Data Structures in C++

Chapter 5

Tim Budd

Oregon State University
Corvallis, Oregon
USA

Outline – Chapter 5

Confidence Building Measures

- Program Proving
- Invariants
- Program Testing

The Role of Program Proving

Program proofs are techniques used, not for the computer, but for the sake of the programmer and other members of a development teams.

Often given as part of a code walk-through.

Just one form of confidence building measure.

Invariants

An *invariant* is simply a statement about what is true when execution reaches a certain point in a program. We can draft an argument concerning the correct functioning of a program around a sequence of invariants.

Example Program with Invariants

```
double minimum (double values [ ], unsigned n)
{
    // make sure there is at least one element
    assert(n > 1);
    double minValue = values [0];
        // inv 1: minValue is the minimum
        // value found in range 0 .. 0
    for (unsigned int i = 1; i < n; i++) {
        // inv 2: minValue is the minimum
        // value found in range 0 .. i-1
        if (values[i] < minValue)
            minValue = values[i];
        // inv 3: minValue is the minimum
        // value found in range 0 .. i
        }
    // inv 4: minValue is the minimum value
    // found in the range 0 .. n-1
    return minValue;
}
```

A proof using Invariants

To make a proof using invariants, must give a bunch of little arguments, one for each possible path from one invariant to the next.

Each argument is a conditional – *if* the first invariant is true, *then* the second must also be true.

Start by showing the first invariant must always be true.

```
double minimum (double values [ ], int n)
{
    // make sure there is at least one element
    assert(n > 1);
    double minValue = values [0];
        // inv 1: minValue is the minimum
        // value in the range 0 .. 0
```

Invariant 1 to Invariant 2

```
...
double minValue = values [0];
    // inv 1: minValue is the minimum
    // in the range 0 .. 0
for (unsigned int i = 1; i < n; i++) {
    // inv 2: minValue is the minimum
    // in the range 0 .. i-1
...
```

Invariant 3 to Invariant 3

```
...
    // inv 2: minValue is the minimum
    // in the range 0 .. i-1
    if (values[i] < minValue)
        minValue = values[i];
    // inv 3: minValue is the minimum
    // in the range 0 .. i
...
```

Invariant 3 back to Invariant 2

```
for (unsigned int i = 1; i < n; i++) {
    // inv 2: minValue is the minimum
    // in the range 0 .. i-1
    if (values[i] < minValue)
        minValue = values[i];
    // inv 3: minValue is the minimum
    // in the range 0 .. i
    }
```

Invariant 3 to Invariant 4

```
for (unsigned int i = 1; i < n; i++) {
        ...
    // inv 3: minValue is the minimum
    // in the range 0 .. i
    }
// inv 4: minValue is the minimum value
// in the range 0 .. n-1
```

Do not forget, Invariant 1 to Invariant 4

```
...
    // inv 1: minValue is the minimum
    // in the range 0 .. 0

for (unsigned int i = 1; i < n; i++) {
    ...
    }

    // inv 4: minValue is the minimum
    // in the range 0 .. n-1
```

Invariants and Mathematical Induction

The use of invariants is put on a solid mathematical foundation by relating to mathematical induction.

The induction is the *number of times* the loop executes.

Argue that if we have executed n times, and the invariant is still true, and if we execute an addition $n+1$ time, that the invariant will still be true. So no matter how many times we loop, the invariants must still be true.

Asserting Outcome is Correct

An invariant must assert the outcome is correct.

May involve ideas not expressly presented in the code.

```
int isPrime (unsigned int n)
{
  for (unsigned int i = 2; i * i < n; i++) {
        // inv 1: n no factors in 2 to i-1
     if (0 == n % i) {
           // inv 2: i divides n, not prime
        return 0;
        }
        // inv 3: n no factors in 2 to i
     }
     // inv 4: n has no factors in 2 to
     // ceiling(sqrt(n)), therefore must
     // be prime
  return 1;
}
```

Making Progress Towards an Objective

```
unsigned int binarySearch (double v [ ], unsigned int n,
double value)
    // search for value in ordered array of data
    // return index of value, or index of
    // next smaller value if not in collection
{
    unsigned int low = 0;
    unsigned int high = n;

    while (low < high) {
        // inv: data[0 .. low-1] less than value
        // data[high .. max] greater than or equal
        // to value
        unsigned mid = (low + high) / 2;
        if (data[mid] < value)
            low = mid + 1;
        else
            high = mid;
        }
    // inv: data[0..low-1] less than value
    // and value less than or equal to data[low+1]
    return low;
}
```

Simulate binary search

Simulate binary search on the values

$$2\ 4\ 5\ 7\ 9\ 12\ 14\ 37\ 96$$

Loops May have their Own Agenda

```
void bubbleSort (double & v[], unsigned int n)
{
    for (unsigned int i = n - 1; i > 0; i--) {
        // inv: elements i+1 to n-1 are correct
        for (unsigned int j = 0; j < i; j++) {
            // inv: v[j] is largest in (0..j)
            if (v[j+1] < v[j]) { // if out of order
                double temp = v[j]; // then swap
                v[j] = v[j + 1];
                v[j + 1] = temp;
                }
            // inv: v[j+1] is largest in (0..j+1)
            }
        // inv: v[i] holds largest in (0..i)
        // inv: therefore, elements i to n-1 are ordered
        }
    // inv: elements indexed 0 to n-1 are ordered
}
```

Invariants and Unnamed Quantities

```
unsigned int gcd (unsigned int n, unsigned int m)
    // compute the greatest common divisor
    // of two positive integer values
{
    assert (n > 0 && m > 0);

    while (m != n) {
        if (n > m)
            n = n - m;
            // inv: gcd of n and m
            // has not been altered
        else
            m = m - n;
            // inv: gcd of n and m
            // has not been altered
        }
        // n equal to m,
        // so n is divisor of both
    return n;
}
```

Invariants and Function Calls

When functions are involved, the argument is again conditional – *if* the called routine is correct, then we can argument that the current routine is correct.

```
void printPrimes (unsigned int n)
    // print numbers between 2 and n
    // indicating which are prime
{
    for (unsigned int i = 2; i <= n; i++) {
        if (isPrime (i))
            cout << i << " is prime\n";
        else
            cout << i << " is not prime\n";
        }
}
```

Recursive Algorithms

The analysis of recursive algorithms breaks into cases:

- Base case argument – same as any other function
- Recursive case argument – conditional, *if* the recursive call works as advertised, then argue that the rest of the program must work correctly.

```
void printUnsigned (unsigned int val)
{
    if (val < 10)
        printChar (digitChar(val));
    else {
            // print high order part
        printUnsigned (val / 10);
            // print last character
        printChar (digitChar(val % 10));
        }
}
```

Another Recursive Algorithm

```
double power (double base, unsigned int n)
    // return the value base
    // raised to the integer n value
{
    if (n == 0)
        return 1.0;
    else if (even(n))
            // base ^ n is same as
            // (base ^ 2) ^ (n / 2) for even n
        return power (base * base, n / 2);
    else
            // for odd n base ^ n is same as
            // base * (base ^ 2) ^ (n / 2)
        return power (base * base, n / 2) * base;
}
```

Program Testing

An alternative (and complementary) way of increasing confidence in correctness. Can be performed on many levels

- Individual function (or method)
- Class
- Complete application

Drivers and Stubs

Testing bits of code in isolation frequently requires writing temporary harness code.

- Drivers to load test cases, run program, print or verify results,
- Stubs to simulate called routines

Goals for Testing

- Make sure every statement in the function is exercised by at least one test value.
- If there is a minimal legal input value, such as an empty array or a smallest integer value, use this as one of your test cases.
- If the function (or program) has both legal and illegal inputs, a set of test cases should include both clearly legal values and clearly illegal values, as well as values that are "barely" legal and "barely" not legal.
- If the program involves loops that can exercise a variable number of iterations, try to develop a test case in which the loop executes zero times.