Burt Rosenberg

Loop Invariants

28 SEPTEMBER, 1995

Overview

The Loop Invariant helps the programmer write while loops which function correctly. It consists of three parts:

1. A *Loop Invariant* is an assertion which is asserted just before the while loop is run the first time, and again after each pass of the loop is completed. It should give you a reasonable picture of what the while loop accomplishes with one step: it holds the invariant steady.

2. A *Termination* condition, and a drive towards termination. Using the method of Loop Invariants you should carefully distinguish between these two important aspects of a while loop: that progress is made towards termination at each pass through the loop; and that each pass through the loop starts with the loop invariant asserted and must end with the loop invariant again asserted.

3. The combination of Termination and the Loop Invariant occuring together must imply the *Goal*.

It is as if the drive forward towards termination tries to unbalance the data, pushing the invariant towards false with a little nudge each time through the loop. The code, after each nudge, reestablishes the balance, making the invariant again true. This gentle nudge-restore continues until the goal is obtained.

A simple example

Finding the maximum integer in an array:

```
int main(int argc, char *arv[])
{
   int A[ARRAY_SIZE], i, a ;
   fill_array( A, ARRAY_SIZE ) ;

   i = 1 ;
   a = A[0] ;
   /* LOOP INVARIANT:
      a = max ( A[j], for j = 0, ..., i-1 )
   */
  /* TERMINATION:
     i is pushed forward by 1 each time, and
     so it reaches ARRAY_SIZE
   */
   while ( i<ARRAY_SIZE )
   {
      if ( a < A[i] ) a = A[i] ;
      i++ ;
```

```
   }
   /* GOAL:
      a = max( A[j], for j = 0, ... , ARRAY_SIZE )
   */
}
```

Example: Selection Sort

Selection Sort of an array of integers works by finding the largest number in the array, placing it in the zero-th position, finding the next largest number in the array, that is, largest among the integers not yet selected, and placing it in the first position, and so on.

The invariant is that after each pass through the loop we have a bunch of integers in order in the first part of the array. The drive forward is that we try in each pass to expand by one the number of integers in order in the first part of the array. We terminated because we increment by one towards a fixed number: the size of the array. When we terminate we have reached our goal: the sorted region includes the entire array.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<assert.h>

/*
 * TITLE:   Selection Sort
 * PURPOSE: To demonstrate Loop Invariants.
 * AUTHOR:  Burt Rosenberg, U. Miami Dept. of Math.
 * DATE:    29 September 1995
*/

#define ARRAY_SIZE 20
#define MULT 17
#define MOD  23

/* The following is a macro which swaps the values
   in locations A and B. Use of the macro will substitute
   over variable names in for A and B.
*/
#define SWAP(A,B) {int temp; temp=A; A=B; B=temp;}

/* Subroutines which follow:
 *
 *   fill_array
 *   print_array
 *   index_of_largest
```

```
 *   main
*/

void fill_array( int a[], int array_size )
{
    int i ;
    for ( i=0; i<array_size; i++ )
       a[i] = (i+MULT)*MULT % MOD ;
}

void print_array( int a[], int array_size )
{
    int i ;
    for ( i=0; i<array_size; i++ )
       printf("%4d: %d\n", i, a[i] ) ;
}

int index_of_largest( int a[], int start_at, int end_before )
{
   int m_idx, idx ;
   assert( start_at<end_before ) ;

   m_idx = start_at ;
   idx =   start_at+1 ;
   /* LOOP INVARIANT:
      a[m_idx] is largest among a[start_at] ... a[idx-1]
   */
   while ( idx < end_before )
   {
      if ( a[m_idx] < a[idx] )
      {
          m_idx = idx ;
      }
      idx++ ;
   }
   return(m_idx) ;
}

int main( int argc, char * argv[] )
{
    int A[ARRAY_SIZE], i, n;

    fill_array( A, ARRAY_SIZE ) ;
    printf("Array before sorting:\n") ;
```

```
    print_array( A, ARRAY_SIZE ) ;

    i = 0 ;
    /* LOOP INVARIANT:
       The i largest numbers in the array A are sorted
       and in locations 0 ... i-1
    */
    while ( i<ARRAY_SIZE )
    {
       n = index_of_largest( A, i, ARRAY_SIZE ) ;
       SWAP( A[i], A[n] )
       i++ ;
    }

    printf("\nArray after sorting:\n") ;
    print_array( A, ARRAY_SIZE ) ;
    printf("\nChow, baby!\n") ;
}
```

Example: Insertion Sort

Insertion Sort of an array of integers keeps invariant that a region consisting the first i numbers in the array are sorted. At first, i is one, a single element by itself is always in sorted order! Each pass through the loop i is increment, attempting the expand the size of the sorted region of the array. The new element finds its place among the old by being inserted into the correct location.

The termination is assured by increment i towards a fixed integer bound: the array size. Each increment requires a little work to reestablish the invariant: that the first i numbers have been sorted. When i hits the termination value, then the goal is achieved: all the numbers in the array have been sorted.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<assert.h>

/*
 * TITLE:   Insertion Sort
 * PURPOSE: To demonstrate Loop Invariants.
 * AUTHOR:  Burt Rosenberg, U. Miami Dept. of Math.
 * DATE:    29 September 1995
*/

#define ARRAY_SIZE 20
#define MULT 17
```

```
#define MOD  23

/* Subroutines which follow:
 *
 *   fill_array
 *   print_array
 *   main
*/

void fill_array( int a[], int array_size )
{
    int i ;
    for ( i=0; i<array_size; i++ )
       a[i] = (i+MULT)*MULT % MOD ;
}

void print_array( int a[], int array_size )
{
    int i ;
    for ( i=0; i<array_size; i++ )
       printf("%4d: %d\n", i, a[i] ) ;
}

int main( int argc, char * argv[] )
{
    int A[ARRAY_SIZE] ;
    int i, n, a ;

    fill_array( A, ARRAY_SIZE ) ;
    printf("Array before sorting:\n") ;
    print_array( A, ARRAY_SIZE ) ;

    i = 1 ;
    /* LOOP INVARIANT:
       The first i items in the array A are sorted.
    */
    while ( i<ARRAY_SIZE )
    {
       n = i ;
       a = A[n] ;
       /* LOOP INVARIANT:
          if A[n] is set to a, then A[j] for n <= j <= i is sorted.
       */
       while ( n>0 && A[n-1]<a )
```

```
      {
         A[n] = A[n-1] ;
         n-- ;
      }
      A[n] = a ;
      i++ ;
   }


   printf("\nArray after sorting:\n") ;
   print_array( A, ARRAY_SIZE ) ;
   printf("\nChow, baby!\n") ;
}
```

Burt Rosenberg
Miami, 1995