# Predicates and Logical Functions



## Predicates and logical functions

- The language of logic expressions used in annotations can be extended by declarations of new logic types, and new constants, logic functions and predicates.
- New functions and predicates can be defined by explicit expressions.

```
//@ predicate is_positive(integer x) = x > 0;

//@ logic real double(real x) = x+x;
```

- Instead of an explicit definition, one may introduce axiomatic definitions.

It is up to the user to ensure that the introduction of axioms does not lead to logical inconsistencies.



## Axiomatic definitions

```
/*@ axiomatic factorial {

    predicate isfact(integer n, integer r);

    axiom isfact0:  isfact(0,1);

    axiom isfactn:  \forall integer n, f;
                        isfact (n-1,f) ==> isfact(n,f*n);

    logic integer fact (integer n);

    axiom fact1:  \forall integer n; isfact (n,fact(n));

    axiom fact2:
         \forall integer n, f; isfact (n,f) ==> f==fact(n);
    }
*/
```



## Inductive predicates

- A predicate may also be defined inductively.
- An alternative to define the predicate `isfact` would be:

```
/*@ inductive isfact(integer n, integer f) {
       case isfact0:  isfact(0,1);
       case isfactn:  \forall integer n, f;
                         isfact(n-1,f) ==> isfact(n,f*n);
    }
*/
```

- The predicate is inductively defined by a set of Horn clauses, *à la* Prolog.
- It is up to the user to ensure that the introduction of inductive definitions does not lead to a logical inconsistency.

## Example 12 (fact)

Run WP with **fact_init.c** and complete the proof making the necessary annotations in the loop.

```
/*@ requires n >= 0;
    ensures \result == fact(n);
*/
int fact (int n) {
  int f = 1;
  int i = 1;

  while (i <= n) {
    f = f * i;
    i = i + 1;
  }
  return f;
}
```



## Example 12 (fact: solution)

```
/*@ requires n >= 0;
    ensures \result == fact(n);
*/
int fact (int n) {
  int f = 1;
  int i = 1;
  /*@ loop invariant i<=n+1 && f == fact(i-1);
      loop assigns f, i;
      loop variant n+1-i;  */
  while (i <= n) {
    f = f * i;
    i = i + 1;
  }
  return f;
}
```



## Example 13 (swap)

Write the contract for swap. See **swap_init.c**.

```
void swap(int t[], int i, int j) {
  int tmp = t[i];
  t[i] = t[j];
  t[j] = tmp;
}
```



## Example 13 (swap: solution)

```
/*@ requires \valid(t+i) && \valid(t+j);
    ensures t[i] == \old(t[j]) && t[j] == \old(t[i]);
    assigns t[i], t[j];
*/
void swap(int t[], int i, int j) {
  int tmp = t[i];
  t[i] = t[j];
  t[j] = tmp;
}
```

## Hybrid functions and predicates

- Logic functions and predicates may take both (pure) C types and logic types arguments.
- Hybrid functions and hybrid predicates can either be defined (or axiomatized) with the same syntax as before.
- An hybrid function (or predicate) usually depends on one or more program points, because it depends upon memory states.
- To make such definitions safe, it is mandatory to add after the declared identifier a set of labels, between curly braces.



## Example 13 (hybrid predicate)

We can define the following hybrid predicate.

```
/*@ predicate Swap{L1,L2}(int* a, integer i, integer j) =
      \at(a[i],L1) == \at(a[j],L2) &&
      \at(a[i],L2) == \at(a[j],L1) &&
      \forall integer k; k!=i && k!=j
                         ==> \at(a[k],L1) == \at(a[k] ,L2);
*/
```

`Swap{L1,L2}(a,i,j)` has the meaning that the contents of array `a` in states `L1` and `L2` are the same, with the exception of indexes `i` and `j`, which are swapped.



## Example 13 (hybrid predicate)

The contract for the function `swap` could now be written as follows (see **swap_predicate.c**)

```
/*@ requires \valid(t+i) && \valid(t+j);
    ensures Swap{Old,Here}(t,i,j);
    assigns t[i], t[j];
*/
void swap(int t[], int i, int j) {
  int tmp = t[i];
  t[i] = t[j];
  t[j] = tmp;
}
```



## Example 14 (`partition`: contract)

Consider the following contract for `partition`.

```
/*@
requires 0 <= p <= r && \valid(A+(p..r));
ensures p<=\result<=r;
ensures \forall integer l; p<=l<\result ==> A[l]<=A[\result];
ensures \forall integer l; \result<l<=r ==> A[l]>A[\result];
ensures A[\result] == \old(A[r]);
assigns A[p..r];
*/
int partition (int A[], int p, int r)
```

What does it tell us?

## Example 14 (partition: implementation)

This is a possible implementation of the contract.
See **partition_swap_init.c**

```
int partition (int A[], int p, int r){
  int x = A[r];
  int j, i = p-1;

  for (j=p; j<r; j++)
    if (A[j] <= x) {
      i++;
      swap(A,i,j);
    }
  swap(A,i+1,r);
  return i+1;
}
```

Find the invariants and prove the correction of this implementation.



## Example 14 (partition: solution)

```
int partition (int A[], int p, int r) {
  int x = A[r];
  int j, i = p-1;

  /*@ loop invariant p <= j <= r && p-1 <= i < j;
      loop invariant \forall integer k; p<=k<=i ==> A[k]<=x;
      loop invariant \forall integer k; i<k<j ==> A[k]>x;
      loop invariant A[r] == x;
      loop assigns j, i, A[p..r];
      loop variant r-j;
  */
  for (j=p; j<r; j++)
    if (A[j] <= x) {
      i++;
      swap(A,i,j);
    }
  swap(A,i+1,r);
  return i+1;
}
```



## Example 14 (permutation)

- The `partition` routine should preserve the elements contained in the original array. The contract of `partition` is incomplete... (give an example.)
- An important property that has been left out is that the multiset of elements in the input array must be preserved in the output.
- Another way of stating this is that there exists a bijection on the set of indices that establishes a permutation between the two arrays.
- It is not easy to formalize this property. Let us try...



## Example 14 (permutation: tentative solutions)

Let us try to formalize this property. Let $B[]$ to denote $A[]$ in the poststate.

- First attempt. *Comments?*

$$\forall k.\, p \le k \le r \rightarrow (\,\exists l.\, p \le l \le r \rightarrow A[k] = B[l]\,)$$
$$\wedge$$
$$\forall k.\, p \le k \le r \rightarrow (\,\exists l.\, p \le l \le r \rightarrow B[k] = A[l]\,)$$

**Too weak!** (give a counterexample)

It does not take into account number of occurrences (i.e. preserves the set but not the multiset).

- Second attempt. *Comments?*

$$\forall k.\, p \le k \le r \rightarrow (\,\exists l.\, p \le l \le r \rightarrow A[k] = B[l] \wedge A[l] = B[k]\,)$$

**Too strong!** (give a counterexample)

It only covers the cases in which $B$ is directly obtained from $A$ by swaping pairs of elements. A sequence of swaps produces an array that is no longer related to the original in this way.

## Example 14 (permutation: a solution)

- The property to be expressed as a postcondition is that the array in the poststate is a *permutation* of the original array.
- One possibility to treat permutations is to see them as sequences of pairwise swaps.
- The hybrid perdicate

  `Permut{L1,L2}(int *a, integer l, integer h)`

  means that array a contains in state L2, between indices `l` and `h`, a permutation of the elements contained in a, in the same range, in state L1.



## Example 14 (permutation: a solution)

```
/*@
inductive Permut{L1,L2}(int *a, integer l, integer h) {
   case Permut_refl{L}:
        \forall int *a, integer l, h; Permut{L,L}(a,l,h);
   case Permut_sym{L1,L2}:
        \forall int *a, integer l, h;
            Permut{L1,L2}(a,l,h) ==> Permut{L2,L1}(a,l,h);
   case Permut_trans{L1,L2,L3}:
        \forall int *a, integer l, h;
            Permut{L1,L2}(a, l, h) && Permut{L2,L3}(a, l, h)
                 ==> Permut{L1,L3}(a,l,h);
   case Permut_swap{L1,L2}:
        \forall int *a, integer l, h, i, j;
            l<=i<=h && l<=j<=h && Swap{L1,L2}(a,i,j)
                 ==> Permut{L1,L2}(a,l,h);
}
*/
```



## Example 14 (`partition`: complete contract)

```
/*@ requires 0 <= p <= r && \valid(A+(p..r));
    ensures p<=\result<=r;
    ensures \forall integer l;
                    p <= l < \result ==> A[l] <= A[\result];
    ensures \forall integer l;
                    \result < l <= r ==> A[l] > A[\result];
    ensures  A[\result] == \old(A[r]);
    ensures Permut{Old,Here}(A,p,r);
    assigns A[p..r];
*/
int partition (int A[], int p, int r);
```

Load **partion_permutation_init.c**, complete the contract and finish its proof.



## Example 14 (complete solution)

```
int partition (int A[], int p, int r) {
  int x = A[r];
  int j, i = p-1;

  /*@ loop invariant p <= j <= r && p-1 <= i < j;
      loop invariant \forall integer k; p<=k<=i ==> A[k]<=x;
      loop invariant \forall integer k; i<k<j ==> A[k]>x;
      loop invariant A[r] == x;
      loop invariant Permut{Pre,Here}(A,p,r);
      loop assigns j, i, A[p..r];
      loop variant r-j;
  */
  for (j=p; j<r; j++)
    if (A[j] <= x) {
      i++;
      swap(A,i,j);
    }
  swap(A,i+1,r);
  return i+1;
}
```

## Lemmas

- One can devise additional assertions or ACSL lemmas to guide the automatic provers.
- Lemmas are user-given propositions, a facility that might help theorem provers to establish validity of the VCs.
  - The reason for this is that ACSL lemmas usually have a much smaller set of hypotheses than proof obligations directly related to the C code.
- Lemmas will generate proof obligations (perhaps to be proved interactively, since they will possibly be complex).
- A complete verification of an ACSL specification has to provide a proof for each lemma.



## Example 14 (a lemma about permutation)

```
/*@ lemma Permut_swap_sequence{L1,L2,L3}:
      \forall int *a, integer l, h, i, j;
              Permut{L1,L2}(a, l, h)
                  ==> l<=i<=h ==> l<=j<=h
                      ==> Swap{L2,L3}(a, i, j)
                          ==> Permut{L1,L3}(a, l, h);
*/
```

Prove the lemma present in **partion_permutation_init.c**.



## Exemple 15 (`max_subarray`: a more complex example)

Let us take a more complex example.

```
int max_subarray(int *a, int len) {
  int max = 0;
  int cur = 0;
  for (int i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0) cur = 0;
    if (cur > max) max = cur;
  }
  return max;
}
```

- We want to prove that this function returns the value of the maximal sum of subarrays (segments) of a given array.
- In order to specify this function, we will need an axiomatic definition about sum.



## Exemple 15 (the predicate `sum`)

- Here is an axiomatic definition about predicate `sum`.

```
/*@
  axiomatic Sum_array{
    logic integer sum(int* array, integer begin, integer end)
                                                  reads array[begin .. (end-1)];

    axiom empty:
      \forall int* a, integer b, e; b >= e ==> sum(a,b,e) == 0;
    axiom range:
      \forall int* a, integer b, e; b < e ==> sum(a,b,e) == sum(a,b,e-1)+a[e-1];
  }
*/
```

- The `reads` clause allows specifying the *footprint* of a hybrid predicate or function, that is, the set of memory locations that it depends on.
  - From such information, one might deduce properties of the form $f\{L1\}(args) = f\{L2\}(args)$ if it is known that between states $L1$ and $L2$, the memory changes are disjoint from the declared footprint.

## Exemple 15 (max_subarray: the specification)

The specification of the function max_subarray is the following:

```
/*@
  requires \valid(a+(0..len-1));
  assigns \nothing;
  ensures \forall integer l, h;
              0 <= l <= h <= len ==> sum(a,l,h) <= \result;
  ensures \exists integer l, h;
              0 <= l <= h <= len &&  sum(a,l,h) == \result;
*/
int max_subarray(int *a, int len);
```



## Exemple 15 (max_subarray: proving the specification)

```
/*@
  requires \valid(a+(0..len-1));
  assigns \nothing;
  ensures \forall integer l, h;
              0 <= l <= h <= len ==> sum(a,l,h) <= \result;
  ensures \exists integer l, h;
              0 <= l <= h <= len &&  sum(a,l,h) == \result;
*/
int max_subarray(int *a, int len) {
  int max = 0;
  int cur = 0;
  for (int i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0) cur = 0;
    if (cur > max) max = cur;
  }
  return max;
}
```



## Exemple 15 (max_subarray: proving the specification)

About the proof of this specification:

- When we want to add the loop invariant, we will realize that we miss some information.
- We want to express what are the values max and cur and what are the relations between them, but we cannot do it!
- Basically, our postcondition needs to know that there exists some bounds low and high such that the computed sum corresponds to these bounds. However, in our code, we do not have anything that express it from a logic point of view.
- We can then use **ghost code** to record these bounds and express the loop invariant.



## Ghost code

- **Ghost code** is regular C code, only visible from the specifications, that is only allowed to modify ghost variables.
- The idea is to add variables and source code that will not be part of the actual program but will model logic states that will only be visible from a proof point of view.
- Using it, we can make explicit some logic properties that were previously only known implicitly.
- Ghost code is added using annotations that will contain C code introduced using the ghost keyword:

```
/*@ ghost
  // code in C language
*/
```

We must be careful using ghost code! The tool will not perform any verification to ensure that we do not write in the memory of the program by mistake.

## Exemple 15 (max_subarray: ghost code)

- We will first need two variables, `low` and `high`, that will allow us to record the bounds of the maximum sum range.
  - Every time we will find a range where the sum is greater than the current one, we will update our ghost variables.
  - This bounds will then correspond to the sum currently stored by `max`.
- We need other bounds: the ones that correspond to the sum store by the variable `cur` from which we will build the bounds corresponding to current `low` bound.
  - For these bounds, we will only add a single ghost variable: the current low bound `cur_low`, the high bound being the variable `i` of the loop.



## Exemple 15 (max_subarray: ghost code)

```
int max_subarray(int *a, int len) {
  int max = 0;
  int cur = 0;
  //@ ghost int cur_low = 0;
  //@ ghost int low = 0;
  //@ ghost int high = 0;

  for (int i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0) {
      cur = 0;
      /*@ ghost cur_low = i+1; */
    }
    if (cur > max) }
      max = cur;
      /*@ ghost low = cur_low; */
      /*@ ghost high = i+1; */
    }
  }
  return max;
}
```



## Exemple 15 (max_subarray: loop annotations)

```
int max_subarray(int *a, int len) {
  int max = 0;
  int cur = 0;
  //@ ghost int cur_low = 0;
  //@ ghost int low = 0;
  //@ ghost int high = 0;

  /*@
    loop invariant BOUNDS: low <= high <= i <= len && cur_low <= i;

    loop invariant REL:    cur == sum(a,cur_low,i) <= max == sum(a,low,high);
    loop invariant POS:    \forall integer l;    0 <= l <= i       ==> sum(a,l,i) <= cur;
    loop invariant POS:    \forall integer l, h; 0 <= l <= h <= i ==> sum(a,l,h) <= max;

    loop assigns i, cur, max, cur_low, low, high;
    loop variant len - i;
  */
  for (int i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0) {
      cur = 0;
      /*@ ghost cur_low = i+1; */
    }
    if (cur > max) }
      max = cur;
      /*@ ghost low = cur_low; */
      /*@ ghost high = i+1; */
    }
  }
  return max;
}
```



## Exemple 15 (max_subarray: check **ghost.c**)

```
/*@ requires \valid(a+(0..len-1));
    assigns  \nothing;
    ensures  \forall integer l, h; 0 <= l <= h <= len ==> sum(a,l,h) <= \result;
    ensures  \exists integer l, h; 0 <= l <= h <= len &&  sum(a,l,h) == \result;
*/
int max_subarray(int *a, int len) {
  int max = 0;
  int cur = 0;
  //@ ghost int cur_low = 0;
  //@ ghost int low = 0;
  //@ ghost int high = 0;
  /*@
    loop invariant BOUNDS: low <= high <= i <= len && cur_low <= i;
    loop invariant REL:    cur == sum(a,cur_low,i) <= max == sum(a,low,high);
    loop invariant POS:    \forall integer l;    0 <= l <= i       ==> sum(a,l,i) <= cur;
    loop invariant POS:    \forall integer l, h; 0 <= l <= h <= i ==> sum(a,l,h) <= max;
    loop assigns i, cur, max, cur_low, low, high;
    loop variant len - i;
  */
  for (int i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0) {
      cur = 0;
      /*@ ghost cur_low = i+1; */
    }
    if (cur > max) }
      max = cur;
      /*@ ghost low = cur_low; */
      /*@ ghost high = i+1; */
    }
  }
  return max;
}
```

## Exercises

- Consider the following function that sorts an array in increasing order.

```
void maxSort (int *a, int size) {
  int i, j;

  for (i=size-1; i>0; i--) {
    j = maxarray(a,i+1);
    swap(a,i,j);
  }
}
```

1. Write a contract that guarantees the safety of this function and prove it.
2. Improve the function contract in order to guarantee that the array produced by the function is sorted in increasing order. Write the loop invariants in order to prove it.
3. Complete the contract in order to claim that the function implements a sorting algorithm. Then complete the proof.



## Exercises

- The following function counts the occurrences of `x` in the array `a` of size `n`.

```
int numOccur (int *a, int n, int x);
```

1. Declare a logical function `count` that determines the number of occurrences of a value in an array, and present an axiomatic definition for it.
2. Write a contract for `numOccur`.
3. Write the function definition and prove its safety and functional correctness.

- The following function reverse an array `a` of size `n`.

```
void reverse (int a[], int n);
```

1. Write a contract for `reverse`.
2. Write the function definition and prove its safety and functional correctness.