Inspection and Data Abstraction

Data abstraction

- *Abstraction*: intentionally ignoring certain aspects of a problem to simplify analysis and to focus attention on the remaining aspects.
- In *procedural abstraction* the procedure implementation is ignored to focus attention on the service provided by the procedure.
- In *data abstraction* a collection of procedures (a module) provide access to some "hidden" data structures. The data structures and the procedure implementations are ignored to focus attention on the service provided by the module.
- Successful data abstraction requires a precise and complete specification of the service offered by the module.

Data abstraction in inspection

- Suppose that you are given specifications and implementations for the C function `F` and the module `M`, and that `F` calls one or more functions provided by `M`.
- To inspect `M`: show that every `M`'s implementation behaves as required by its specification.
- To inspect `F`: show that `F`'s implementation behaves as required by its specification. When reasoning about a call in `F`'s implementation to a function provided by `M`, refer to `M`'s specification and *not* its implementation.

Module interface specifications

- A Module Interface Specification (MIS) is a precise, written description of the required observable behavior of a module
- The need for standard sections:
  There will be many modules. The specifications will be far easier to read and write if the information is presented in the same form in each MIS.
- The standard sections in CSCI 265 module interface specifications:
  - overview
  - constants
  - types
  - exported functions
  - usage example

Example

- Inspect the `findItem`, `ILAdd`, and `main` functions shown below.
- First show that `findItem` and `ILAdd` are correct with respect to the `IntList` MIS, or produce a list of the faults found.
- Then show that `main` is correct or produce a list of the faults found. When reasoning about the calls to `IntList`, refer to its specification *not* its implementation.

Module Interface Specification: `IntList.h`

```
/********** IntList module---interface specification **********/

/***** overview *****
The IntList module stores a list of integers.  List elements
may be added, removed, modified, and retrieved at any position.
*/

/***** constants *****/

/***** types *****/

typedef enum {SUCCESS,MEMORYERROR,RANGEERROR} ILStatus;

/***** exported functions *****/

/* initialize the list to empty */
void ILInit();

/* if i is not in [0..ILSize()]
*       return RANGEERROR
* else if memory is available to add an additional element
*       add x at position i, shifting elements i,i+1, ... right
*       return SUCCESS
* else
*       return MEMORYERROR
*/
int ILAdd(int i,int x);

/* if i is in [0..ILSize()-1]
*       remove the element at position i, shifting elements i+1,i+2 ... left
* else
*       return RANGEERROR
*/
int ILRemove(int i);

/* if i is in [0..ILSize()-1]
*       replace the value at position i with x
* else
*       return RANGEERROR
*/
int ILSetVal(int i,int x);

/* if i is in [0..ILSize()-1]
*       return the value stored at position i
* else
```

```
*       return RANGEERROR
* Note: Because RANGEERROR is a possible element value, ILGetVal parameters
* should be explictly checked using ILSize
*/
int ILGetVal(int i);

/* return the number of elements in the list */
int ILSize();

/***** usage example *****
 *
 * Program:
 *      #include &stdio.h>
 *      #include "IntList.h"
 *      int main()
 *      {
 *              int i;
 *
 *              ILInit();
 *
 *              for (i = 0; i < 5; i++)
 *                      ILAdd(0,i*i);
 *
 *              for (i = 0; i < ILSize(); i++)
 *                      printf("%d\n",ILGetVal(i));
 *
 *              return 0;
 *      }
 *
 * Output:
 *      16
 *      9
 *      4
 *      1
 *      0
 */
```

Module Implementation (partial, with faults): `IntList.c`

```
#include <stdio.h>
#include "IntList.h"

/*****constants*****/

/*****types*****/

typedef struct item {
        int val;
        struct item *nextPtr;
        struct item *prevPtr;
} item;

/***** local variables *****/

static item *headPtr; /*pointer to the head of the list*/
static int size; /*list size*/
```

```
/***** local functions *****/

/* if i in [0..size-1] then
*       return the address of ith item
* else
*       return NULL
*/
static item* findItem(int i)
{
        item *tmpPtr = headPtr;

        if (i < 0 || i >= size)
                return NULL;
        while(tmpPtr && i--)
                tmpPtr = tmpPtr->nextPtr;
        return tmpPtr;
}

/***** exported functions *****/

int ILAdd(int i,int x)
{
        item *tmpPtr,*newPtr;

        if (i < 0 || i > size) {
                return RANGEERROR;
        }

        newPtr = (item*) malloc(sizeof(item));
        newPtr->val = x;

        if (size == 0) { /* empty list */
                headPtr = newPtr;
                newPtr->nextPtr = newPtr;
                newPtr->prevPtr = newPtr;
        } else { /* non-empty list */
                if (i == 0 || i == size) /* adding first or last */
                        tmpPtr = headPtr;
                else /* adding at any other position */
                        tmpPtr = findItem(i);
                newPtr->nextPtr = tmpPtr;
                newPtr->prevPtr = tmpPtr->prevPtr;
                tmpPtr->prevPtr->nextPtr = newPtr;
                tmpPtr->prevPtr = newPtr;
                if (i == 0)
                        headPtr = newPtr;
        }
}
```

Function that uses `IntList`: `main.c`

```
/* Read a list of integers from stdin
 * Print those integers that are greater than the list average
 *
 * Assumed: stdin contains a list of integers
```

```
 */
int main()
{
        int i,sum;
        float average;

        ILInit();

        while (scanf("%d",&i) != EOF) {
                ILAdd(ILSize(),i);
                sum += i;
        }
        average = (float)sum/ILSize();

        for (i = 0; i < ILSize(); i++)
                if (ILGetVal(i) > average)
                        printf("%d\n",ILGetVal(i));

        return 0;
}
```