

Introduction to Deductive Program Verification with Frama-C

Nikolaus Huber

February 2023

Contents

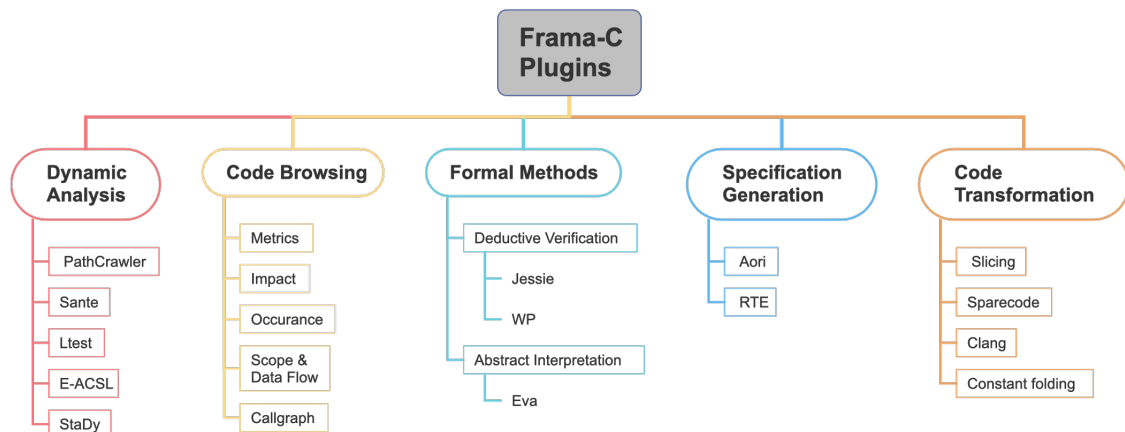
Contents	1
1 Introduction	2
2 Basics	4
2.1 A first example	4
2.2 Preconditions	6
2.3 Partial vs complete specification	6
2.4 Exercises	9
3 Pointers	11
3.1 Exercises	13
4 Loops	15
4.1 A first example	15
4.2 Loop invariants	17
4.3 Loop assignments	17
4.4 Loop variants	19
4.5 Exercises	20
5 Behaviors	22
5.1 Exercises	23
Bibliography	25

Chapter 1

Introduction

In this tutorial we will see some basic concepts of deductive program verification of C programs. We will use a tool called Frama-C¹ (FRAmework for Modular Analysis of C code), and its specification language ACSL (ANSI C Specification Language). ACSL annotations can however also be used by tools other than Frama-C, such as the model checker TriCera [4] developed at Uppsala University.

Frama-C is actually not a single tool, but a collection of different code analysis tools. It provides a common interface for different *plugins* performing different analyses on the code we provide. Here is an overview of the plugins available in the standard Frama-C distribution:



In this tutorial we will only use the WP plugin, which performs deductive verification on our C code. WP stands for *weakest precondition*, a theory we have seen during the verification lecture that builds the basis of this plugin. We will however not look deeper into the theory, and only see how we can use Frama-C in practice for proving properties of our programs.

This introduction barely scratches the surface of what is possible with Frama-C. It shall just give you an idea of the power of static program verification, and hopefully inspire you to look deeper on your own.

While this text is mostly self-contained, I would like to mention a few resources that can come in handy

¹<https://www.frama-c.com>

if you should ever find yourself stuck, or you would like to gain some deeper insight into some of the provided functionality.

The first important resource is the manual of Frama-C itself [3]. ACSL has its own separate manual [1]. A slightly more complex, but also more thorough introduction into Frama-C and ACSL can be found in [6].

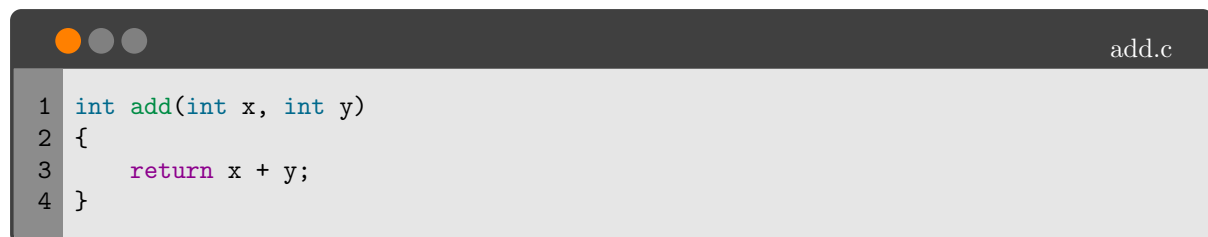
In [2] the author gives a tutorial that covers almost everything ACSL offers. It is very approachable, and might be a good starting point after working through this tutorial. Last but not least, [5] tries to teach ACSL through examples by going through the verification of many algorithms that are part of the C++ standard library.

Chapter 2

Basics

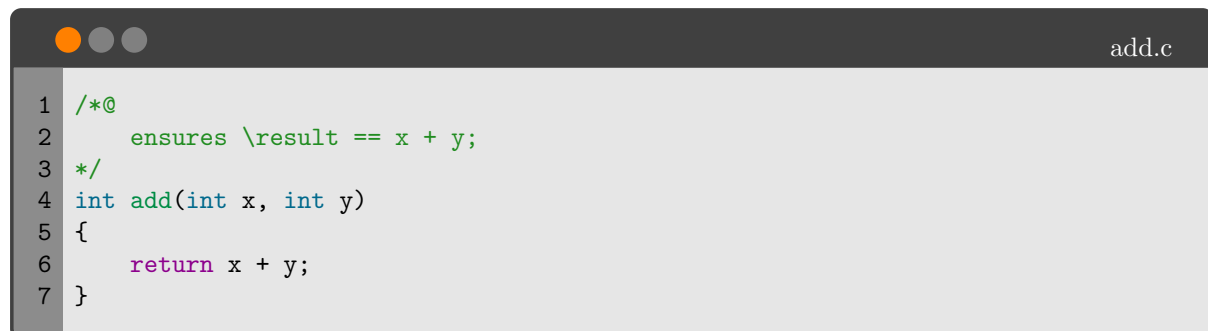
2.1 A first example

Let's start with an easy example. Assume we want to define a function which can add two integers:

A screenshot of a code editor window titled 'add.c'. The editor shows a C function 'add' with two integer parameters. The code is as follows:

```
1 int add(int x, int y)
2 {
3     return x + y;
4 }
```

If we want to prove, that this function is correct, we can define a *function contract*, which specifies the behavior we expect:

A screenshot of a code editor window titled 'add.c'. The editor shows a C function 'add' with two integer parameters. Above the function signature, there is an ACSL contract. The code is as follows:

```
1 /*@
2     ensures \result == x + y;
3 */
4 int add(int x, int y)
5 {
6     return x + y;
7 }
```

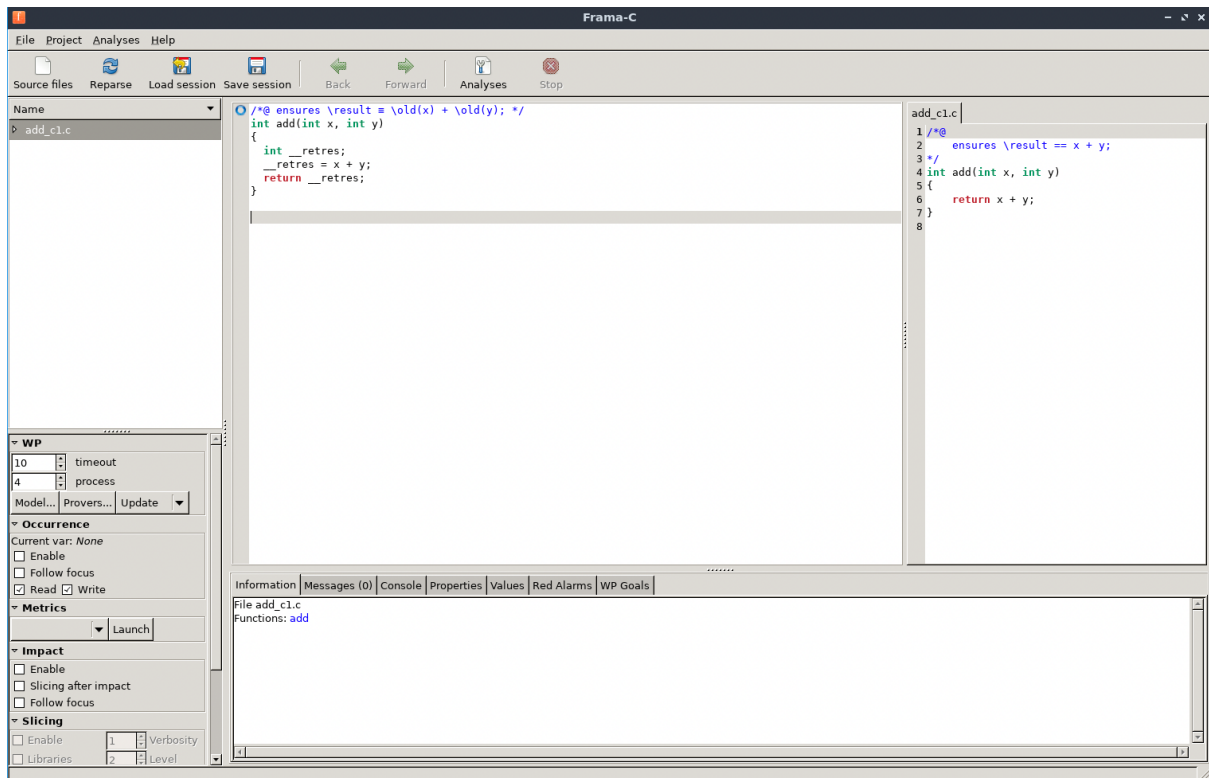
As you can see, function contracts are written as a special comment section (`/*@ ... */`) just above the function signature. The *ensures* keyword introduces a *postcondition*, i.e. a formula which must be true when the function returns. ACSL offers the keyword `\result` as a placeholder for the return value of a function.

We can try proving this contract by invoking Frama-C on our C file:

A screenshot of a terminal window. The prompt is 'user@linux:~#'. The command entered is 'frama-c-gui add.c'.

```
user@linux:~# frama-c-gui add.c
```

This will start the graphical user interface of Frama-C:



There are a few important parts in the Frama-C GUI. On the left side we have an overview of open files and functions defined therein. On the right side we can see our original function and our contract. In between we can see the output of the parser of Frama-C, for the most part we can just accept it as a slightly altered version of the code we provided, which has exactly the same meaning as the one we wrote.

If you look carefully, the function contract has been slightly pretty printed in the parsed view (e.g. the `==` has been changed to `≡`). This will happen with many of the mathematical definitions we are going to use later on as well.

Probably the most interesting part for us is the little blue circle next to our contract in the middle window. This indicates, that we have not yet proven the contract. So let's start with that.

When we want to prove function contracts we can do so in multiple ways. In the middle window right-click on the function name (i.e. **add**) and select "Prove function annotations by WP". This will turn the blue circle into a green bullet, which indicates that Frama-C was successful in proving our contract.

The careful reader might object at this point, and say something intelligent along the lines of "but what about overflows?". And indeed, those might very well happen if we add up two integers that are big enough. When Frama-C proved our contract it did so under the assumption that no runtime error occurred. However, as mere mortal programmers, we do have to take into account the finite machine precision of our hardware.

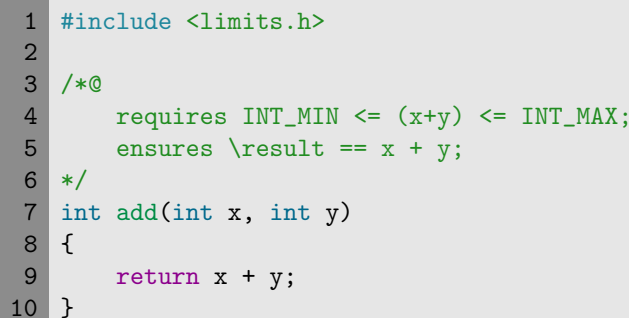
Luckily, Frama-C can help us with that. It provides a plugin called WP-RTE (Run Time Errors), which can automatically insert assertions into our code to check for runtime errors, such as overflows. To do so, we right-click on our function name again and select "Insert wp-rte guards". If you don't see this option, then the RTE plugin has not been activated yet. In that case, click on "Analyses" in the header menu (the one that looks like a sheet of paper and a tool in front of it). Then select "WP" in the list on the left, unfold "Computational Strategies" and select "-wp-rte".

Now we can try and prove our function annotations again. You will see, that our own contract still turns green, however, the automatically inserted RTE guards are yellow, which means that Frama-C could not prove those for all possible inputs to our function.

2.2 Preconditions

To allow Frama-C to prove the absence of runtime errors, we have to give certain guarantees that are valid when we call our **add** function. These guarantees are called *preconditions*. Pre- and postconditions are the bread and butter of functional contracts. We could formulate the process of what Frama-C is doing as the following: If we can guarantee, that all preconditions are valid when we call our function, and the function itself terminates, then Frama-C will prove that the given postcondition(s) are also satisfied.

In our case we can state the following:



```
1 #include <limits.h>
2
3 /*@
4   requires INT_MIN <= (x+y) <= INT_MAX;
5   ensures \result == x + y;
6 */
7 int add(int x, int y)
8 {
9     return x + y;
10 }
```

Preconditions are introduced by the *requires* keyword. As you can see, ACSL allows for a nicer syntax when it comes to defining conditionals, as we can define the lower and upper limit of $x + y$ in one formula instead of two (i.e. `INT_MIN <= (x + y) && (x + y) <= INT_MAX`). The macros `INT_MIN` and `INT_MAX` can be provided using the `limits` library.

You cannot edit your code in Frama-C directly, so you have to do your changes in a separate editor and then reparse the file. You can also just close the current Frama-C window, change the source code and then open it again. We can tell Frama-C to immediately insert the RTE guards and perform the contract analysis:

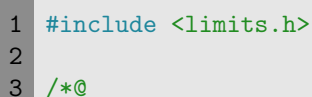


```
user@linux:~$ frama-c-gui -wp -wp-rte add.c
```

If we check the status of our function annotations, all of them should be green now. The only blue circle that remains is our precondition, since that can of course not be proven without actually calling the function.

2.3 Partial vs complete specification

At this point we might assume that we were successful in specifying the behavior and proving the correctness of our implementation of the **add** function. Unfortunately we are not finished yet. To see that, let us observe what happens if we actually try to call our **add** function from another function:



```
1 #include <limits.h>
2
3 /*@
```

```

4     requires INT_MIN <= (x+y) <= INT_MAX;
5     ensures \result == x + y;
6 */
7 int add(int x, int y)
8 {
9     return x + y;
10 }
11
12 int v;
13
14 void main(void)
15 {
16     v = 10;
17     int r = add(3,4);
18     //@ assert r == 7;
19     //@ assert v == 10;
20 }

```

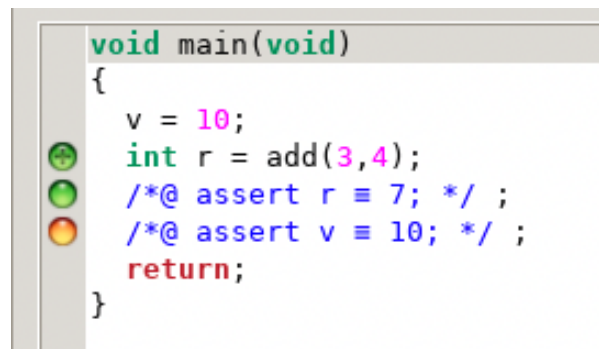
As you can see, we can also write ACSL annotations with single line comments. The *assert* keyword allows us to insert assertions about our program variables at any point in our code. Here we also included a global variable *v*, which after calling **add** should still have the same value it had before. When we try and run

```

user@linux:~$ frama-c-gui -wp -wp-rte add.c

```

now, we will get the following for the main function:



```

void main(void)
{
    v = 10;
    int r = add(3,4);
    /*@ assert r == 7; */ ;
    /*@ assert v == 10; */ ;
    return;
}

```

The green bullet next to the **add** function tells us that we are fulfilling the preconditions of the **add** function, and the green bullet on the next line shows that the return value is indeed 7. However, Frama-C failed to prove that the global variable *v* has not changed value.

This is due to the fact, that we have not specified anything about the memory locations that our **add** function is allowed to alter. Therefore, Frama-C cannot simply assume that no global variables have been changed once we call **add**. We have to list all the locations that **add** can read/write to in the contract of our function, which we can do by using the *assigns* keyword. With that in place, our file looks as follows:


```
add.c
1 #include <limits.h>
2
3 /*@
4   requires INT_MIN <= (x+y) <= INT_MAX;
5   ensures \result == x + y;
6   assigns \nothing;
7 */
8 int add(int x, int y)
9 {
10     return x + y;
11 }
12
13 int v;
14
15 void main(void)
16 {
17     v = 10;
18     int r = add(3,4);
19     //@ assert r == 7;
20     //@ assert v == 10;
21 }
```

ACSL provides the special value `\nothing` to express that the function is not changing any global state. If we run the same command as before, we will now see that also the second assert statement turns green:

```
void main(void)
{
    v = 10;
    int r = add(3,4);
    /*@ assert r == 7; */ ;
    /*@ assert v == 10; */ ;
    return;
}
```

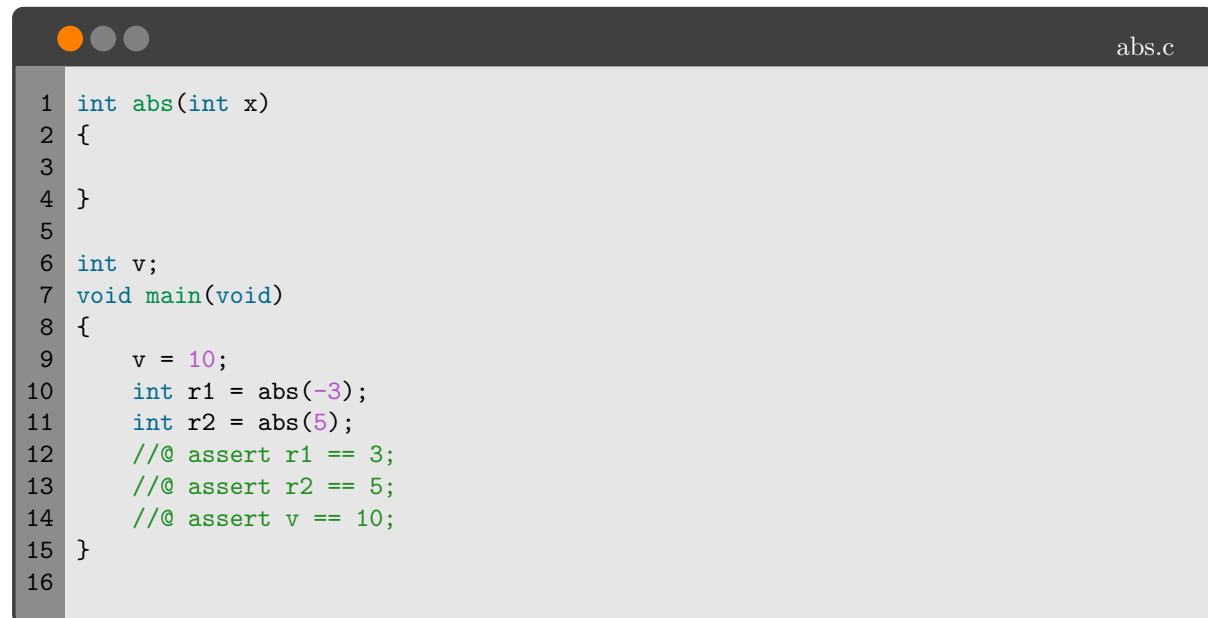
This helps illustrate an important fact about using contracts for function specification. It can be difficult to formulate a complete specification (that is, one that covers every aspect of the behavior we are expecting). Depending on the calling contexts of the function we are trying to specify, a partial specification might be sufficient to prove correctness of our program. It can also happen that a contract that fully specifies our implementation is outside the scope of what ACSL can do. Such cases, we have to evaluate against a more abstract model of our software system.

So far we have only dealt with contracts that had one pre- and one postcondition. We can however specify arbitrarily many by either having multiple requires/ensures clauses, or using the logical operators `&&` and `||` respectively. We will make use of that in the next chapter, and you might need it in the exercises.

2.4 Exercises

2.4.1 Abs function

Implement the **abs** function, which shall return the absolute value of the integer parameter *x* given to it:

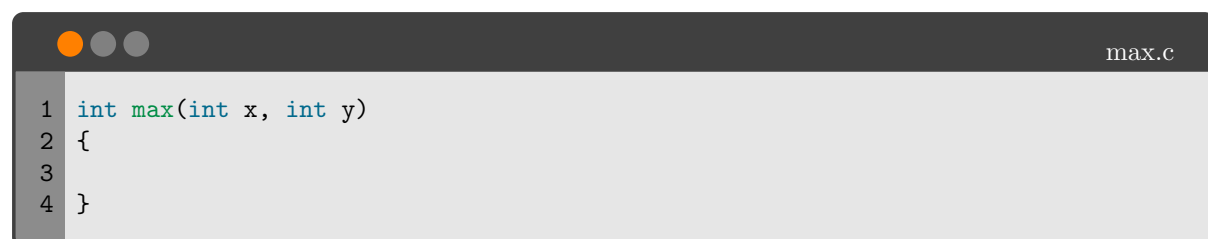


```
1 int abs(int x)
2 {
3
4 }
5
6 int v;
7 void main(void)
8 {
9     v = 10;
10    int r1 = abs(-3);
11    int r2 = abs(5);
12    //@ assert r1 == 3;
13    //@ assert r2 == 5;
14    //@ assert v == 10;
15 }
16
```

Write the necessary specification for the **abs** function and prove its correctness. Prove the assertions in **main** and the automatically generated run-time exceptions within the **abs** function. For specifying the postcondition, you may want to use the \Rightarrow (implies) operator. For further details consult [1].

2.4.2 Max function

Implement the **max** function, which shall return the bigger of the two input parameters:

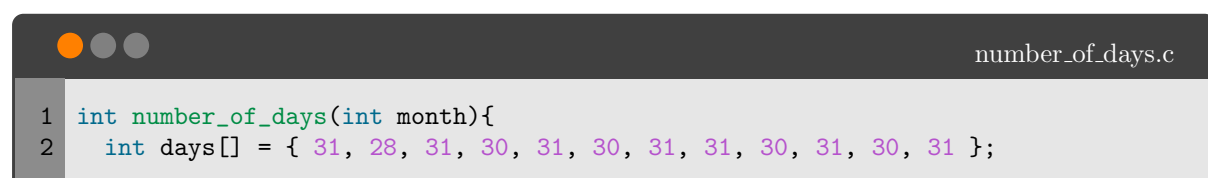


```
1 int max(int x, int y)
2 {
3
4 }
```

This time, write your own main function and try to include and prove some assertions!

2.4.3 Days per month

Assume we have the following given function, which given a month (i.e. an integer between 1 and 12) returns the number of days in that month:



```
1 int number_of_days(int month){
2     int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
3 }
```

```
3   return days[month-1];  
4 }
```

Write the required pre- and postconditions for this function. You will realize, that the postcondition is quite awkward to write, luckily ACSL can help us here. ACSL provides the `\in` keyword, that allows to reason about sets like this: `//@ assert 3 \in {1, 2, 3, 4};`. Try using it in your postcondition!

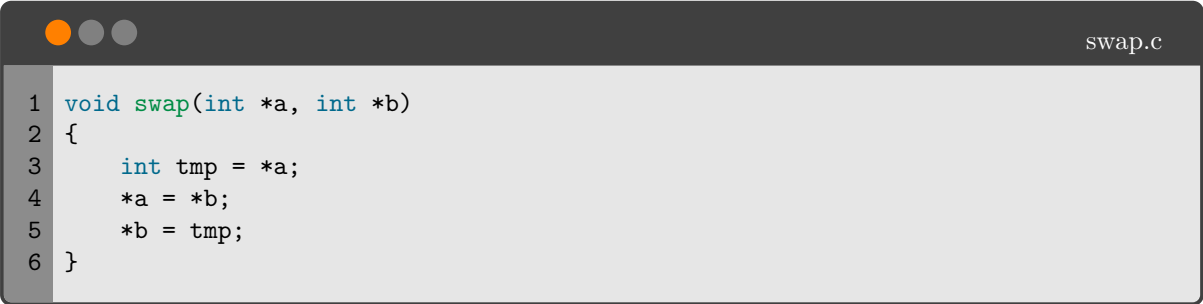
Chapter 3

Pointers

As you know, pointers are one of the main features of the C programming languages, and ultimately one that makes it such a popular language for low-level programming, since pointers allow us to access specific memory regions directly. They are, however, also one of the most common sources of bugs and errors in C programs, and can easily lead to the failure of a complete system.

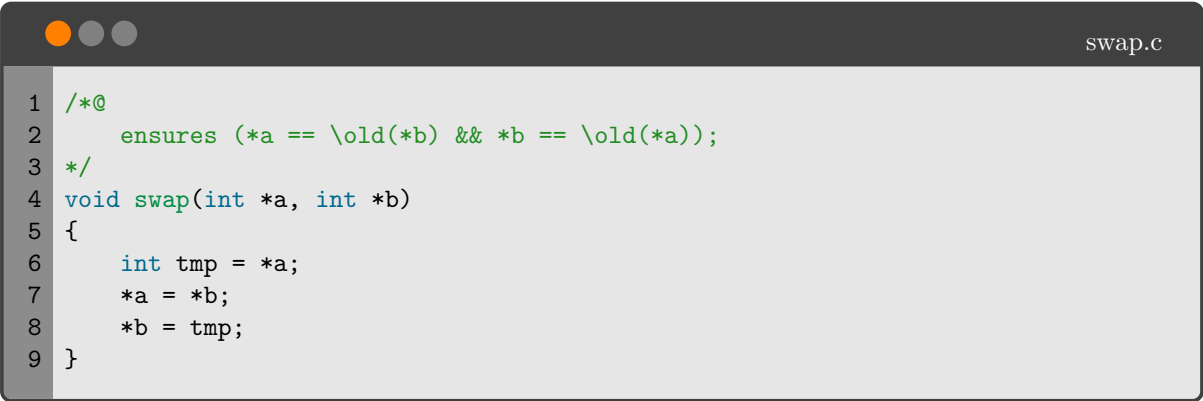
Therefore, ACSL has some special constructs we can use for expressing specifications that include pointers.

Let's have a look at a simple example again. Assume we want to write a function **swap**, which takes two pointers to integer variables as input and swaps the values that are saved at these locations:



```
1 void swap(int *a, int *b)
2 {
3     int tmp = *a;
4     *a = *b;
5     *b = tmp;
6 }
```

So let's think for a moment what kind of contract we need for such a function. We already know from the last chapter that we need to specify at least a postcondition we want to prove. In case of the **swap** function that might look something like this:



```
1 /*@
2     ensures (*a == \old(*b) && *b == \old(*a));
3 */
4 void swap(int *a, int *b)
5 {
6     int tmp = *a;
7     *a = *b;
8     *b = tmp;
9 }
```

This introduces a new ACSL keyword that is very important. We can use **\old** in a postcondition to refer to the value of a variable at the beginning of the function. That is an important concept, since the value of a variable (especially when dealing with pointers) might change during the execution of the function we are trying to specify.

We can also see in this example, that we can combine multiple postconditions by using `&&` (similarly you can use `||` if you need it). Instead of using a logical AND, we could have also just written two separate ensures clauses. Both of them would work.

Perhaps surprisingly, trying to use Frama-C on our contract yields a negative result:

```

/*@ ensures *\old(a) == \old(*b) ^ *\old(b) == \old(*a); */
void swap(int *a, int *b)
{
    /*@ assert rte: mem_access: \valid_read(a); */
    int tmp = *a;
    /*@ assert rte: mem_access: \valid(a); */
    /*@ assert rte: mem_access: \valid_read(b); */
    *a = *b;
    /*@ assert rte: mem_access: \valid(b); */
    *b = tmp;
    return;
}

```

The guards that the RTE plugin inserted give us a hint as to why Frama-C could not prove our contract. We never said anything about the pointers `*a` and `*b`. What would happen if we did something like `swap(NULL, NULL)`?

So in order for Frama-C to know that we are not calling `swap` with any invalid pointer, we need to have a precondition:

```

swap.c
1  /*@
2     requires \valid(a) && \valid(b);
3     ensures (*a == \old(*b) && *b == \old(*a));
4  */
5  void swap(int *a, int *b)
6  {
7     int tmp = *a;
8     *a = *b;
9     *b = tmp;
10 }

```

The ACSL keyword `\valid` allows us to specify a guarantee that the provided pointer is valid. There is also a `\valid_read` which specifies that we can read from the given pointer, but we cannot write to it. That might be interesting for example when you want to read out certain configuration registers that are actually hardwired or can only be set during the boot up process.

So with our `\valid` precondition in place, we can finally prove the correctness of `swap`:

```

/*@ requires \valid(a) ^ \valid(b);
   ensures *\old(a) == \old(*b) ^ *\old(b) == \old(*a);
*/
void swap(int *a, int *b)
{
    /*@ assert rte: mem_access: \valid_read(a); */
    int tmp = *a;
    /*@ assert rte: mem_access: \valid(a); */
    /*@ assert rte: mem_access: \valid_read(b); */
    *a = *b;
    /*@ assert rte: mem_access: \valid(b); */
    *b = tmp;
    return;
}

```

But is this specification actually complete? If we would try a similar main function as we did in the previous chapter, we would see that Frama-C cannot prove that our `swap` function does not alter any

global memory besides the two provided pointers. We again have to specify concretely, which locations our function can alter with the ACSL assigns statement:

```
1 /*@
2   requires \valid(a) && \valid(b);
3   assigns *a, *b;
4   ensures (*a == \old(*b) && *b == \old(*a));
5 */
6 void swap(int *a, int *b)
7 {
8     int tmp = *a;
9     *a = *b;
10    *b = tmp;
11 }
```

As we will see later, the `\valid` and `\valid_read` functions provided by ACSL are not only important when dealing with simple pointers. We will encounter them again when we talk about proving code that deals with arrays, since an array can also be seen as a collection of pointers to its elements.

3.1 Exercises

3.1.1 Max-pointer function

Suppose we want to implement a function `max_ptr` which takes two pointers to integer values as arguments and puts the bigger value at the first location and the smaller one at the second:

```
1 void max_ptr(int* x, int* y)
2 {
3
4 }
5
6 int v = 5;
7 void main(void)
8 {
9     int x = 10;
10    int y = 7;
11    max_ptr(&x, &y);
12    //@ assert x == 10;
13    //@ assert y == 7;
14    //@ assert v == 5;
15
16    x = 8;
17    y = 15;
18    max_ptr(&x, &y);
19    //@ assert x == 15;
20    //@ assert y == 8;
21    //@ assert v == 5;
22 }
```

Write the required contract so that Frama-C can prove all assertions in the main function!

3.1.2 Decrementing integer by another integer

Have a look at the following implementation of a decrement function:

```
decr_by.c
1 void decr_by(int* x, int const* y)
2 {
3     *x = *x - *y;
4 }
5
6 void main(void)
7 {
8     int x = 7;
9     const int y = 3;
10    decr_by(&x, &y);
11    //@ assert x == 4;
12    //@ assert y == 3;
13 }
```

We provide two pointers to integers as input and the function **decr_by** shall subtract the value the second pointer points to from the first one. Try writing a contract for this function. You will realize, that Frama-C actually fails to prove your postconditions. This is due to the possibility of aliasing. Two pointers can actually point to the same memory location. But what would happen if we did something like `int a = 5; decr_by(&a, &a);`? We have to tell Frama-C in a precondition that the two pointers are actually different, which we can do like this: `requires \separated(x, y);`. Try proving your function contract again, now it should be able to prove it correct.

3.1.3 Adding of pointed values

In this exercise we want to create a function that takes two integer pointers as input and returns the sum of the values they point to:

```
add_ptr.c
1 int add_ptr(int* a, int* b)
2 {
3
4 }
5
6 void main(void)
7 {
8     int a = 13;
9     int b = 15;
10    int r = add_ptr(&a, &b);
11    //@ assert r == 28;
12 }
```

Implement the function **add_ptr** and write the required pre- and postconditions to prove the assertion in the main function and the automatically generated run-time exceptions within **add_ptr**! Do we need to use `\separated` here?

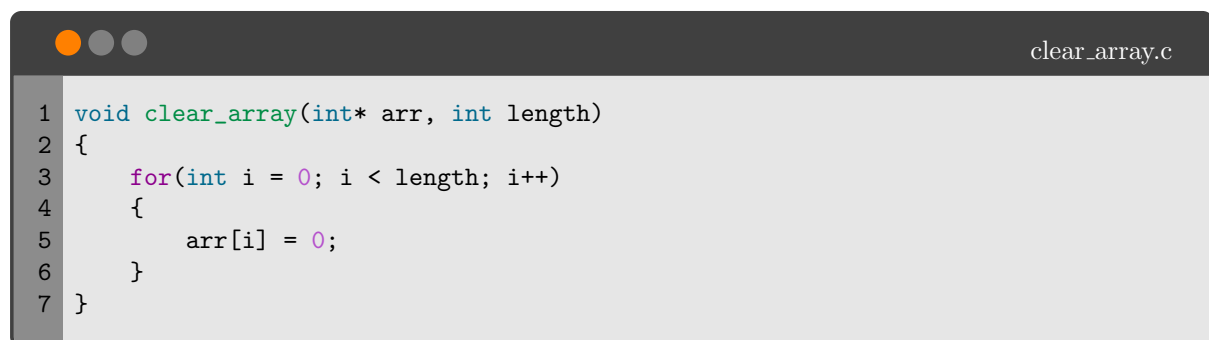
Chapter 4

Loops

By now we can actually already specify non-trivial programs with ACSL and have Frama-C try and prove that our implementation is according to the specification. Something we are missing so far is any form of repeated code application. We therefore take a look at loops next. We will also take a look at one of the most important data structures in C, arrays. You will realize that we actually already know quite a bit about how to write contracts including arrays, since in C arrays are just lists of pointers!

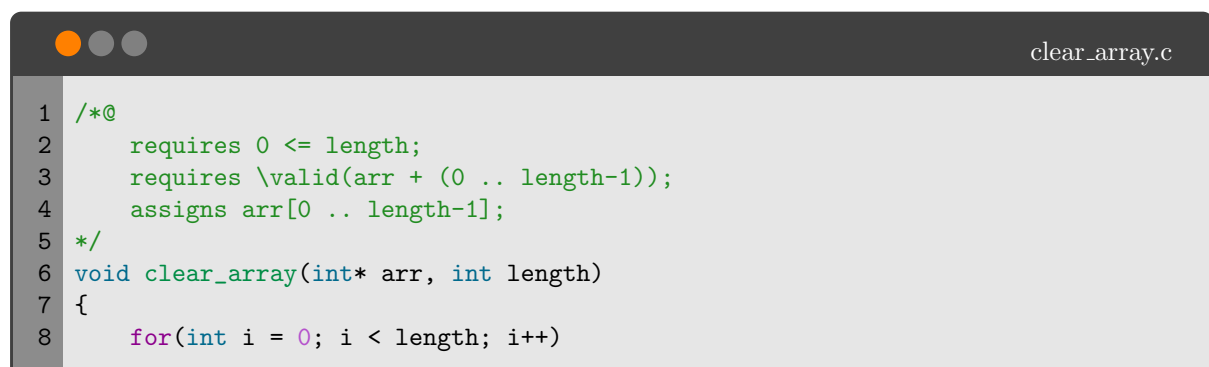
4.1 A first example

We'll start by using a simple example again that we will try and refine throughout this chapter! Assume that we need to implement a function that takes as arguments a pointer to the first element of an array and the length of the array, and assigns the value 0 to all array positions:



```
1 void clear_array(int* arr, int length)
2 {
3     for(int i = 0; i < length; i++)
4     {
5         arr[i] = 0;
6     }
7 }
```

Let's start with the things we already know how to specify. We know that we have two preconditions, one about the length of the array (i.e. it cannot be negative), and one about the validity of the array itself. When we call the function, we need to guarantee that `arr[0]` to `arr[length-1]` are actually valid pointers. Luckily, ACSL provides a short and concise syntax for that:



```
1 /*@
2     requires 0 <= length;
3     requires \valid(arr + (0 .. length-1));
4     assigns arr[0 .. length-1];
5 */
6 void clear_array(int* arr, int length)
7 {
8     for(int i = 0; i < length; i++)
```



```

9      {
10         arr[i] = 0;
11     }
12 }

```

We can see that `\valid` can take a range of pointers by including the interval of indices of the array. The assigns keyword we have seen many times before now as well, the only difference here is that we can use a special array syntax to specify, that the `clear_array` function can read and write to any address pointed to by the given input array. For specifying the postcondition, we need another keyword from ACSL, namely `\forall`:

```

1  /*@
2     requires 0 <= length;
3     requires \valid(arr + (0 .. length-1));
4     assigns arr[0 .. length-1];
5     ensures \forall int i; 0 <= i < length ==> arr[i] == 0;
6  */
7  void clear_array(int* arr, int length)
8  {
9      for(int i = 0; i < length; i++)
10     {
11         arr[i] = 0;
12     }
13 }

```

The syntax of `\forall` is fairly self explanatory, the only interesting part is probably the implication primitive `==>` which we know from our basic logic courses. If we ask Frama-C to verify our code now by issuing

```

user@linux:~$ frama-c-gui -wp -wp-rte clear_array.c

```

it will give a negative result:

```

1  /*@ requires 0 ≤ length;
2     requires \valid(arr + (0 .. length - 1));
3     ensures ∀ int i; 0 ≤ i < \old(length) => *(&old(arr) + i) == 0;
4     assigns *(arr + (0 .. length - 1));
5  */
6  void clear_array(int *arr, int length)
7  {
8      int i = 0;
9      while (i < length) {
10         /*@ assert rte: mem_access: \valid(arr + i); */
11         *(arr + i) = 0;
12     }
13     /*@ assert rte: signed_overflow: i + 1 ≤ 2147483647; */
14     i++;
15 }
16 return;
17 }

```

This is due to the fact that Frama-C does not know what to make of our for loop. Loops are difficult in deductive verification, since we usually do not a-priori know how many times we will execute them. If we knew, we would not need a loop in the first place, we could just copy the loop body the required amount of times (better leave that task to the compiler though!). Frama-C does not even know if the

loop will actually ever terminate! So we have to supply a bit more information to help Frama-C in the proving procedure.

4.2 Loop invariants

Certain things are true before we enter a loop, and they are still true at the end of every loop iteration. These kinds of properties are called *loop invariants*. More precisely, a loop invariant is true whenever we check the looping condition. In our example function `clear_array`, we have two invariants. One is about the iteration variable `i`, for which we know the lower and upper value boundry (zero and length respectively). The other is about the value of the array elements we have already set to zero. We can express this in the following way:

```
clear_array.c
1  /*@
2     requires 0 <= length;
3     requires \valid(arr + (0 .. length-1));
4     assigns arr[0 .. length-1];
5     ensures \forall int i; 0 <= i < length ==> arr[i] == 0;
6  */
7  void clear_array(int* arr, int length)
8  {
9      /*@
10         loop invariant 0 <= i <= length;
11         loop invariant \forall int j; 0 <= j < i ==> arr[j] == 0;
12      */
13      for(int i = 0; i < length; i++)
14      {
15          arr[i] = 0;
16      }
17  }
```

As you can see, we can annotate a loop inside a function with its own contract. Loop invariants are introduced with the *loop invariant* keyword, and just like pre- and postconditions they are logical formulas. The two invariants we have introduced in the code shown above are hopefully self-explanatory. If we try and run Frama-C again on our changed contract, we unfortunately do not get a better result than before, so we still need to provide some more information.

4.3 Loop assignments

Similar to how Frama-C could not automatically deduce which memory locations a function can access, it can also not reason about memory assignments inside a loop. We therefore have to manually provide a list of variables and locations that can change inside our loop body. This is done through the *loop assigns* clause:

```
clear_array.c
1  /*@
2     requires 0 <= length;
3     requires \valid(arr + (0 .. length-1));
4     assigns arr[0 .. length-1];
5     ensures \forall int i; 0 <= i < length ==> arr[i] == 0;
```

```

6  */
7  void clear_array(int* arr, int length)
8  {
9      /*@
10         loop invariant 0 <= i <= length;
11         loop invariant \forall int j; 0 <= j < i ==> arr[j] == 0;
12         loop assigns i, arr[0 .. length-1];
13     */
14     for(int i = 0; i < length; i++)
15     {
16         arr[i] = 0;
17     }
18 }

```

This pretty much follows the same syntax as the assigns clause for function contracts. Any variable, that is visible in the function but is not listed in the loop assigns clause, is considered to have the same value at the beginning and at the end of the loop execution. Lets see how Frama-C is doing with our new contract:

```

/*@ requires 0 <= length;
    requires \valid(arr + (0 .. length - 1));
    ensures \forall int i; 0 <= i < \old(length) => *(&\old(arr) + i) == 0;
    assigns *(arr + (0 .. length - 1));
*/
void clear_array(int *arr, int length)
{
    int i = 0;
    /*@ loop invariant 0 <= i <= length;
        loop invariant \forall int j; 0 <= j < i => *(arr + j) == 0;
        loop assigns i, *(arr + (0 .. length - 1));
    */
    while (i < length) {
        /*@ assert rte: mem_access: \valid(arr + i); */
        *(arr + i) = 0;
        /*@ assert rte: signed_overflow: i + 1 <= 2147483647; */
        i ++;
    }
    return;
}

```

Fairly well! It could prove our postcondition and also verify that we do not change any memory locations outside of the boundaries of the array. So let's try and see how it performs when used in another function:

```

1  /*@
2     requires 0 <= length;
3     requires \valid(arr + (0 .. length-1));
4     assigns arr[0 .. length-1];
5     ensures \forall int i; 0 <= i < length ==> arr[i] == 0;
6
7  */
8  void clear_array(int* arr, int length)
9  {
10     /*@
11        loop invariant 0 <= i <= length;
12        loop invariant \forall int j; 0 <= j < i ==> arr[j] == 0;
13        loop assigns i, arr[0 .. length-1];
14    */
15    for(int i = 0; i < length; i++)
16    {

```

```

17     arr[i] = 0;
18 }
19 }
20
21 void main(void)
22 {
23     int a[] = {1, 2, 3, 4, 5, 6};
24     int len = 6;
25
26     clear_array(a, len);
27     //@ assert \forall int i; 0 <= i < len ==> a[i] == 0;
28 }

```

If we ask Frama-C to verify the above code, it will happily do so. However, we have actually not fully specified the intended behavior of our function yet.

4.4 Loop variants

There is an important notion in deductive program verification about the correctness of programs. What Frama-C tries to provide is a proof of partial correctness. This means, that it will try and prove our postconditions from our preconditions, *assuming that the code we provide terminates*. We then call the proven program *partially correct*. If we can successfully prove its termination as well, then we speak of *total correctness*. In our case, Frama-C actually succeeded in proving termination of the loop itself. This is not always the case. It is therefore good practice to define a *loop variant*. A loop variant is an integer value that at each beginning of a loop iteration must be non-negative, and which must strictly decrease with each loop iteration. Providing such a loop variant allows Frama-C to prove termination of the loop itself.

A complete contract for our `clear_array` function looks like the following:

```

clear_array.c
1  /*@
2   requires 0 <= length;
3   requires \valid(arr + (0 .. length-1));
4   assigns arr[0 .. length-1];
5   ensures \forall int i; 0 <= i < length ==> arr[i] == 0;
6
7  */
8  void clear_array(int* arr, int length)
9  {
10     /*@
11      loop invariant 0 <= i <= length;
12      loop invariant \forall int j; 0 <= j < i ==> arr[j] == 0;
13      loop assigns i, arr[0 .. length-1];
14      loop variant length - i;
15     */
16     for(int i = 0; i < length; i++)
17     {
18         arr[i] = 0;
19     }
20 }
21

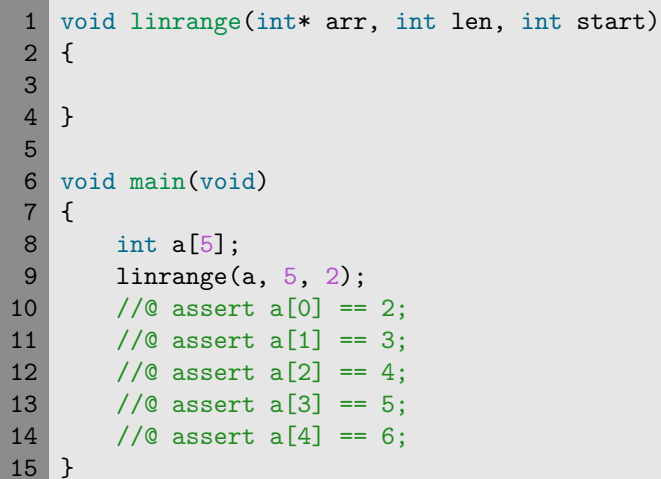
```

If Frama-C succeeds in proving the loop variant, we know that the loop will always terminate.

4.5 Exercises

4.5.1 Linrange function

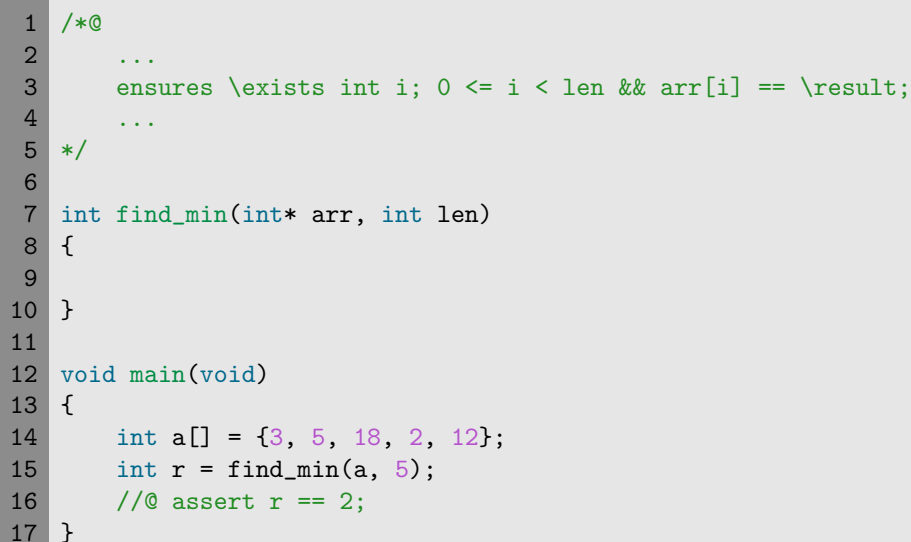
Implement and specify the following **linrange** function which takes as inputs an array, its length and a starting value, and assigns incrementing values to each array element starting with the start value:



```
1 void linrange(int* arr, int len, int start)
2 {
3
4 }
5
6 void main(void)
7 {
8     int a[5];
9     linrange(a, 5, 2);
10    //@ assert a[0] == 2;
11    //@ assert a[1] == 3;
12    //@ assert a[2] == 4;
13    //@ assert a[3] == 5;
14    //@ assert a[4] == 6;
15 }
```

4.5.2 Find minimum function

Implement and specify the **find_min** function, which takes an array of integers and the length of the array as input and returns the smallest element of the array:



```
1 /*@
2 ...
3     ensures \exists int i; 0 <= i < len && arr[i] == \result;
4 ...
5 */
6
7 int find_min(int* arr, int len)
8 {
9
10 }
11
12 void main(void)
13 {
14     int a[] = {3, 5, 18, 2, 12};
15     int r = find_min(a, 5);
16     //@ assert r == 2;
17 }
```

To formulate this contract you will need to use `\exists` keyword to specify that a value should exist that satisfies a property. A `requires` clause with this keyword was added to the function's contract. Depending on how you implement the function and formulate the contract you might run into the problem that Frama-C is unable to prove parts of your (possibly complete) postcondition correct. This happens with this task in particular if your postcondition makes a statement regarding the index of the smallest element in the array, while the implementation does not record this information. This mismatch between the implementation and contract gives Frama-C difficulties when applying the deduction procedure. You have multiple options of how to mitigate the problem. You can avoid statements about the index of the smallest element in your contract. If you would like to use the index of the smallest element in the postcondition, then you can either modify the implementation by using a local variable that actually records the (current) position of the minimum element, or you can make use of a **ghost variable**. Ghost variables are variables that we need for the proof procedure, which are however not useful for the actual implementation of the algorithm. They have the advantage of not showing up in the final code and therefore saving space. You can read more on ghost variables in the ACSL manual [1] in section 2.12.

Chapter 5

Behaviors

As you have seen in the previous chapter, our contracts are becoming longer and longer. At a certain point it is easy to loose track of what we are trying to prove, especially when you have many different possible results that we need to combine with consecutive `||` operators.

Frama-C therefore offers a nice way of dividing a contract into multiple different behaviors. Let's look at an example again. This example is adapted from [5]:

```
1 int array_equals(int* a, int* b, int n);
```

I leave the implementation of this function as an exercise. We provide the pointers to two arrays `a` and `b` and an integer `n`. If the first `n` elements of both arrays are equal, `array_equals` shall return 1, otherwise 0. This leads us to the following preconditions:

```
1 /*@
2   requires n >= 0;
3   requires \valid_read(a + (0 .. n-1));
4   requires \valid_read(b + (0 .. n-1));
5
6   assigns \nothing;
7 */
8 int array_equals(int* a, int* b, int n);
```

We only need to read from the array locations, therefore we use `\valid_read` instead of `\valid` here. The assigns clause is therefore also obvious.

We can formulate our postcondition as follows:

```
1 /*@
2   requires n >= 0;
3   requires \valid_read(a + (0 .. n-1));
4   requires \valid_read(b + (0 .. n-1));
5
6   assigns \nothing;
```

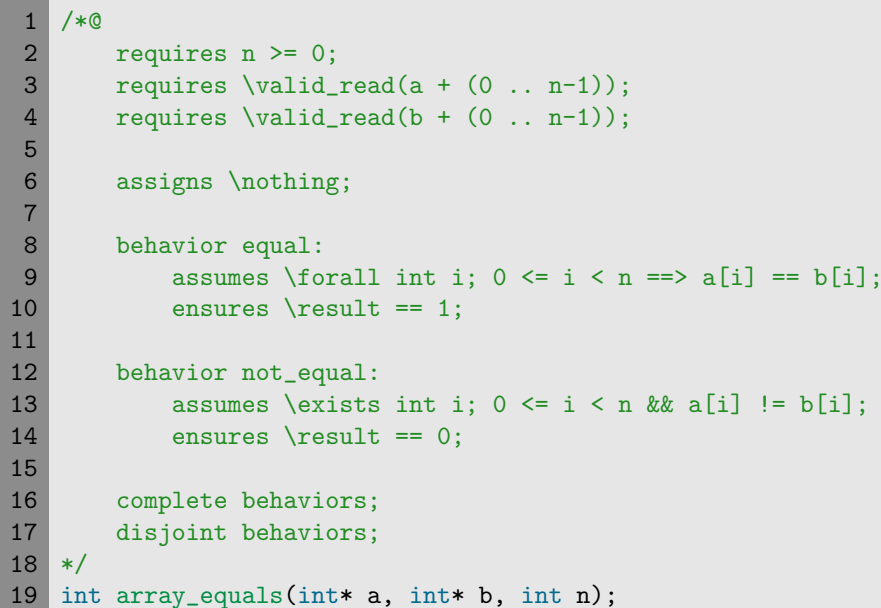
```

7
8   behavior equal:
9       assumes \forall int i; 0 <= i < n ==> a[i] == b[i];
10      ensures \result == 1;
11
12   behavior not_equal:
13       assumes \exists int i; 0 <= i < n && a[i] != b[i];
14       ensures \result == 0;
15  */
16  int array_equals(int* a, int* b, int n);

```

There are a few noteworthy things about the code above. As you can see, we can introduce different behaviors with the **behavior** <label> construct. The label is not important, and only interesting for us, Frama-C is mostly going to ignore it. Another new keyword is *assumes*, which holds the logical condition for each introduced behavior clause. We can then formulate different postconditions that are only true for the specific behavior they have been introduced in.

By default, Frama-C does not check if our different behavior clauses are complete (i.e. they cover all possible cases), or if they are disjoint (i.e. their assumes clauses do not overlap). We can ask Frama-C to check for these properties though:



```

1  /*@
2     requires n >= 0;
3     requires \valid_read(a + (0 .. n-1));
4     requires \valid_read(b + (0 .. n-1));
5
6     assigns \nothing;
7
8     behavior equal:
9         assumes \forall int i; 0 <= i < n ==> a[i] == b[i];
10        ensures \result == 1;
11
12    behavior not_equal:
13        assumes \exists int i; 0 <= i < n && a[i] != b[i];
14        ensures \result == 0;
15
16    complete behaviors;
17    disjoint behaviors;
18  */
19  int array_equals(int* a, int* b, int n);

```

5.1 Exercises

5.1.1 Implementation of array_equals

Implement the **array_equals** function. You will have to write the loop contract inside it as well. Try using it in a main function and prove the equivalence of two example arrays.

5.1.2 Mismatch function

Create a **mismatch** function with the following signature:

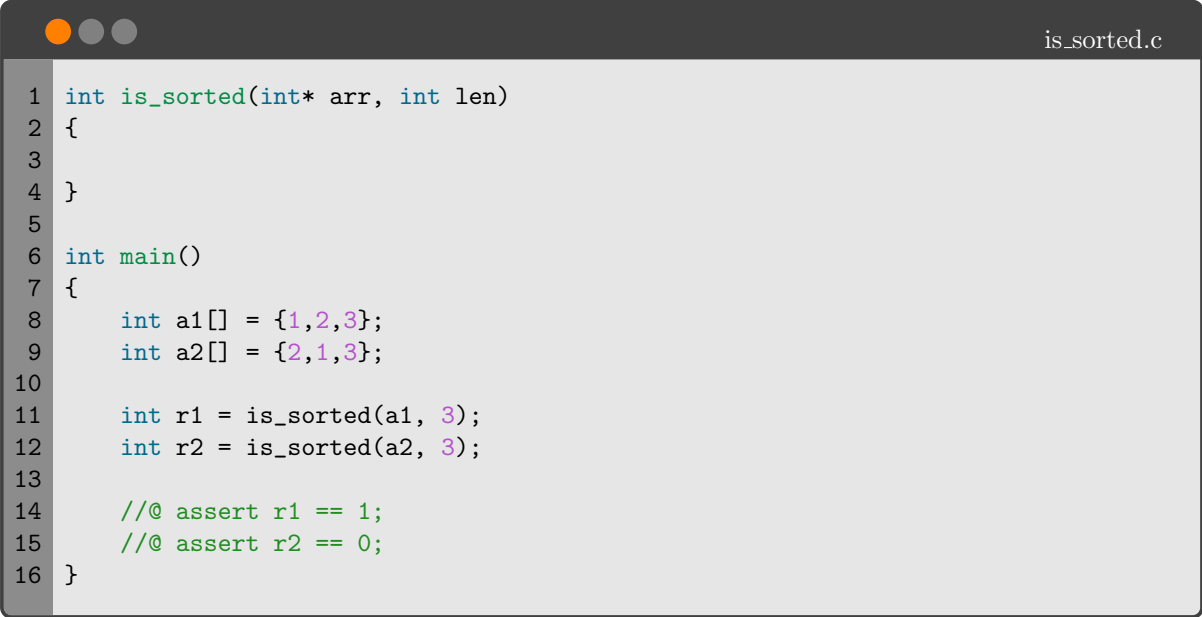


```
1 int mismatch(int* a, int* b, int n);
```

The **mismatch** function shall check the two given arrays up to length *n* and return the first index for which they differ. If they are equal on all first *n* elements, the function shall return *n* itself.

5.1.3 Is_sorted function

Implement the **is_sorted** function according to the following example usage:



```
1 int is_sorted(int* arr, int len)
2 {
3
4 }
5
6 int main()
7 {
8     int a1[] = {1,2,3};
9     int a2[] = {2,1,3};
10
11     int r1 = is_sorted(a1, 3);
12     int r2 = is_sorted(a2, 3);
13
14     //@ assert r1 == 1;
15     //@ assert r2 == 0;
16 }
```

The **is_sorted** function shall return 1 if the given array is sorted, and 0 otherwise. To ease the proof, you can assume/require that the array is not empty.

Bibliography

- [1] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C Specification Language Version 1.18. <https://frama-c.com/download/acsl.pdf>.
- [2] Allan Blanchard. Introduction to C program proof with Frama-C and its WP plugin. https://github.com/AllanBlanchard/tutoriel_wp, July 2020.
- [3] Loic Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Armand Puccetti, Julien Signoles, and Boris Yakobowski. Frama-C User Manual. <https://frama-c.com/download/frama-c-user-manual.pdf>.
- [4] Zafer Esen and Philipp Rümmer. Tricera: Verifying c programs using the theory of heaps. In *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN-FMCAD 2022*, page 380, 2022.
- [5] Jens Gerlach, Malte Brodmann, Jochen Burghardt, Andreas Carben Robert Clausecker, Denis Efremov, Liangliang Gu, Kerstin Hartig, Timon Lapawczyk, Hans Werner Pohl, Tim Sikatzki, Juan Soto, and Kim Völlinger. ACSL by Example. <https://github.com/fraunhoferfokus/acsl-by-example>, November 2020.
- [6] Virgile Prevosto. ACSL Mini-Tutorial. <https://frama-c.com/download/acsl-tutorial.pdf>.