Summary of steps needed to verify while-loops

1. Select an invariant and show that the loop preserves the invariant.
2. Show that the invariant holds before entering the loop.
3. Show that the invariant and the negation of the loop condition imply the post-condition (possibly after some finalizing assignments).
4. Show that the loop terminates. For well-designed while-loops[1] often can include bounds for the loop variable in the invariant.

Additional Verification Techniques.

Next we discuss verification of for-loops. This material is from Chapter 4 in the textbook.

In order to avoid "re-inventing the wheel", we will rewrite a for-loop

$$\texttt{for}(\texttt{A}_0, \texttt{B}, \texttt{A}_1)\texttt{C}$$

as a while-loop

$$\texttt{A}_0; \texttt{while}(\texttt{B})\texttt{CA}_1;$$

[1] In general, termination is uncomputable. This will be discussed next week.

Now the proof tableau schema looks as follows:

```
ASSERT( P )  //pre-condition
...
A_0 ;
ASSERT(I)   // I is the invariant
while (B) {
     ASSERT( I && B )
     CA_1;
     ASSERT(I)
     } //end while
ASSERT( I && !B )
     ...
ASSERT(Q)    // post-condition
```

As an example we verify the partial correctness of the following:

```
ASSERT(0 <= n <= max)
{ int i;
  for (i=0; A[i] != x && i < n; i++)
    {}
  present = i<n;
}
ASSERT(present iff x in A[0:n-1])
```

Note that the code may not terminate normally if x does not occur in `A[0:n-1]`. Why?

As the loop invariant (denoted as `I`) we choose:

```
0<=i<=n && ForAll(k=0; k<i) x != A[k]
```

Using the schema for for-loops, we must verify the following:

```
ASSERT(pre-condition)

i=0; /* initial assignment */

ASSERT(I) /*loop invariant*/

while( A[i] != x && i < n) {

   ASSERT(I && A[i] != x && i < n)

   /* for-loop has empty body*/

   i++;

   ASSERT(I)

}

ASSERT(I && !(A[i] != x && i < n))

present = i<n;  /*final assignment*/

ASSERT(post-condition)
```

The complete construction is given in class. Here we have to be a little careful in how the post-condition is established from the invariant and the negation of the loop condition.

Array component assignment rule

To deal with array component assignments we need to modify the Hoare axiom schema so that instead of substituting individual array components we modify the entire array.

The notation `(A | I` $\mapsto$ `E)` refers to an array obtained from `A` by replacing the value at position `I` by the value of the expression `E`.

More formally,

$$(\mathtt{A}|\mathtt{I} \mapsto \mathtt{E})[\mathtt{I}'] = \begin{cases} \mathtt{E} \text{ when } \mathtt{I}' = \mathtt{I}, \\ \mathtt{A}[\mathtt{I}'] \text{ when } \mathtt{I}' \neq \mathtt{I}. \end{cases}$$

Now the modified array component assignment rule can be written as:

$$[\mathtt{Q}](\mathtt{A} \mapsto \mathtt{A}') \ \ \{\mathtt{A}[\mathtt{I}] = \mathtt{E};\} \ \ \mathtt{Q}$$

where `A'` is `(A | I` $\mapsto$ `E)`.

It has to be verified separately that the value of `I` is within the subscript range of the array `A`.

Example. Use the array component assignment axiom (two times) to find the weakest sufficient pre-condition `P` for the following code fragment

```
ASSERT( P )
A[i] = x;
A[k] = 5;
ASSERT( A[j] == 0 )
```

Above `x` is an integer variable, `A` is an array of integers and we assume that all the subscripts are within the range of subscripts for `A`.

Example. We consider an array of even length.

The program should move elements from even numbered positions to a contiguous chunk at the beginning, see Figure 1.

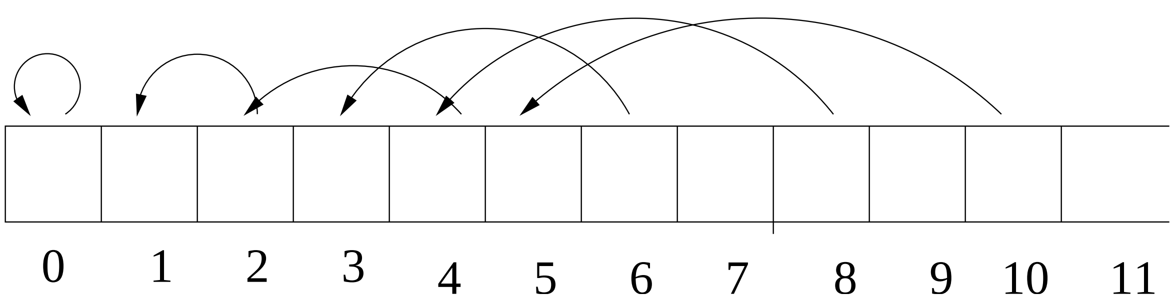

Figure 1: Moving of the array elements.

The specification for the program is as follows:

```
Interface:  const int n;
            Entry A[2n]; /*entries numbered 0,...,2n-1 */
```

```
Pre-condition:  n >= 1 && A == A0

Post-condition: ForAll(i=0; i<n) A[i] == A0[2i]
```

On the basis of the post-condition we can select a suitable loop invariant and using it "derive" the program. (To be done in class.)