Apontamentos

- `\result` : palavra reservada para o valor final do programa.
- `\nothing` : palavra reservada para sinalizar que nenhuma variável é modificada.
- `\old(x)` : palavra reservada (APENAS PARA PÓS-CONDIÇÕES) que representa o valor da variável antes da função ser executada.

- **requires** : introduz uma pré-condição.
- **ensures** : introduz uma pós-condição.
- **assigns** : identifica variáveis cuja memória pode ser modificada pela função.
- **assert** : introduz uma propriedade que deve ser verificada a um certo ponto no programa.s

COMANDOS:

```
$ frama-c -wp -wp-rte <nome_do_ficheiro>.c
```

- Interface gráfica: `frama-c-gui`

Para *pointers*, nós temos as seguintes funções lógicas:

`\valid(p)` : por exemplo, `int *p`
`\valid(p+(0..n))` : por exemplo, `int p[0..n]`
`\separated(p+(0..n), q+(0..3))`
`\block_length(p)`

Exemplos

- Exemplo 1:

```
/*@
    ensures \result >= a && \result >= b;
    ensures \result == a || \result == b;
    assigns \nothing;
*/
int max(int a, int b) {
    return (a >= b) ? a : b ;
}
```

```
extern int x ;

int main() {
    x = 3;
    int r = max(4, 2);
    //@ assert r == 4 ;
    //@ assert x == 3 ;
}
```

A **pós-condição** `ensures \result >= a && \result >= b;` garante que, o resultado final será maior ou igual aos passados como argumento.

A **pós-condição** `ensures \result == a || \result == b;` garante que, o resultado final será um dos valores passados como argumento.

- Exemplo 2:

```
/*@ requires x > INT_MIN;
    ensures (x >= 0 ==> \result == x) &&
            (x < 0 ==> \result == -x);
    assigns \ nothing ;
*/
int abs (int x) {
    if (x >= 0) return x ;
    return -x ;
}
```

A **pré-condição** `requires x > INT_MIN;` exige que o input é estritamente maior que INT*MIN para garantir que não ocorre _under_flow*.

- Exemplo 3:

```
/*@ requires 0 < size && \valid (u+(0..size -1));
    ensures 0 <= \result < size;
    ensures
        \forall integer a; 0 <= a < size ==> u[a] <= u[\result];
    assigns \nothing;
*/
int maxarray(int u[], int size) {
    int i = 1;
    int max = 0;

    /*@ loop invariant \forall integer a;
            0 <= a < i ==> u[a] <= u[max];
```

```
        loop invariant 0 <= max < i <= size;
        loop assigns max , i;
        loop variant size -i;
    */
    while (i < size) {
        if (u[i] > u[max]) max = i;
        i++;
    }
    return max;
}
```

- Exemplo 4:

```
/*@
    requires \valid (p) && \valid (q);
    ensures \result >= *p && \result >= *q;
    ensures \result == *p || \result == *q;
    assigns \nothing;
*/
int max_ptr (int *p, int *q);
```

A **pré-condição** `requires \valid (p) && \valid (q);` exige que os *pointers* passados como argumento são válidos.

Uma maneira de não ser preciso indicar o `assigns \nothing` , é em vez de dizer `requires \valid (p) && \valid (q);` , dizer `requires \valid_read(p) && \valid_read(q);` .

Ficando o contrato deste modo:

```
/*@
    requires \valid_read(p) && \valid_read(q);
    ensures \result >= *p && \result >= *q;
    ensures \result == *p || \result == *q;
*/
```

- Exemplo 5:

```
#include <limits.h>
/*@ requires x > INT_MIN;
    assigns \nothing ;

    behavior pos :
        assumes x >= 0;
        ensures \result == x;
```

```
    behavior neg :
        assumes x < 0;
        ensures \result == -x;

    complete behaviors;
    disjoint behaviors;
*/
int abs (int x) {
    if (x >= 0) return x;
    return -x;
}
```

Dependendo do comportamento, ou seja, se o valor for positivo ou negativo, a pós-condição será diferente.

- **Exemplo 6**:

Porque é que este contrato falha?

```
/*@ requires \ valid (a) && \ valid (b);
    ensures *a==10 && *b==20;
    assigns *a, *b;
*/
void proc(int *a, int *b) {
    *a = 10;
    *b = 20;
}
```

Falta adicionar esta pré-condição:

```
requires a != b;
```

A função `findArray` procura um valor x num array ordenado **arr** de comprimento **len** e retorna o índice onde o valor é encontrado ou -1 se o valor não existir.

```
/*@ requires \forall integer i, j;
                0 <= i <= j < len ==> arr[i] <= arr[j];
    requires len >= 0;
    requires \valid (arr +(0..(len -1)));
    assigns \nothing ;

    behavior belongs:
        assumes \exists integer i;
                    0 <= i < len && arr[i] == x;
        ensures 0 <= \result < len;
        ensures arr[\result] == x;
```

```
    behavior not_belongs:
        assumes \forall integer i;
            0 <= i < len ==> arr[i] != x;
        ensures \result == -1;
*/
int find_array(int* arr , int len , int x);
```

Podemos ainda definir dois predicados **sorted** e **elem**, que indicam se um array está ordenado e se um valor é elemento do array, respetivamente.

```
/*@
    predicate sorted(int* arr , integer length) =
        \forall integer i, j; 0 <= i <= j < length
            ==> arr[i] <= arr[j];

    predicate elem(int v, int* arr , integer length) =
        \exists integer i; 0 <= i < length && arr[i] == v;
*/
```

Deste modo, o contrato anterior pode ser refatorizado para isto:

```
/*@ requires sorted(arr, len);
    requires len >= 0;
    requires \valid (arr +(0..(len -1)));
    assigns \nothing ;

    behavior belongs:
        assumes elem(x, arr, len);
        ensures 0 <= \result < len;
        ensures arr[\result] == x;

    behavior not_belongs:
        assumes !elem(x, arr, len);
        ensures \result == -1;
*/
int find_array(int* arr, int len, int x);
```

Em termos de verificações cíclicas, temos os seguintes invariantes e variantes de ciclo:

```
int find_array(int* arr , int len , int x) {
    int mean;
    int low = 0;
    int high = len - 1;

    /*@
        loop invariant 0 <= low;
        loop invariant high < len;
```

```
        loop invariant \forall integer i; 0 <= i < low ==> arr[i] < x;
        loop invariant \forall integer i; high < i < len ==> arr[i] > x;
        loop assigns low, high, mean;
        loop variant high - low + 1;
    */
    while (low <= high) {
        mean = low + (high -low) / 2;
        if (arr[mean] == x) return mean;
        if (arr[mean] < x) low = mean + 1;
        else high = mean - 1;
    }
    return -1;
}
```

- **Exemplo 7**:

```
/*@
    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;
}
```

Podemos simplicar este contrato com o predicato **Swap**:

```
/*@
    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);
*/
```

Ficando do seguinte modo:

```
/*@
    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;
}
```

- Exemplo 8:

```
/*@
    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) {
    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;
}
```

O contrato desta função pode ser refatorizado definindo o inductive Permut:

```
/*@
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);
}
*/
```

Prosseguindo para as substituições, temos:

```
/*@
    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) {
    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;
}
```