element obj[size-1] of stack.

```
/*@
    requires valid: \valid(s) && StackInvariant(s);
    assigns         \nothing;
    ensures  top:   !StackEmpty(s) ==> \result == StackTop(s);
*/
value_type
stack_top(const Stack* s);
```

Listing 12.23: Specification of stack_top

For a non-empty stack the implementation of stack_top in Figure 12.24 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 - 1u];
  }
  else {
    return s->obj[0u];
  }
}
```

Listing 12.24: Implementation of stack_top

Listing 12.25 shows our check whether stack_top well-defined for non-empty stacks.

```
/*@
  requires valid:  \valid(s) && StackInvariant(s) && !StackEmpty(s);
  requires valid:  \valid(t) && StackInvariant(t) && !StackEmpty(t);
  requires equal:  StackEqual{Here,Here}(s, t);
  assigns          \nothing;
  ensures  equal:  \result;
*/
bool
stack_top_wd(const Stack* s, const Stack* t)
{
  return stack_top(s) == stack_top(t);
}
```

Listing 12.25: Well-definition of stack_top

Since our axioms in Section 12.2 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.

12.6.7. The function stack_push

Listing 12.26 shows the ACSL specification of the function stack_push. In accordance with Axiom (12.5), 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 (12.6)).

```
/*@
  requires valid:      \valid(s) && StackInvariant(s);
  assigns              s->size, s->obj[s->size];

  behavior full:
    assumes            StackFull(s);
    assigns            \nothing;
    ensures valid:     \valid(s) && StackInvariant(s);
    ensures full:      StackFull(s);
    ensures unchanged: StackUnchanged{Old,Here}(s);

  behavior not_full:
    assumes            !StackFull(s);
    assigns            s->size;
    assigns            s->obj[s->size];
    ensures valid:     \valid(s) && StackInvariant(s);
    ensures size:      StackSize(s) == StackSize{Old}(s) + 1;
    ensures top:       StackTop(s) == v;
    ensures storage:   StackStorage(s) == StackStorage{Old}(s);
    ensures capacity:  StackCapacity(s) == StackCapacity{Old}(s);
    ensures not_empty: !StackEmpty(s);
    ensures unchanged: Unchanged{Old,Here}(StackStorage(s), StackSize{Old}(s));

  complete behaviors;
  disjoint behaviors;
*/
void
stack_push(Stack* s, value_type v);
```

Listing 12.26: Specification of stack_push

The implementation of stack_push is shown in Listing 12.27. 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)) {
    //@ assert not_full: s->size < s->capacity;
    s->obj[s->size++] = v;
  }
}
```

Listing 12.27: Implementation of stack_push

Listing 12.28 shows our formalization of the well-definition for 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 use the predicate StackSeparated from Listing 12.7.

```
/*@
  requires valid:     \valid(s) && StackInvariant(s);
  requires valid:     \valid(t) && StackInvariant(t);
  requires equal:     StackEqual{Here,Here}(s, t);
  requires not_full:  !StackFull(s) && !StackFull(t);
  requires sep:       StackSeparated(s, t);
  assigns             s->size, s->obj[s->size];
  assigns             t->size, t->obj[t->size];
  ensures  valid:     StackInvariant(s) && StackInvariant(t);
  ensures  equal:     StackEqual{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:   StackTop(s) == v;
  //@ assert top:   StackTop(t) == v;
  //@ assert equal: EqualRanges{Here,Here}(StackStorage(s), StackSize{Pre}(s),
      StackStorage(t));
}
```

Listing 12.28: Well-definition of stack_push

In order to achieve an automatic verification of the well-definition of stack_push we added in Listing 12.28 the assertions top and equal and introduced the lemma StackPush_Equal from Listing 12.29.

```
/*@
  axiomatic StackLemmas
  {
    lemma StackPush_Equal{K,L}:
      \forall Stack *s, *t;
        StackEqual{K,K}(s,t)                      ==>
        StackSize{L}(s) == StackSize{K}(s) + 1    ==>
        StackSize{L}(s) == StackSize{L}(t)        ==>
        StackTop{L}(s)  == StackTop{L}(t)         ==>
        EqualRanges{L,L}(StackStorage{L}(s),
                         StackSize{K}(s),
                         StackStorage{L}(t))      ==>
        StackEqual{L,L}(s,t);
  }
*/
```

Listing 12.29: The lemma StackPush_Equal

12.6.8. The function `stack_pop`

Listing 12.30 shows the ACSL specification of the function `stack_pop`. In accordance with Axiom (12.7) `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: \valid(s) && StackInvariant(s);
  assigns         s->size;
  ensures  valid: \valid(s) && StackInvariant(s);

  behavior empty:
    assumes            StackEmpty(s);
    assigns            \nothing;
    ensures empty:     StackEmpty(s);
    ensures unchanged: StackUnchanged{Old,Here}(s);

  behavior not_empty:
    assumes            !StackEmpty(s);
    assigns            s->size;
    ensures size:      StackSize(s) == StackSize{Old}(s) - 1;
    ensures full:      !StackFull(s);
    ensures storage:   StackStorage(s) == StackStorage{Old}(s);
    ensures capacity:  StackCapacity(s) == StackCapacity{Old}(s);
    ensures unchanged: Unchanged{Old,Here}(StackStorage(s), StackSize(s));

  complete behaviors;
  disjoint behaviors;
*/
void
stack_pop(Stack* s);
```

Listing 12.30: Specification of `stack_pop`

The implementation of `stack_pop` is shown in Listing 12.31. 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 12.31: Implementation of `stack_pop`

Listing 12.32 shows our check whether stack_pop is well-defined. As in the case of stack_push we use the predicate StackSeparated (Listing 12.7) in order to express that there is no aliasing between the two stack arguments.

```
/*@
  requires valid:  \valid(s) && StackInvariant(s);
  requires valid:  \valid(t) && StackInvariant(t);
  requires equal:  StackEqual{Here,Here}(s, t);
  requires sep:    StackSeparated(s, t);
  assigns          s->size;
  assigns          t->size;
  ensures valid:   StackInvariant(s);
  ensures valid:   StackInvariant(t);
  ensures equal:   StackEqual{Here,Here}(s, t);
*/
void
stack_pop_wd(Stack* s, Stack* t)
{
  stack_pop(s);
  stack_pop(t);
}
```

Listing 12.32: Well-definition of stack_pop

12.7. Verification of stack axioms

In this section we show that the stack functions defined in Section 12.6 satisfy the stack Axioms of Section 12.2.

The annotated code has been obtained from the axioms in a fully systematical way. In order to transform a condition equation $p \to s = t$:

- Generate a clause `requires p`.
- Generate a clause `requires x1 == ... == xn` for each variable `x` with n occurrences in s and t.
- Change the i-th occurrence of `x` to `xi` in s and t.
- Translate both terms s and t to reversed polish notation.
- Generate a clause `ensures y1 == y2`, where `y1` and `y2` denote the value corresponding to the translated s and t, respectively.

This makes it easy to implement a tool that does the translation automatically, but yields a slightly longer contract in our example.

12.7.1. Resetting a stack

Our formulation in ACSL/C of the axiom in Equation (12.1) on Page 235 is shown in Listing 12.33.

```
/*@
  requires valid:  \valid(s);
  requires valid:  \valid(a + (0..n-1));
  requires sep:    \separated(s, a + (0..n-1));
  requires pos:    0 < n;
  assigns          s->obj, s->capacity, s->size;
  ensures  size:   \result == 0;
  ensures  valid:  StackInvariant(s);
*/
size_type
axiom_size_of_init(Stack* s, value_type* a, size_type n)
{
  stack_init(s, a, n);
  return stack_size(s);
}
```

Listing 12.33: Specification of Axiom (12.1)

12.7.2. Adding an element to a stack

Axioms (12.5) and (12.6) describe the behavior of a stack when an element is added.

```
/*@
  requires valid:    \valid(s) && StackInvariant(s);
  requires not_full: !StackFull(s);
  assigns            s->size, s->obj[s->size];
  ensures size:      \result == StackSize{Old}(s) + 1;
  ensures valid:     StackInvariant(s);
*/
size_type
axiom_size_of_push(Stack* s, value_type v)
{
  stack_push(s, v);
  return stack_size(s);
}
```

Listing 12.34: Specification of Axiom (12.5)

Except for the `assigns` clauses, the ACSL-specification refers only to encapsulating logic functions and predicates defined in Section 12.4. If ACSL would provide a means to define encapsulating logic functions returning also sets of memory locations, the expressions in `assigns` clauses would not need to refer to the details of our `stack` implementation.[46] As an alternative, `assigns` clauses could be omitted, as long as the proofs are only used to convince a human reader.

```
/*@
  requires valid:     \valid(s) && StackInvariant(s);
  requires not_full:  !StackFull(s);
  assigns             s->size, s->obj[s->size];
  ensures  top:       \result == v;
*/
value_type
axiom_top_of_push(Stack* s, value_type v)
{
  stack_push(s, v);
  return stack_top(s);
}
```

Listing 12.35: Specification of Axiom (12.6)

[46]In [15, §2.3.4], a powerful sublanguage to build memory location set expressions is defined. We will explore its capabilities in a later version.

12.7.3. Removing an element from a stack

This section shows the Listings for Axioms 12.7, 12.8 and 12.9 which describe the behavior of a stack when an element is removed.

```
/*@
  requires valid:  \valid(s) && StackInvariant(s);
  requires empty:  !StackEmpty(s);
  assigns          s->size;
  ensures   size:  \result == StackSize{Old}(s) - 1;
*/
size_type
axiom_size_of_pop(Stack* s)
{
  stack_pop(s);
  return stack_size(s);
}
```

Listing 12.36: Specification of Axiom (12.7)

```
/*@
  requires  valid:    \valid(s) && StackInvariant(s);
  requires  not_full: !StackFull(s);
  assigns             s->size, s->obj[s->size];
  ensures   equal:    StackEqual{Old,Here}(s, s);
*/
void
axiom_pop_of_push(Stack* s, value_type v)
{
  stack_push(s, v);
  stack_pop(s);
}
```

Listing 12.37: Specification of Axiom (12.8)

```
/*@
  requires  valid:     \valid(s) && StackInvariant(s);
  requires  not_empty: !StackEmpty(s);
  assigns              s->size, s->obj[s->size-1];
  ensures   equal:     StackEqual{Old,Here}(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 12.38: Specification of Axiom (12.9)

Part VI.

Appendices

A. 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 4–12 could be deductively verified.

Within Frama-C, the Frama-C/WP plug-in [1] enables deductive verification of C programs that have been annotated with the ANSI/ISO-C Specification Language (ACSL) [9]. The Frama-C/WP plug-in uses weakest precondition computations to generate proof obligations. To formally prove the ACSL properties, these proof obligations can be submitted to external automatic theorem provers or interactive proof assistants. For the precise settings for Frama-C/WP we employed in this release we refer to Chapter 1.

A.1. Verification settings

Here are the most important options of Frama-C that we used in for almost all functions.[47]

```
-pp-annot
-no-unicode
-wp
-wp-rte
-wp-model Typed
-warn-unsigned-overflow
-warn-unsigned-downcast
-wp-steps 10000
-wp-timeout 2
-wp-coq-timeout 5
```

Note that we use a relative small timeout value for the provers. For a couple of algorithms, however, we had to use a considerably larger timeout.

For the precise versions of the employed provers we refer to Table 1.1 on Page 3.

[47]For the `my_lrand48()` function in `shuffle`, the option `-warn-unsigned-overflow` is disabled as explained in Section 7.18.

A.2. Verification results (sequential)

In the *sequential verification scenario* each proof obligation is processed by a set of automatic and interactive theorem provers that are arranged as a *pipe*.[48] This means that each prover passes on to the next prover only those proof obligations that it could not verify. This *verification pipeline* is shown in Figure A.1.

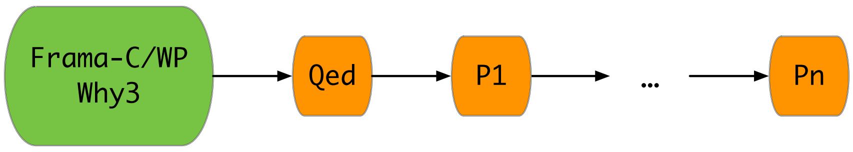


Figure A.1.: Verification pipeline of automatic and interactive theorem provers

For each algorithm we list in the following tables the number of generated verification conditions (VC), the percentage of proven verification conditions, and the number of VC proven by each prover. 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 A.1).

Algorithm		**Verification Conditions**		**Individual Provers**					
				QD	AE	C4	C3	Z3	CQ
`find`	§4.1	25/25	(100%)	16	9	·	·	·	·
`find2`	§4.2	27/27	(100%)	14	13	·	·	·	·
`find3`	§4.3	31/31	(100%)	8	18	1	·	·	4
`find_first_of`	§4.4	41/41	(100%)	30	11	·	·	·	·
`adjacent_find`	§4.5	28/28	(100%)	16	12	·	·	·	·
`mismatch`	§4.6	26/26	(100%)	16	10	·	·	·	·
`equal`	§4.6	7/ 7	(100%)	6	1	·	·	·	·
`search`	§4.7	44/44	(100%)	32	12	·	·	·	·
`search_n`	§4.8	93/93	(100%)	62	31	·	·	·	·
`find_end`	§4.9	34/34	(100%)	21	13	·	·	·	·
`count`	§4.10	28/28	(100%)	7	14	1	·	·	6
`count2`	§4.11	36/36	(100%)	7	18	·	1	·	10

Table A.2.: Results for non-mutating algorithms

[48] Sequential processing is achieved by passing the option `-wp-par 1` to Frama-C/WP.

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	CQ
properties of operator <	§5.1	6/ 6	(100%)	4	2	·	·	·	·
`clamp`	§5.3	22/22	(100%)	18	4	·	·	·	·
`max_element`	§5.4	30/30	(100%)	19	11	·	·	·	·
`max_element2`	§5.5	30/30	(100%)	18	12	·	·	·	·
`max_seq`	§5.6	8/ 8	(100%)	5	3	·	·	·	·
`min_element`	§5.8	30/30	(100%)	18	12	·	·	·	·
`make_pair`	§5.7	4/ 4	(100%)	4	·	·	·	·	·
`minmax_element`	§5.9	60/60	(100%)	43	17	·	·	·	·

Table A.3.: Results for maximum and minimum algorithms

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	CQ
`lower_bound`	§6.1	19/19	(100%)	5	14	·	·	·	·
`upper_bound`	§6.2	19/19	(100%)	7	12	·	·	·	·
`equal_range`	§6.3	22/22	(100%)	17	5	·	·	·	·
`equal_range2`	§6.3	66/66	(100%)	24	37	·	·	2	3
`binary_search`	§6.4	10/10	(100%)	8	2	·	·	·	·
`binary_search2`	§6.4	12/12	(100%)	8	4	·	·	·	·

Table A.4.: Results for binary search algorithms

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	CQ
`fill`	§7.2	14/14	(100%)	4	10	·	·	·	·
`swap`	§7.3	8/ 8	(100%)	5	3	·	·	·	·
`swap_ranges`	§7.4	22/22	(100%)	5	17	·	·	·	·
`copy`	§7.5	15/15	(100%)	4	11	·	·	·	·
`copy_backward`	§7.6	17/17	(100%)	7	10	·	·	·	·
`reverse_copy`	§7.7	17/17	(100%)	4	13	·	·	·	·
`reverse`	§7.8	24/24	(100%)	5	19	·	·	·	·
`rotate_copy`	§7.9	17/17	(100%)	5	12	·	·	·	·
`rotate`	§7.10	24/24	(100%)	10	14	·	·	·	·
`replace_copy`	§7.11	19/19	(100%)	7	12	·	·	·	·
`replace`	§7.12	15/15	(100%)	4	11	·	·	·	·
`remove_copy`	§7.13	23/23	(100%)	9	14	·	·	·	·
`remove_copy2`	§7.14	68/68	(100%)	9	40	3	·	·	16
`remove_copy3`	§7.15	96/96	(100%)	10	58	8	·	·	20
`remove`	§7.16	95/95	(100%)	9	59	6	1	·	20
`shuffle`	§7.17	48/48	(100%)	12	25	3	·	·	8
`random_number`	§7.18	33/33	(100%)	19	14	·	·	·	·

Table A.5.: Results for mutating algorithms

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	CQ
`unique_copy2`	§8.3.2	24/24	(100%)	9	15	·	·	·	·
`unique_copy3`	§8.3.3	27/27	(100%)	9	18	·	·	·	·
`unique_copy4`	§8.3.4	67/67	(100%)	11	50	·	1	·	5

Table A.6.: Results for variants of `unique_copy`

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	CQ
`iota`	§9.1	16/16	(100%)	7	9	·	·	·	·
`accumulate`	§9.2	19/19	(100%)	6	11	·	·	·	2
`inner_product`	§9.3	24/24	(100%)	6	16	·	·	·	2
`partial_sum`	§9.4	42/42	(100%)	9	27	3	·	·	3
`adjacent_difference`	§9.5	35/35	(100%)	11	23	1	·	·	·
`partial_sum_inv`	§9.6	32/32	(100%)	8	17	3	·	·	4
`adjacent_difference_inv`	§9.6	32/32	(100%)	8	18	2	·	·	4

Table A.7.: Results for numeric algorithms

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	CQ
`heap_parent`	§10.3	9/ 9	(100%)	2	7	·	·	·	·
`heap_child_max`	§10.3	24/ 24	(100%)	8	15	·	·	·	1
`is_heap_until`	§10.4	29/ 29	(100%)	6	22	·	·	·	1
`is_heap`	§10.5	14/ 14	(100%)	5	8	·	·	·	1
`push_heap`	§10.6	100/100	(100%)	32	51	10	·	·	7
`pop_heap`	§10.7	104/105	(99%)	49	44	4	·	·	7
`make_heap`	§10.8	48/ 48	(100%)	15	23	3	·	·	7
`sort_heap`	§10.9	57/ 57	(100%)	16	33	1	·	·	7

Table A.8.: Results for heap algorithms

Algorithm		Verification Conditions		Individual Provers					
				QD	AE	C4	C3	Z3	CQ
`is_sorted`	§11.1	20/ 20	(100%)	7	11	·	·	·	2
`partial_sort`	§11.2	118/118	(100%)	39	59	4	1	·	15
`bubble_sort`	§11.3	67/ 67	(100%)	20	36	3	·	·	8
`selection_sort`	§11.4	58/ 58	(100%)	14	29	2	·	3	10
`insertion_sort`	§11.5	68/ 68	(100%)	17	39	3	·	·	9
`heap_sort`	§11.6	33/ 33	(100%)	8	17	1	·	·	7
`merge`	§11.7	268/268	(100%)	166	96	4	·	·	2

Table A.9.: Results for algorithms related to sorting

Algorithm		Verification Conditions		QD	AE	C4	C3	Z3	CQ
				Individual Provers					
`stack_equal`	§12.6.1	18/18	(100%)	7	11	·	·	·	·
`stack_init`	§12.6.2	14/14	(100%)	4	10	·	·	·	·
`stack_size`	§12.6.3	6/ 6	(100%)	1	5	·	·	·	·
`stack_full`	§12.6.4	11/11	(100%)	5	6	·	·	·	·
`stack_empty`	§12.6.5	10/10	(100%)	5	5	·	·	·	·
`stack_top`	§12.6.6	16/16	(100%)	6	10	·	·	·	·
`stack_push`	§12.6.7	41/41	(100%)	25	16	·	·	·	·
`stack_pop`	§12.6.8	29/29	(100%)	17	12	·	·	·	·

Table A.10.: Results for `stack` functions

Algorithm		Verification Conditions		QD	AE	C4	C3	Z3	CQ
				Individual Provers					
`stack_size_wd`	§12.6.3	12/12	(100%)	8	4	·	·	·	·
`stack_empty_wd`	§12.6.5	12/12	(100%)	8	4	·	·	·	·
`stack_top_wd`	§12.6.6	12/12	(100%)	8	4	·	·	·	·
`stack_push_wd`	§12.6.7	15/15	(100%)	3	10	2	·	·	·
`stack_pop_wd`	§12.6.8	12/12	(100%)	6	6	·	·	·	·

Table A.11.: Results for the well-definition of the `stack` functions

Algorithm		Verification Conditions		QD	AE	C4	C3	Z3	CQ
				Individual Provers					
`axiom_size_of_init`	§12.7.1	15/15	(100%)	11	4	·	·	·	·
`axiom_size_of_push`	§12.7.2	12/12	(100%)	9	3	·	·	·	·
`axiom_top_of_push`	§12.7.2	11/11	(100%)	8	3	·	·	·	·
`axiom_size_of_pop`	§12.7.3	11/11	(100%)	8	3	·	·	·	·
`axiom_pop_of_push`	§12.7.3	10/10	(100%)	6	4	·	·	·	·
`axiom_push_of_pop_top`	§12.7.3	15/15	(100%)	9	6	·	·	·	·

Table A.12.: Results for `stack` axioms

B. Changes in previous releases

This chapter describes the changes in previous versions of this document. For the most recent changes we refer to Chapter 1.

The version numbers of this document are related to the versioning of Frama-C [2]. 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[49] of the chemical element after which Frama-C is named.

Y the Frama-C subrelease number

Z the subrelease number of this tutorial

B.1. New in Version 20.0.0 (Calcium, December 2019)

Aside from the above-mentioned version of Frama-C we are using for this release the Why3 platform [3, v1.2.1] and the provers listed in the following table. Note that all automatic provers are use the Why3 interface. In other words, we do not use anymore the native interface for Alt-Ergo.

Prover	Type	Version	Reference
Alt-Ergo	automatic	2.3.0	[4]
CVC4	automatic	1.6	[5]
CVC3	automatic	2.4.1	[6]
Z3	automatic	4.8.6	[7]
Coq	interactive	8.9.1	[8]

Table B.1.: Information on automatic and interactive theorem provers

New examples

- add `bubble_sort`

Improvements

- remove Why3 and Alt-Ergo lemmas from driver
- switch from memory model 'Typed+Ref' to 'Typed'
- the E theorem prover is not yet supported by this version of Frama-C
- no results on parallel verification are reported in this release
- rewrite `random_shuffle` to `shuffle`

[49]See `http://en.wikipedia.org/wiki/Atomic_number`

  - adapt signature of `random_number`
  - add auxiliary function `random_init`
- replace, where applicable, ghost labels by loop labels or statement labels
- remove lemma `SwapImpliesMultisetUnchanged` by using predicate `SwappedInside` and its related lemmas
- improve specification and verification rate of numeric algorithms
  - resolve overloaded version of `Accumulate` into `AccumulateDefault`
  - resolve overloaded version of `AccumulateBounds` into `AccumulateDefaultBounds`
  - improve definition of predicate `PartialSum`
  - add lemmas `Difference_Zero` and `Difference_Next`
  - add predicate `DefaultBounds`
- add assigns in behaviors of maxmin and non-mutating algorithms
  - `find`, `find2`, `find_first_of`, `adjacent_find`, `mismatch`, `search`, `find_end`
  - `max_element`, `max_element2`, `min_element`, `minmax_element`
- rename predicate `Sorted` to `Increasing`; also rename related logic names
  - rename `EqualRangesPreservesSorted` ↦ `EqualRangesPreservesIncreasing`
  - rename `SortedUpperBound` ↦ `IncreasingUpperBound`
  - rename `WeaklySortedAddElement` ↦ `WeaklyIncreasingAddElement`
  - rename `WeaklySortedShift` ↦ `WeaklyIncreasingShift`
  - rename `EqualRangesWeaklySorted` ↦ `EqualRangesWeaklyIncreasing`
  - rename `WeaklySortedJoin` ↦ `WeaklyIncreasingJoin`
  - rename `WeaklySortedLemmas` ↦ `WeaklyIncreasingLemmas`
  - rename `SortedIFFWeaklySorted` ↦ `IncreasingIFFWeaklyIncreasing`
  - rename `SortedImpliesWeaklySorted` ↦ `IncreasingImpliesWeaklyIncreasing`
  - rename `WeaklySortedImpliesSorted` ↦ `WeaklyIncreasingImpliesIncreasing`
  - rename `WeaklySorted` ↦ `WeaklyIncreasing`
  - rename `SortedShift` ↦ `IncreasingShift`
- remove lemma `SortedDownIsHeap`

Open issues

The following algorithms and/or lemmas are not completely verified

- `adjacent_difference_inv`
- `pop_heap`
- `random_number`

- `ReorderImpliesMatch`

B.2. New in Version 19.1.0 (Potassium, October 2019)

This release is intended for Frama-C 19.1 (*Potassium*), issued in September 2019. [2]

Aside from the above-mentioned version of Frama-C we are using for this release the Why3 platform [3, v1.2.0] and the provers listed in the following table. Note that all automatic provers are use the Why3 interface. In other words, we do not use anymore the native interface for Alt-Ergo.

Prover	**Type**	**Version**	**Reference**
Alt-Ergo	automatic	2.3.0	[4]
CVC4	automatic	1.6	[5]
CVC3	automatic	2.4.1	[6]
Z3	automatic	4.8.6	[7]
E	automatic	2.3	[35]
Coq	interactive	8.9.1	[8]

Table B.2.: Information on automatic and interactive theorem provers

Improvements

- Rename arguments of `search` and `find_end` and improve also the description of these algorithms.
- Rename and reorder arguments of `search_n`, make the verification more robust and improve its description.
- Make verification of property `size` of `remove_copy2` more robust.
- Explain role of lemma `RemoveImpliesNotHasValue` in `remove_copy3` and `remove`.
- Simplify definition of `RemoveSize` and `RemovePartition`.
- Make verification of property `reorder` of `partial_sort` more robust.
- Strengthen precondition of `replace_copy`.
- Rename lemma `random_number_modulo` into `RandomNumberModulo`.
- Differentiate between properties `unique` and `solitary` for `unique_copy` examples.
- Simplify the implementation of `is_heap` by calling the new function `is_heap_until`.
- Replace remaining instances of label `Pre` in contracts by `Old`.
- Unify use of `Unchanged` predicate for mutating algorithms.

New examples

- Add the algorithm `clamp` which “clips” a value between a pair of boundary values.
- Add the algorithm `minmax_element` and improve description of other algorithms related to finding minimum and maximum values.

- Add new example `is_heap_until` that generalizes `is_heap`.
- The following examples are not new since they were implicitly used as helper functions for other examples. They are now explicitly listed as examples.
  - `make_pair`
  - `random_number`
  - `heap_parent`
  - `heap_child_max` (formerly known as `heap_maximum_child`

Open issues

The following algorithms and/or lemmas are not completely verified

- `adjacent_difference_inv`
- `partial_sum_inv`
- `pop_heap`
- `ReorderImpliesMatch`

B.3. New in Version 19.0.0 (Potassium, June 2019)

- Structure of document
  - The document is now structured into several parts.
  - The chapter on classic sorting algorithms has been merged into the chapter on sorting.
  - The various variants of `unique_copy` are now grouped into a separate chapter.
- Fix various inconsistencies
  - Change the return types of the logic functions `Accumulate`, `Difference`, `Capacity`, `Size`, `Top` from bounded one (e.g., `value_type`, `size_type`) to integer. A combination of bounded type for a logic function with an arithmetic operations in the logical definitions may lead to inconsistency. This fixes the inconsistencies in the `accumulate`, `stack` and `stack_wd` examples.
  - Fix an inconsistency in `Difference_Read` axiom: restriction on the array size added to premises.
- Various improvements
  - An important change is the rewriting of the implicit, *axiomatic* definitions of `Accumulate`, `Count`, `Difference`, `InnerProduct` and `UniqueSize` logic functions to explicit, *recursive* ones. Accordingly, all axioms in the respective examples have been rewritten as lemmas.
  - Generalize `CountSectionMonotonic`, `UnchangedSection` lemmas: remove restriction on lower bound for the range.
  - Fix typo in postcondition of `find`.
  - Rewrite specifications of `remove_copy` and `remove` examples.
  - Rename predicate `RemoveCount` to `RemoveSize`.

    - Gather all versions of `MultisetRetainRest` in section on `push_heap`.
    - Add another figure to highlight simple contract for `unique_copy`.
    - Adapt Coq proofs to the fact that the `Z` scope is not available by default.
- New examples
    - Add `count2` example with an inductive predicate instead of a logic function in `count`.
    - Add `merge` example.
- Infrastructure
    - Travis-CI configuration for the GitHub repository added as an illustrative example of how the verification results could be reproduced.
    - Add support for Frama-C/AstraVer plugin.

B.4. New in Version 18.0.0 (Argon, December 2018)

- Replace the links to the (now abandoned) original site of *Standard Template Library* (STL) by references to the C++ standard.
- Add new algorithm `unique_copy` (two versions).
- Add another assertion `half` for `reverse`.
- Add two overloaded versions of predicate `ConstantRange` and use them for the algorithms `fill` and `unique_copy`, respectively.

B.5. New in Version 17.1.0 (Chlorine, July 2018)

The exact version number of Frama-C originally was Chlorine-20180502. This version number was changed in October 2018 to 17.1

- Slightly change the definition of predicate `HasEqualNeighbors` and its use in the specification of `adjacent_find`.
- Remove the algorithm `remove` and the more elaborate version of `remove_copy`. We are currently working on new specifications of these algorithms.
- Adapt some Coq proofs related to the logic function `Count` in order to reflect changes in output of Frama-C/WP.
- Remove table on ACSL lemmas that had to be proved by Coq.

B.6. New in Version 16.1.1 (Sulfur, March 2018)

- fix several errors reported by Aaron Rocha, including,
    - fix an error in figure for `upper_bound` algorithms
- fix merging of contracts in second version of `binary_search`

- improve and justify the `retain` annotations of in the implementation of `remove`
- Alt-Ergo is now directly called in the parallel setting (instead of going through Why3) to be compatible with the sequential setting
- add a third assertion `reorder` in the `random_shuffle` body to keep verification rate at 100% after prover upgrade

B.7. New in Version 16.1.0 (Sulfur, December 2017)

- special thanks to Aaron Rocha who provided various improvements for Chapters 4, 5, and 6
- improve some mutating algorithms
  - add more assertions to `reverse` to reduce reliance on CVC3
  - improve structure and ACSL annotations of `remove_copy` and `remove`
    - add overloaded version of predicate `MultisetRetainRest`
    - add lemma `HasValueImpliesPositiveCount`
    - add lemma `PositiveCountImpliesHasValue`
    - remove lemma `HasValueShiftInversion`
    - remove lemma `HasValueCountInversion`
  - add custom lemma `random_number_modulo` for `random_shuffle`
- add new Chapter 11 with more algorithms related to sorting
  - add algorithm `is_sorted` including predicate `WeaklyIncreasing`
    - add lemma `IncreasingImpliesWeaklyIncreasing`
    - add lemma `WeaklyIncreasingImpliesIncreasing`
  - add algorithm `partial_sort` including predicate `Partition`
    - add lemma `ReorderImpliesMatch`
    - add lemma `ReorderPreservesUpperBound`
    - add lemma `ReorderPreservesLowerBound`
    - add lemma `PartialReorderPreservesLowerBounds`
    - add lemma `SwappedInside`
    - add lemma `SwappedInsideMultisetUnchanged`
    - add lemma `SwappedInsidePreservesMultisetUnchanged`
- improve various lemmas
  - rename lemma `SortedUp` to `IncreasingUpperBound`
  - generalize lemma `UnchangedSection`
  - refactor lemma `HeapBounds` into `C_Division_Two`

B.8. New in Version 15.1.2 (Phosphorus, October 2017)

- fix several typos reported by `seniorlackey@github` (thanks a lot!)
- add a new chapter on classic sorting algorithms which comprises
  - `selection_sort` including lemma `SwapImpliesMultisetUnchanged`
  - `insertion_sort` including lemmas
    - `RotatePreservesStrictLowerBound`
    - `RotateImpliesMultisetUnchanged`
    - `EqualRangesPreservesIncreasing`
    - `EqualRangesPreservesCount`
  - `heap_sort`
- heap algorithms
  - remove length requirements in `pop_heap`, `sort_heap`, `make_heap`, and `heap_sort`
    - introduce `SIZE_TYPE_MAX` to catch border cases in ACSL and C
  - improve description of `pop_heap`
    - add predicate `HeapChildMax`
    - provide the auxiliary function `heap_child_max`
    - the postcondition `reorder` is still not verified
  - improve description of `push_heap`
  - other, minor improvements
    - add auxiliary function `heap_parent`
    - add predicate `SortedDown` and lemma `SortedDownIsHeap`
    - add lemmas `HeapParentChild` and `HeapChilds`
    - add lemmas `HeapParentBounds` and `HeapChildBounds`

B.9. New in Version 15.1.1 (Phosphorus, September 2017)

- add ensures clause to default behavior of the following algorithms
  - `find`, `find_first_of`, `adjacent_find`, `mismatch`, `search`, `search_n`, `find_end`
  - `max_element`, `min_element`
- rewrite axiomatic definitions to ensure disjoint guards which is better suited for E-ACSL
  - concerns the axiomatic definitions of `Count`, `Accumulate`, `InnerProduct` and `Difference`
  - some Coq proofs related to `Count` had to be adapted as well

- shorten names of some auxiliary algorithms
  - `adjacent_difference_inverse` ↦ `adjacent_difference_inv`
  - `partial_sum_inverse` ↦ `partial_sum_inv`
- heap algorithms
  - fix a typo in Figure 10.4
  - fix a typo in Figure 10.33
  - explain that there can be multiple representations of an array as a heap
  - add a version of `pop_heap` that is, however, not completely verified

B.10. New in Version 15.1.0 (Phosphorus, June 2017)

- The verification results are now part of the appendix.
- Fix an error in the specification of the well-definition of `stack_size`.
- This release of Frama-C/WP could not discharge some of our assertions of `push_heap`. We therefore have completely rewritten the annotations and also tweaked the implementation of `push_heap`. We also added some new predicates and lemmas to maintain a concise specification that can easily be verified by automatic provers.
  - add predicate `MultisetAdd` and lemma `MultisetAddDistinct`
  - add predicate `MultisetMinus` and lemma `MultisetMinusDistinct`
  - add predicate `MultisetRetain` and lemma `MultisetPushHeapRetain`
  - provide an additional version of predicate `MultisetRetainRest`
  - and lemma `MultisetPushHeapClosure`

B.11. New in Version 14.1.1 (Silicon, April 2017)

- changes in verification infrastructure
  - add verification results for the case where each proof obligation is submitted to all automatic theorem provers
- changes in algorithms
  - simplify loop invariants of `search_n` and improve description
  - rename predicate `CountOneHit` to `CountHit`
  - rename predicate `CountOneMiss` to `CountMiss`
  - rewrite predicates `EqualRanges` and `Reverse` in order to simplify the task for automatic theorem provers
  - remove lemmas on `Reverse` that were necessary for `rotate` but are not needed anymore
  - rename predicate `Valid(Stack*)` to `Invariant(Stack*)` and
    remove `\valid` from `Invariant(Stack*)`

  - add a simple random number generator to random_shuffle and verify it
- fix an inconsistency in the axioms for Count (thanks to Denis Efremov for reporting this issue)
  - add more guards to axioms CountSectionHit and CountSectionMiss
  - add corresponding guards to lemmas
    - CountSectionOne, CountHit, CountMiss and CountOne
    - RemoveCountHit and RemoveCountMiss
  - add lemma Unchanged_Shift and add more assertions to remove in order to simplify the task for automatic theorem provers

B.12. New in Version 14.1.0 (Silicon, January 2017)

- use label Old instead of Pre in function contracts
- add algorithm rotate
- rewrite definition of predicates EqualRanges and Reverse and provide more overloaded versions
- add figures for algorithms rotate and replace_copy
- update figure for predicate Reverse
- update Coq proofs and add a table with more information on the ACSL lemmas that had to be verified with Coq

B.13. New in Version 13.1.1 (Aluminium, 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 Count_Shift
- 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`

B.14. 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 10. This new description of heap algorithms is based to a large extend on the bachelor thesis of one of the authors [28].

- 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 `Count_Bounds` 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`
    - 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`

B.15. 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 Frama-C/WP Magnesium and Sodium where prover processes are not properly terminated.[50] 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
    - 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`

[50]See `https://bts.frama-c.com/view.php?id=2154`

        * 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 StackPush_Equal to enable the automatic verification of the well-definition of stack_push

B.16. New in Version 11.1.1 (Sodium, June 2015)

- add Chapter on numeric algorithms
    - move iota algorithm to numeric algorithms (Section 9.1)
    - add accumulate algorithm (Section 9.2)
    - add inner_product algorithm (Section 9.3)
    - add partial_sum algorithm (Section 9.4)
    - add adjacent_difference algorithm (Section 9.5)

B.17. New in Version 11.1.0 (Sodium, March 2015)

- Use built-in predicates \valid and \valid_read instead of valid_range.
- Simplify loop invariants of find_first_of.
- Replace two loop invariants of remove_copy by ACSL lemmas.
- Rename several predicates
    - IsEqual $\mapsto$ 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.

B.18. 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

B.19. 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

B.20. 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`

B.21. 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 [31] and now has a much better verification rate.

B.22. 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

B.23. 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 Frama-C/WP with Alt-Ergo
- replaced deprecated \valid_range by \valid
- 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

B.24. New in Version 7.1.0 (Nitrogen, December 2011)

- changed to Frama-C Nitrogen
- changed to Why 2.30
- discussed both plug-ins Frama-C/WP and Jessie
- removed swap_values algorithm

B.25. New in Version 6.1.0 (Carbon, not published)

- changed definition of stack
- renamed reset_stack to init_stack

B.26. New in Version 5.1.1 (Boron, February 2011)

- prepared algorithms for checking by the new Frama-C/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`

B.27. 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`*`)`

B.28. 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.

B.29. 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`

B.30. New in Version 4.2.0 (Beryllium, January 2010)

- complete rewrite of pre-release
- adaption to Frama-C Beryllium 2 release

Bibliography

[1] WP Plug-in. `http://frama-c.com/wp.html`.

[2] Frama-C Software Analyzers. `http://frama-c.com`, 2018.

[3] Why – Where Programs Meet Provers. `http://why3.lri.fr`, 2018.

[4] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. The Alt-Ergo SMT Solver. `http://alt-ergo.lri.fr`, 2018.

[5] Clark Barrett and Cesare Tinelli. Homepage of CVC4. `http://cvc4.cs.stanford.edu/web/`, 2018.

[6] Clark Barrett and Cesare Tinelli. Homepage of CVC3. `http://www.cs.nyu.edu/acsys/cvc3/`, 2010.

[7] Microsoft Research. The Z3 Theorem Prover. `https://github.com/Z3Prover/z3`, 2018.

[8] The Coq Consortium. The Coq Proof Assistant. `https://coq.inria.fr`, 2018.

[9] ANSI/ISO C Specification Language. `http://frama-c.com/acsl.html`, 2018.

[10] CEA LIST, Laboratory of Applied Research on Software-Intensive Technologies. `http://www-list.cea.fr/gb/index_gb.htm`.

[11] INRIA-Saclay, French National Institute for Research in Computer Science and Control . `http://www.inria.fr/saclay/`.

[12] LRI, Laboratory for Computer Science at Université Paris-Sud. `http://www.lri.fr/`.

[13] Fraunhofer-Institut für Offene Kommunikationssysteme (FOKUS). `http://www.fokus.fraunhofer.de`.

[14] Virgile Prevosto. ACSL Mini-Tutorial. `http://frama-c.com/download/acsl-tutorial.pdf`.

[15] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL 1.13 Implementation in Argon 18.0. `https://frama-c.com/download/acsl-implementation-18.0-Argon.pdf`, 2018.

[16] Allan Blanchard. Introduction to C Program Proof using Frama-C and its wp plugin. `http://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf`, December 2017.

[17] Programming languages – C, Committee Draft. `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1362.pdf`, 2009.

[18] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[19] 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.

[20] Richard Smith. Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf, 2017. publicly available draft of C++ 17 standard.

[21] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J Comput*, 6(2):323–350, Jun 1977.

[22] Lincoln E. Moses and Robert V. Oakford. *Tables of Randon Permutations*. Stanford University Press, 1963.

[23] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Graduate texts in computer science. Springer, New York, 1997.

[24] Repository of "libc++" C++ Standard Library. https://llvm.org/svn/llvm-project/libcxx/trunk, 2018. Revision 345375: /libcxx/trunk.

[25] Patrick Baudin, François Bobot, Loïc Correnson, and Zaynah Dargaye. WP Plug-in Manual — Frama-C Argon 18.0. http://frama-c.com/download/frama-c-wp-manual.pdf, 2018.

[26] Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. Frama-C User Manual, Release Argon 18.0. https://frama-c.com/download/user-manual-18.0-Argon.pdf.

[27] Philippe Herrmann and Julien Signoles. Frama-C's annotation generator plug-in for Frama-C Argon 18.0. https://frama-c.com/download/rte-manual-18.0-Argon.pdf.

[28] Timon Lapawczyk. Formale Verifikation von Heap-Algorithmen mit Frama-C. bachelor thesis, Humboldt-Universität zu Berlin, July 2016.

[29] D.E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[30] Sellibitze. How to Implement Classic Sorting Algorithms in Modern C++. https://stackoverflow.com/questions/24650626/how-to-implement-classic-sorting-algorithms-in-modern-c, Aug 2014.

[31] Kim Völlinger. Einsatz des Beweisassistenten Coq zur deduktiven Programmverifikation. Diplomarbeit, Humboldt-Universität zu Berlin, August 2013. https://www2.informatik.hu-berlin.de/top/_media/www/mitarbeiter/diplomarbeit-kim-voellinger.pdf.

[32] Richard Fitzpatrick J.L. Heiberg. *Euclid's Elements of Geometry*. http://farside.ph.utexas.edu/euclid.html, Austin/TX, 2008.

[33] David Hilbert. *Grundlagen der Geometrie*. B.G.Teubner, Stuttgart, 1968.

[34] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.

[35] Stephan Schulz. The E Theorem Prover. https://wwwlehre.dhbw-stuttgart.de/~sschulz/E/E.html, 2018.