CS 101:
Computer Programming and Utilization

Jan-Apr 2016

Bernard Menezes
(cs101@cse.iitb.ac.in)

Lecture 7: Loop Invariants

About These Slides

- Based on Chapter 7 of the book

  An Introduction to Programming Through C++

  by Abhiram Ranade (Tata McGraw Hill, 2014)

- Original slides by Abhiram Ranade
  - First update by Varsha Apte
  - Second update by Uday Khedker

Reasoning About Loops

- Reasoning about loop-free programs is easy
- Reasoning about loops is tricky
- How do we know whether what we have written is correct?

Euclid's Algorithm For GCD

- Greatest Common Divisor (GCD) of positive integers m, n :

  largest positive integer p that divides both m, n
- Standard method: factorize m,n and multiply common factors
- Euclid’s algorithm (2300 years old!) is different and much faster
- A program based on Euclid’s method will be much faster than program based on factoring

Euclid's Algorithm

Basic Observation: If d divides both m, n, then d divides m-n also, assuming m > n

Proof: m=ad, n=bd, so m-n=(a-b)d

Converse is also true: If d divides m-n and n, then it divides m too

m, n, m-n have the same common divisors

The largest divisor of m,n is also the largest divisor of m-n,n

Observation: Instead of finding GCD(m,n), we might as well find GCD(n, m-n)

Example

GCD(3977, 943)

=GCD(3977-943,943) = GCD(3034,943)

=GCD(3034-943,943) = GCD(2091,943)

=GCD(2091-943,943) = GCD(1148,943)

=GCD(1148-943,943) = GCD(205, 943)

- We should realize at this point that 205 is just 3977 % 943 (repeated subtraction is division)
- So we could have got to this point just in one shot by writing GCD(3977,943) = GCD(3977 % 943, 943)

Example

Should we guess that GCD(m,n) = GCD(m%n, n)?

This is not true if m%n = 0, since we have defined GCD only for positive integers. But we can save the situation, as Euclid did

Euclid's theorem: If m>n>0 are positive integers, then if n divides m then GCD(m,n) = n. Otherwise GCD(m,n) = GCD(m%n, n)

Example Continued

GCD(3977,943)

= GCD(3977 % 943, 943)

= GCD(205, 943) = GCD(205, 943%205)

= GCD(205,123) = GCD(205%123,123)

= GCD(82, 123) = GCD(82, 123%82)

= GCD(82, 41)

= 41 because 41 divides 82

GCD Algorithm to Program

input: values M, N which are stored in variables m, n.

iteration : Either discover the GCD of M, N, or find smaller numbers whose GCD is same as GCD of M, N

Details of an iteration:

- At the beginning we have numbers stored in m, n, whose GCD is the same as GCD(M,N).
- If n divides m, then we declare n to be the GCD.
- If n does not divide m, then we know that GCD(M,N) = GCD(n, m%n)
- So we have smaller numbers n, m%n, whose GCD is same as GCD(M,N)

Program For GCD

```
main_program{
    int m, n; cin >> m >> n;
    while(m % n != 0){
        int nextm = n;
        int nextn = m % n;
        m = nextm;
        n = nextn;
    }
    cout << n << endl;
}
// To store n, m%n into m,n, we cannot
// just write m=n; n=m%n;
// Can you say why?  Hint: take an example!
```

Remark

- We have defined variables nextm, nextn for clarity
- We could have done the assignment with just one variable as follows
  - int r = m%n; m = n; n = r;
- It should be intuitively clear that in writing the program, we have followed the idea from Euclid’s theorem. However, having written the program, we should check this again

Invariants

Let M, N be the values typed in by the user into variables m, n

We can make the following claim

Just before and just after every iteration,

GCD(m,n) = GCD(M,N)

The values m and n change, M and N do not

Loop Invariant: *A property (describing a pattern of values of variables) which does not change due to the loop iteration.*

Loop Invariant for GCD

```
main_program{
    int m, n; cin >> m >> n; // Assume M, N
    // Invariant: GCD(m,n) = GCD(M,N)
    // because m=M and n=N
    while(m % n != 0){
        int nextm = n;           // the invariant may
        int nextn = m % n;       // not hold after
        m = nextm;               // these statements
        n = nextn;
    // Invariant: GCD(m,n) = GCD(M,N)
    // inspite of the fact that m, n have changed
    }
    cout << n << endl;
}
```

Loop Invariant for GCD

GCD(3977,943) m=M=3977, n=N=943

= GCD(3977 % 943, 943)

= GCD(205, 943) = GCD(205, 943%205) m=943, n=205

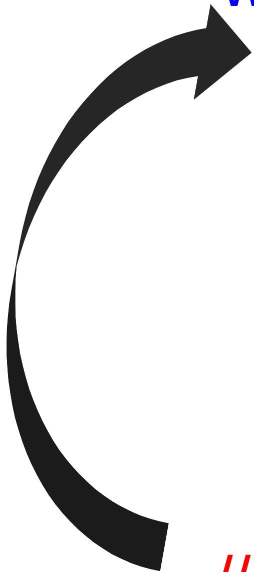
= GCD(205,123) = GCD(205%123,123) m=205, n=123

= GCD(82, 123) = GCD(82, 123%82) m=123, n=182

= GCD(82, 41) m=82, n=41

= 41 because 41 divides 82

The Intuition Behind Loop Invariant



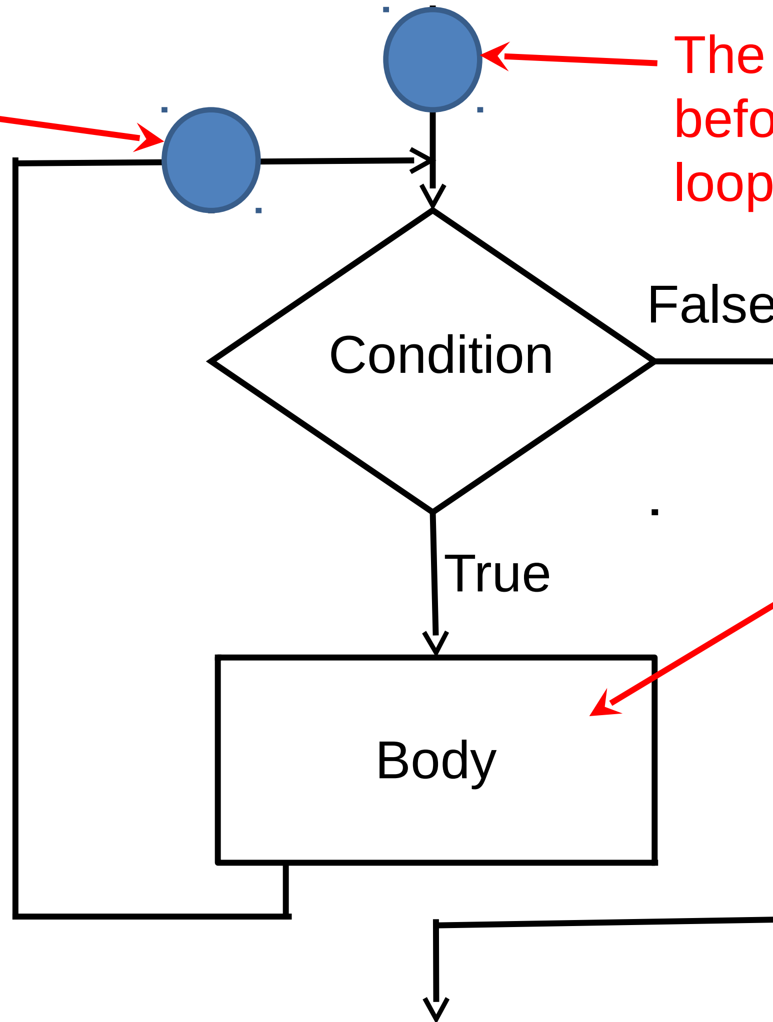
```
// Invariant holds here
while(m % n != 0) {
    // Invariant holds at the start of the loop
    // The loop body may disturb the invariant
    // by changing the values of variables
    // but the invariant must hold at the start
    // of the next iteration
    // Hence invariant must be restored
// Invariant must hold here too
}
```

The Intuition Behind Loop Invariant

Previous statement in the program

The invariant holds here before the execution every subsequent iteration

The invariant holds here before the execution of the loop begins



The loop body may disturb the invariant but it must be restored before beginning the execution of the next iteration

Next statement in the Program

Proof of the Invariant in GCD Program

- Clearly, the invariant is true just before the first iteration
- In any iteration, the new values assigned to m,n are as per Euclid’s theorem, and hence the invariant must be true at the end, and hence at the beginning of the next iteration
- And the above argument applies to all iterations

Invariants In Simple Programs

Correctness of very simple loops may be obvious, and it may not be necessary to write invariants etc

However, invariants can be written, and they still make our intent more explicit

Example: Cube table program

Next

Invariants In The Cube Table Program

```
for(int i=1; i<=100; i++)
    cout << i <<‘ ‘<<i*i*i<<endl;
```

Invariant: Cubes until i-1 have been printed

True for every iteration!

For programs so simple, writing invariants seems to make simple things unnecessarily complex. But invariants are very useful when programs are themselves complex/clever

Invariant in Max Finding Program

```
main_program {
  int n;
  int max = 0;
  // Invariant: max is the largest number seen so far
    while (true) {
          cin >> n;
           if(n < 0) break;                      // end of input
           else if (n > max) max = n;      // max becomes n
           else ;                                          // do nothing
 // Invariant: max is the largest number seen so far
   }
  cout << "Largest Number is" << max << endl;
}
```

Invariants in Mark Averaging Program

```
main_program{
  float nextmark, sum = 0; int count = 0;
// Invariants: the values in variables sum and count
// are the sum and count of relevant marks seen so far
    while (true){
          cin >> nextmark;
          if(nextmark > 100) continue;
          if(nextmark < 0) break;
          sum += nextmark; count++;
// Invariants: the values in variables sum and count
// are the sum and count of relevant marks seen so far
       }
  cout << sum/count << endl;
}
```

What is the Loop Invariant Here?

```
unsignd int x;
int y = 0;
while (x != y)
      y++;
```

- What is the loop invariant?

  x >= y

- Is x == y after the loop terminates?

  We will shortly prove it

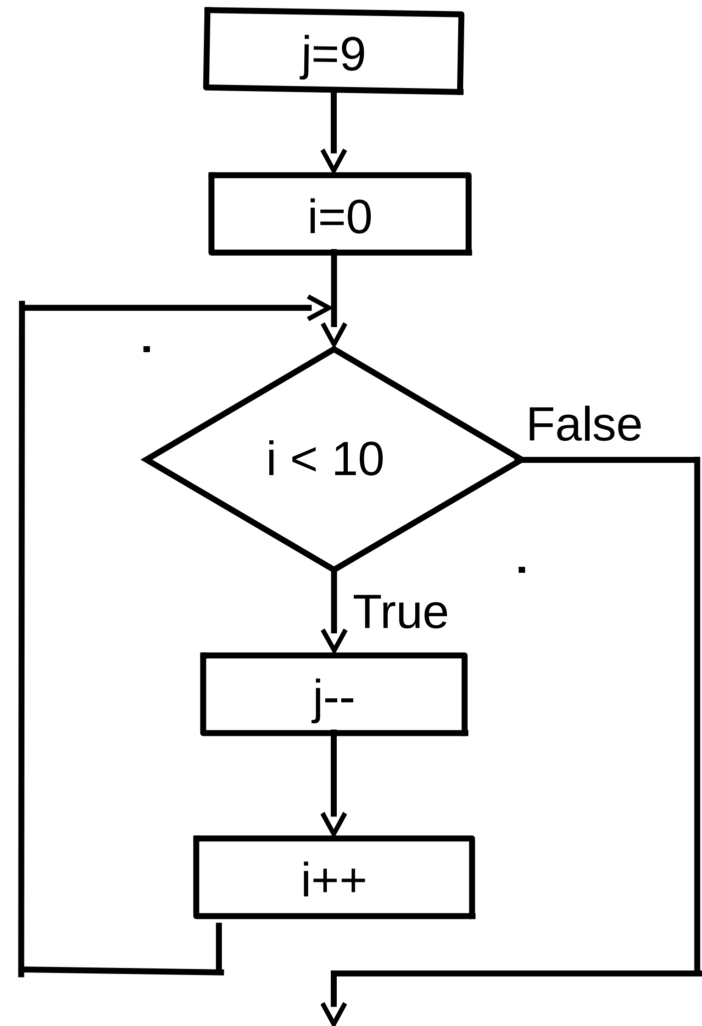
What is the Loop Invariant Here?

```
int j=9;
for (int i=0; i<10; i++)
    j--;
```

- 0 <= i < 10 — NO
- 0 <= i <=10 — Yes, but not precise (misses j) (must also hold before condition becomes false and loop ends)
- i+j = 9 — Yes, but not precise
- i+j=9, 0<=i<10 — Yes, most precise

Is i+j=9 a Loop Invariant Here?

Visit to the condition	Value of i	Value of j	Loop body executed?
1	0	9	Yes
2	1	8	Yes
3	2	7	Yes
4	3	6	Yes
5	4	5	Yes
6	5	4	Yes
7	6	3	Yes
8	7	2	Yes
9	8	1	Yes
10	9	0	No



Every Loop Invariant May Not be Useful

```
int j=9;
for (int i=0; i<10; i++)
    j--;
```

- Some loop invariants

  $i >= 0$

  $j <= 9$

  $i+j <= 100$

  $-1000 <= i <= 100$

- Usefulness: The invariant should be as "close" to the actual values as possible
- May not be possible always: "Undecidable"

Why Discover Loop Invariants?

- A precise loop Invariant represents the essential behaviour of a loop
  - Allow relating the values before the loop to the values after the loop
  - without executing the loop
- For understanding a program, loops in the program can be represented by their loop invariants
- Finding good loop invariants is critical for proving correctness with respect to a chosen property (also known as verification)

Use Of The Invariant

Proving correctness of GCD Program

- Using the invariant we can show the algorithm will give the correct answer, if it terminates
- If the algorithm terminates, `m%n` must have been 0
- But in this case GCD(`m,n`) = n, which is what the algorithm prints
- But this is correct because by the invariant, GCD(`m,n`) = GCD(M,N) which is what we wanted

Proof of Termination

The only thing that remains is to show termination

- The value of the variable n must decrease in each iteration. (because, nextn = m%n which must be smaller than n),
- But n must always be a positive integer in every iteration: (because we enter an iteration only if m%n != 0, and then set nextn = m%n)
- Thus n cannot decrease indefinitely, it cannot go below 1
- n starts with the value N, thus the algorithm must terminate after at most N iterations

This argument is called a potential function argument. You have to creatively choose the potential

Proving x == y Using the Loop Invariant

```
unsignd int x;
int y = 0;
while (x != y)
      y++;
```

Loop invariant: x>=y (or y<=x)

- y is initially 0
- y keeps increasing
- Since y<=x, it can at most be equal to x
- But that is the termination condition of the loop

Remarks

- The remarks in Chapter 3 about making a plan and identifying the general pattern of actions apply here also. The actions in the loop may include preparing for the next iteration, e.g. incrementing a variable, or setting new values of `m,n` as in the GCD program
- Think about the invariants and the potential. This is a good way to cross-check that your loop is doing the right thing, and that it will terminate. Write these down explicitly when the algorithm is even slightly clever