Testing and Procedural Abstraction

Ad hoc testing versus systematic testing

- Usually testing is ad hoc:
  - Unplanned, left until the implementation is complete
  - Not maintained
  - Mostly manual
  - Consequently, it is expensive and ineffective

- Systematic testing is
  - Planned: to permit design for testability.
  - Maintained: so that the tests can be run on every new version
  - Exploits opportunities for automation, so that repeating the tests is very cheap
  - Can be: inexpensive and effective

Testing tasks

1. Build the test harness: utility programs, tools, stubs, and drivers.
2. Generate the inputs: choose inputs likely to cause failures.
3. Determine the expected outputs from the specification.
4. Execute the test cases, capturing the actual outputs.
5. Compare the actual and expected outputs: can be very expensive; planning pays off here.

Testing and Procedural Abstraction

- With procedural abstraction, the service provided by each function is carefully specified.
- Inspection is used to show that the function implementation is correct with respect to the specification. Testing is used for the same purpose. In combination, testing and inspection can be very effective in uncovering errors.
- A function F is written with some intended purpose, that is, it will later be called by some other function, say G. Ideally, F will be tested in isolation before it is called by G.

Stubs and drivers

- Suppose that the C function F must be tested in isolation.
- To produce a test program for F we must write code to invoke F and to check that the values returned by F are correct.

- A *driver* for function F is a program written to invoke F and check its behavior for correctness. Usually the driver invokes F repeatedly, and with a wide variety of parameter values.
- Suppose that F invokes some other function G. If the implementation for G is not available, to produce a test program for F, we must also write code to invoke simulate G.
- Suppose that F is the function under test and that F invokes G. A *stub* is code that serves as a substitute for G while testing F. Usually the stub for G is much simpler than G's actual implementation.

Example: `findLongestPlateau` driver

Specification for the function under test:

```
/* Assign to *pStart and *pLen the starting position and length
 * of the longest plateau in a[0..aLen-1].
 *
 * Assumed: a has at least aLen elements and aLen > 0
 */
void findLongestPlateau(int a[], int aLen, int* pStart, int *pLen);
```

Test driver:

```
void compare(int actStart,int actLen,int expStart,int expLen)
{
        if (actStart != expStart)
                printf("Start. Actual: %d Expected: %d\n",actStart,expStart);
        else if (actLen != expLen)
                printf("Length. Actual: %d Expected: %d\n",actLen,expLen);
}

int main()
{
        int a[100],aLen,pStart,pLen;

        /* aLen == 1 */
        printf("Test 1\n");
        aLen = 1; a[0] = 1;
        findLongestPlateau(a,aLen,&pStart,&pLen);
        compare(pStart,pLen,0,1);

        /* plateau of length 1 */
        printf("Test 2\n");
        aLen = 4; a[0] = 0; a[1] = 1; a[2] = 0; a[3] = 1;
        findLongestPlateau(a,aLen,&pStart,&pLen);
        compare(pStart,pLen,0,1);

        /* plateau of length aLen */
        printf("Test 3\n");
        aLen = 4; a[0] = 1; a[1] = 1; a[2] = 1; a[3] = 1;
        findLongestPlateau(a,aLen,&pStart,&pLen);
        compare(pStart,pLen,0,4);

        /* longest plateau at front */
        printf("Test 4\n");
        aLen = 4; a[0] = 1; a[1] = 1; a[2] = 3; a[3] = 3;
```

```
        findLongestPlateau(a,aLen,&pStart,&pLen);
        compare(pStart,pLen,0,2);

        /* longest plateau at end */
        printf("Test 5\n");
        aLen = 4; a[0] = 1; a[1] = 2; a[2] = 3; a[3] = 3;
        findLongestPlateau(a,aLen,&pStart,&pLen);
        compare(pStart,pLen,2,2);

        /* two longest plateaus */
        printf("Test 6\n");
        aLen = 6; a[0] = 1; a[1] = 2; a[2] = 2; a[3] = 3; a[4] = 3; a[5] = 1;
        findLongestPlateau(a,aLen,&pStart,&pLen);
        compare(pStart,pLen,1,2);
}
```

Example: `removeLongestPlateau` driver and stub

Specification for the function under test:

```
/* Remove P, the longest plateau in a, shifting left all the elements
 * to the right of P and decreasing *aLen appropriately.
 *
 * Assumed: a has at least *aLen elements and *aLen > 0
 */
void removeLongestPlateau(int a[], int *aLen);
```

Test driver:

```
int stubStart,stubLen;

void findLongestPlateau(int a[],int aLen,int* pStart,int *pLen)
{
        *pStart = stubStart;
        *pLen = stubLen;
}

void load(int a[],int *aLen,int n)
{
        int i;
        for (i = 0; i < n; i++)
                a[i] = pow(10,i);
        *aLen = n;
}

void check(int a[],int aLen,int expSum)
{
        int i,actSum = 0;

        for (i = 0; i < aLen; i++)
                actSum += a[i];

        if (actSum != expSum)
                printf("Sum. Actual: %d Expected: %d\n",actSum,expSum);
}
```

```
int main()
{
        int a[100],aLen;

        /* short plateau */
        printf("Test 1\n");
        load(a,&aLen,4);
        stubStart = 0;
        stubLen = 1;
        removeLongestPlateau(a,&aLen);
        check(a,aLen,1110);

        /* medium-sized plateau */
        printf("Test 2\n");
        load(a,&aLen,4);
        stubStart = 1;
        stubLen = 2;
        removeLongestPlateau(a,&aLen);
        check(a,aLen,1001);

        /* long plateau */
        printf("Test 3\n");
        load(a,&aLen,4);
        stubStart = 0;
        stubLen = 4;
        removeLongestPlateau(a,&aLen);
        check(a,aLen,0);
}
```