CS 101: Computer Programming and Utilization

Jan-Apr 2017

Sunita Sarawagi
(cs101@cse.iitb.ac.in)

Lecture 7: General Loops

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

The Need of a More General Loop

Read marks of students from the keyboard and print the average

- Number of students not given explicitly
- If a negative number is entered as marks, then it is a signal that all marks have been entered

  Examples
  - Input: 98 96 -1, Output: 97
  - Input: 90 80 70 60 -1, Output: 75
- The repeat statement repeats a fixed number of times. Not useful
- We need a more general statement

  while, do while, or for

Outline

- The while statement
  - Some simple examples
  - Mark averaging
- The break statement
- The continue statement
- The do while statement
- The for statement

The WHILE Statement

```
while (condition)
        body

next_statement
```

1. Evaluate the condition

   If true, execute body. body can be a single statement or a block, in which case all the statements in the block will be executed

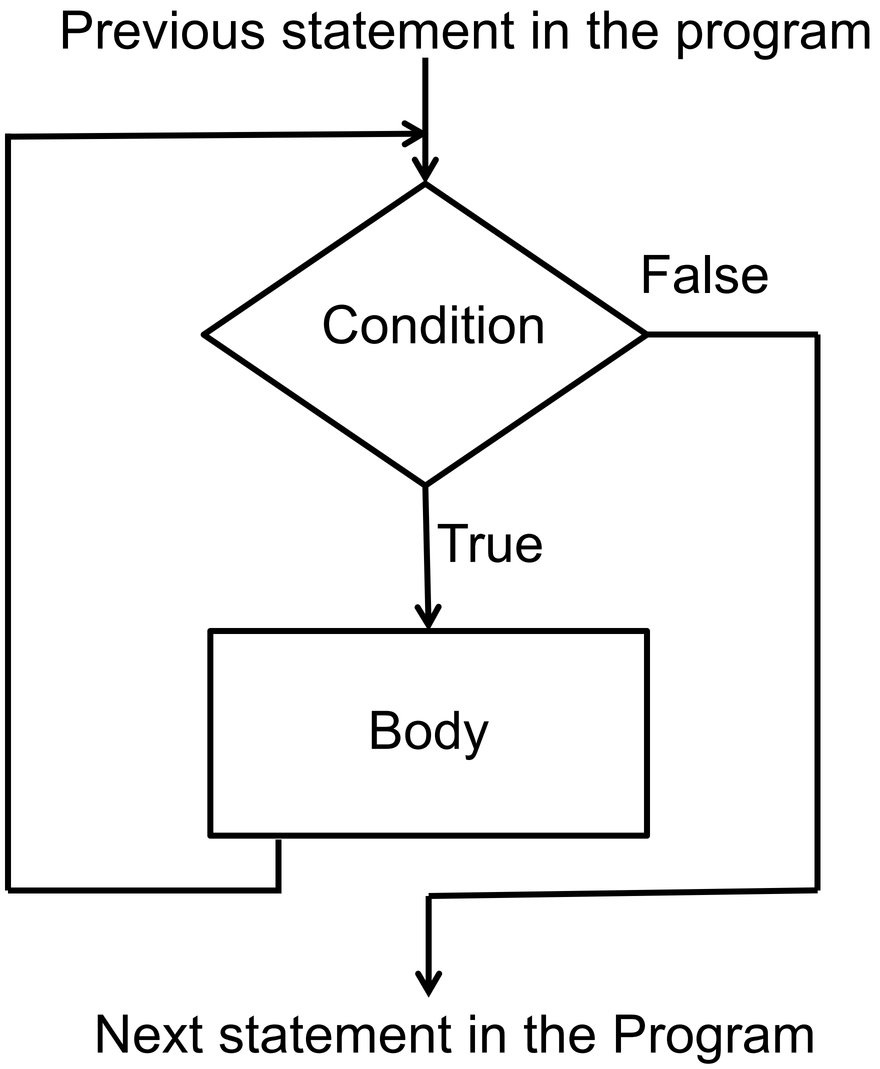
2. Go back and execute from step 1
3. If false, execution of while statement ends and control goes to the next statement

The WHILE Statement

```
while (condition)
      body
```

- The condition must eventually become false, otherwise the program will never halt. Not halting is not acceptable
- If the condition is true originally, then the value of some variable used in condition must change in the execution of body, so that eventually condition becomes false
- Each execution of the body = iteration

WHILE Statement Flowchart


Previous statement in the program
Condition
False
True
Body
Next statement in the Program

A Program That Does Not Halt

```
main_program{
    int x=10;
    while(x > 0){
        cout << “Iterating” << endl;
    }
}
// Will endlessly keep printing
// Not a good program
```

A Program That Does Halt

```
main_program{
      int x=3;
      while(x > 0){
            cout << “Iterating” << endl;
            x--; // Same as x = x – 1;
      }
}
// Will print “Iterating.” 3 times
// Good program (if that is what
// you want)!
```

Explanation

```
main_program{
    int x=3;
    while(x > 0){
    cout << “Iterating” <<
endl;
        x--;
    }
}
```

- First x is assigned the value 3
- Condition x > 0 is TRUE
- So body is executed (prints Iterating)
- AFTER x-- is executed, the value of x is 2

Explanation

```
main_program{
     int x=3;
     while(x > 0){
        cout << “Iterating” <<
endl;
          x--;
     }
}
```

- Again the condition is evaluated. For x with value 2, condition is still TRUE
- So execute this
  - print iterating
- Decrement x
- Value now is 1

Explanation

```
main_program{
    int x=3;
    while(x > 0){
        cout << Iterating <<
endl;
        x--;
    }
}
```

- Again the condition is evaluated. For x with value 1, condition is still TRUE
- So execute this
  – print iterating
- Decrement x
- Value now is 0

Explanation

```
main_program{
    int x=3;
    while(x > 0){
        cout << Iterating <<
endl;
        x--;
    }
}
```

- Again the condition is evaluated. For x with value 0, condition is still FALSE
- So control goes outside the body of the loop
- Program exits

WHILE vs. REPEAT

Anything you can do using repeat can be done using while (but not vice-versa)

```
repeat(n){ any code }
```

Equivalent to

```
int i=n;
while(i>0){i--; any code}
```

This is a simplistic explanation

See file include/simplecpp for a more precise explanation

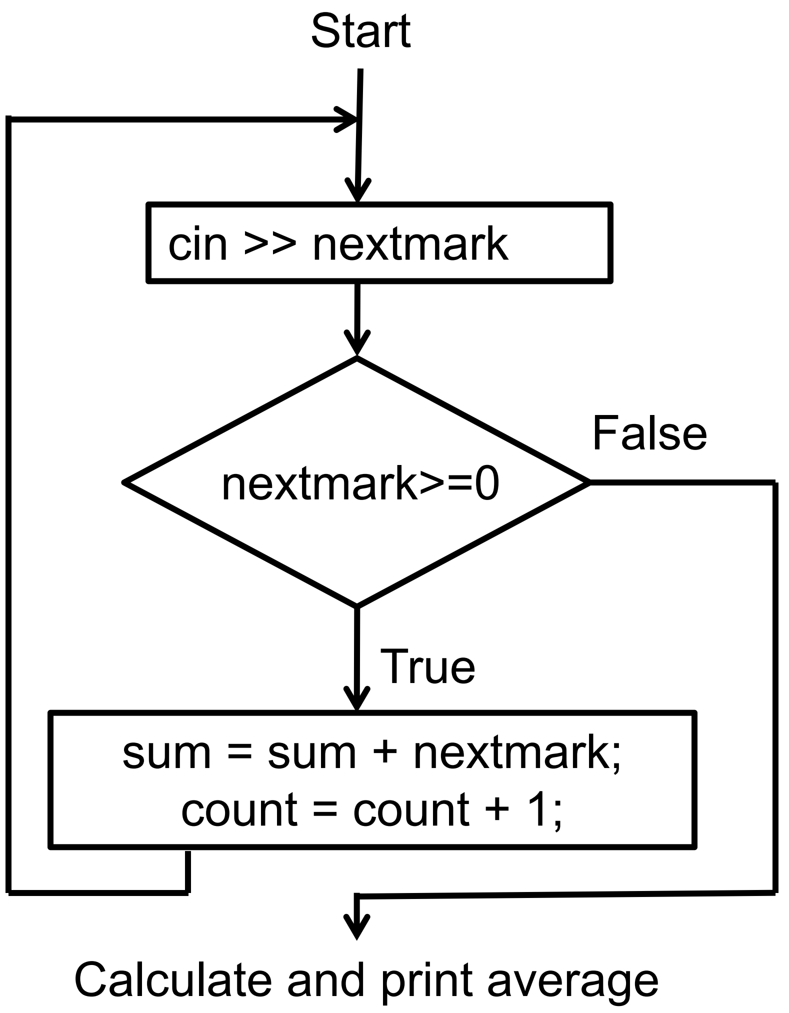
Mark Averaging

Natural strategy

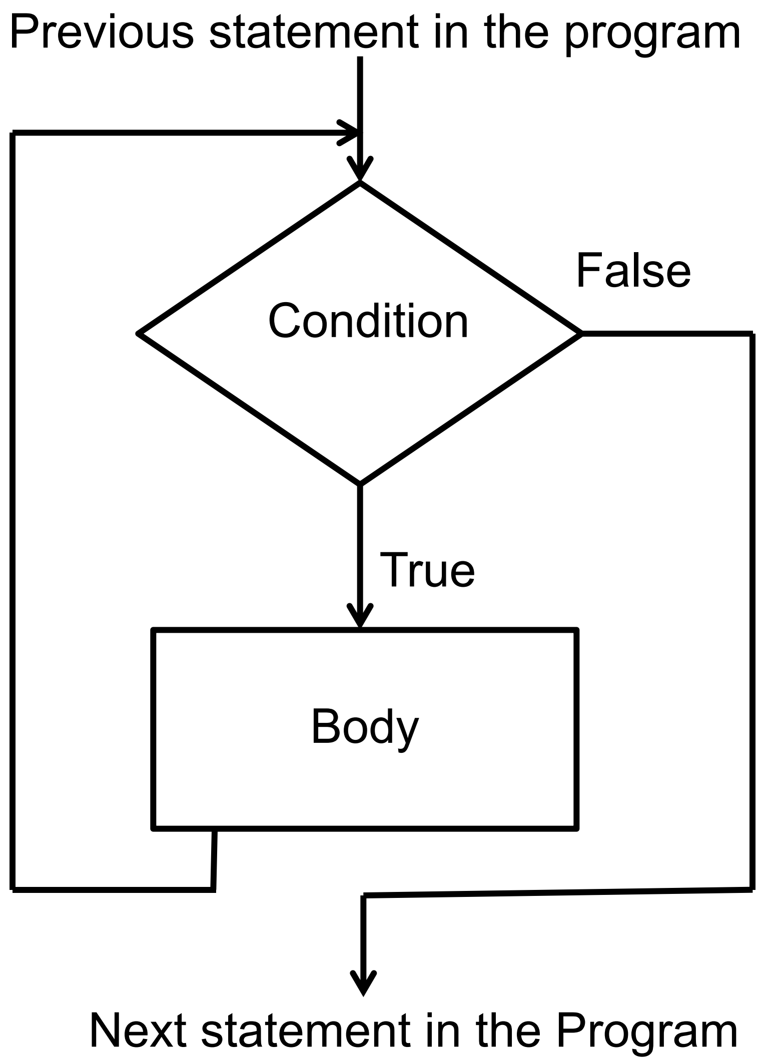
1. Read the next value
2. If it is negative, then go to step 5, if it is >= 0, continue to step 3
3. Add the value read to the sum of values read so far, Add 1 to the count of values read so far.
4. Go to step 1
5. Print sum/count

A bit tricky to implement using while

Flowchart Of Mark Averaging vs. Flowchart Of While



Flowchart of mark averaging

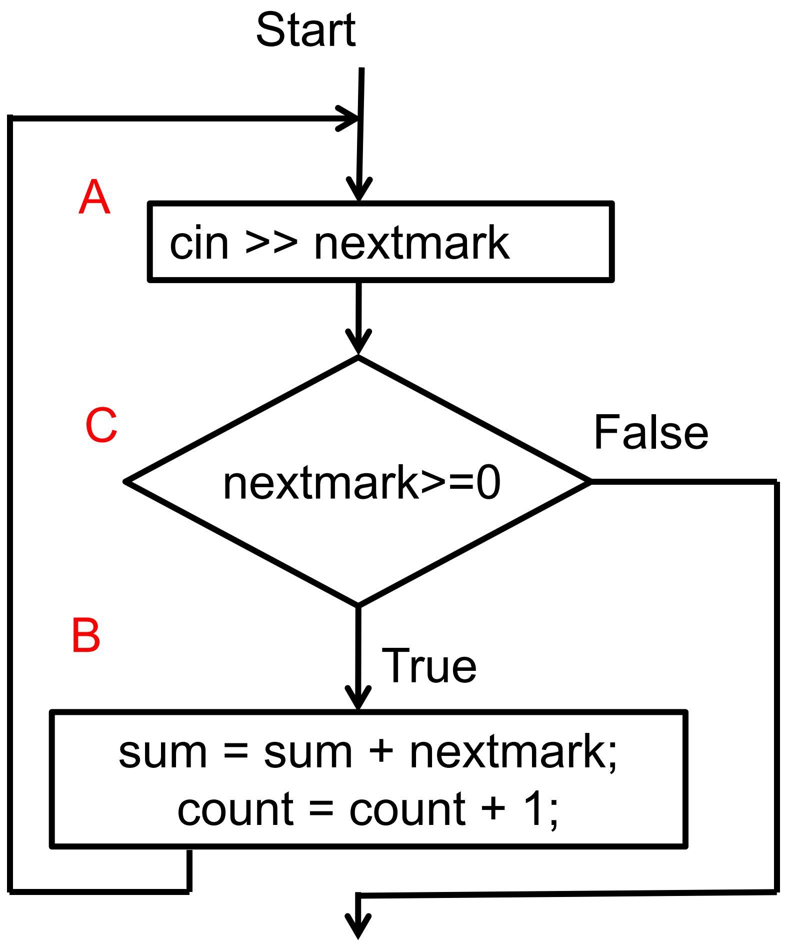

Previous statement in the program
Condition
False
True
Body
Next statement in the Program

Flowchart of WHILE

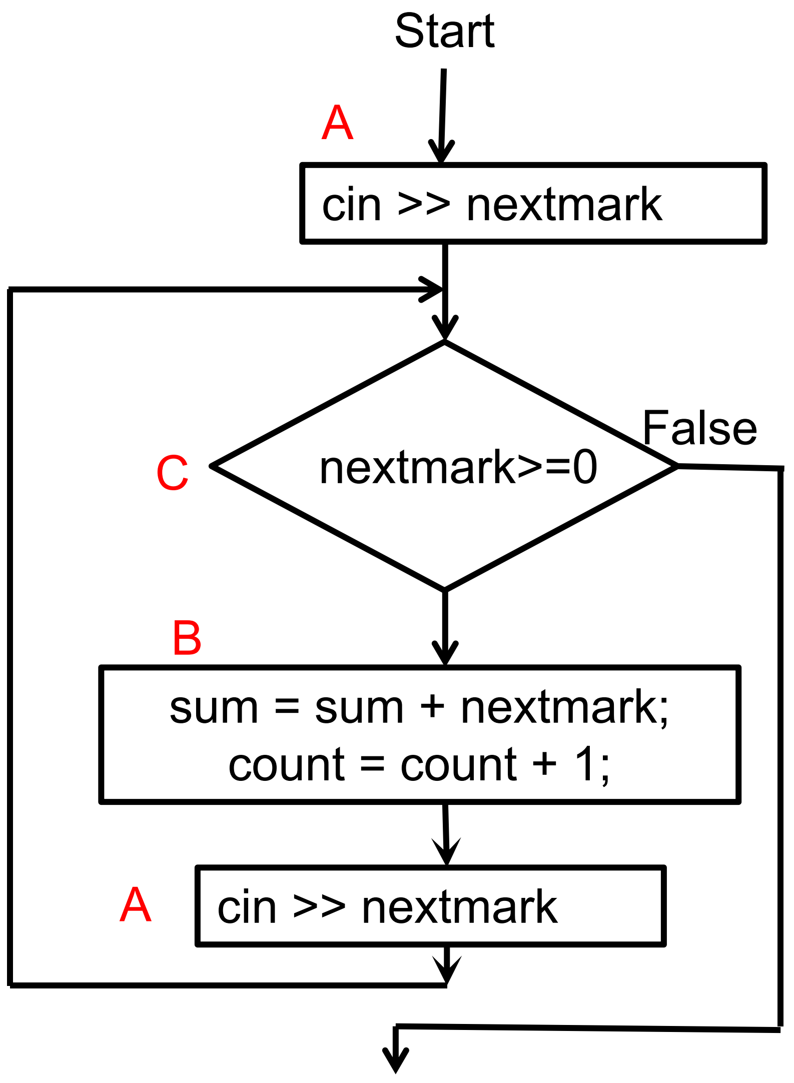
Flowchart Of Mark Averaging vs. Flowchart Of WHILE

- In the flowchart of mark averaging, the first statement to be repeated is not the condition check
- In the flowchart of while, the first statement to be repeated, is the condition check
- So we cannot easily express mark averaging using while

Flowchart Of Mark Averaging vs. Flowchart of WHILE



Original



Modified

A Different Flowchart For Mark Averaging

- Let's label the statements as A (input), C (condition), and B (accumulation)

- The desired sequence of computation is

  A-C-B   A-C-B   A-C-B ... A-C

- We just rewrite it is

  A   C-B-A   C-B-A   C-B-A ... C

- Thus we take input outside of the loop once and then at the bottom of the loop body

Program

```
main_program{
  float nextmark, sum = 0;
  int count = 0;
  cin >> nextmark;                    // A
  while(nextmark >= 0){
      sum += nextmark; count++;
      cin >> nextmark;                // copy of A!!
  }
  cout << sum/count << endl;
}
```

Remarks

- Often, we naturally think of flowcharts in which the repetition does not begin with a condition check. In such cases we must make a copy of the code, as we did in our example
- Also remember that the condition at the beginning of the while must say under what conditions we should enter the loop, not when we should get out of the loop. Write the condition accordingly
- Note that the condition can be specified as true, which is always true. This may seem puzzling, since it appears that the loop will never terminate. But this will be useful soon..

Nested WHILE Statements

We can put one while statement inside another The execution is as you might expect. Example:

```
int i=3;
while(i > 0) {
    i--;
    int j=5;
    while(j > 0){
        j--;
        cout << “A”;
    }
    cout << endl;
}
```

What do you think this will print?

The BREAK Statement

- The break keyword is a statement by itself
- When it is encountered in execution, the execution of the innermost while statement which contains it is terminated, and the execution continues from the next statement following the while statement

Example of BREAK

```
main_program{
  float nextmark, sum = 0;
  int count = 0;
  while(true){
         cin >> nextmark;
         if(nextmark < 0)
                    break;
         sum += nextmark;
         count++;
         }
 cout << sum/count << endl;
}
```

If break is executed, control goes here, out of the loop

Explanation

- In our mark averaging program, we did not want to check the condition at the beginning of the repeated portion
- The break statement allows us just that!
- So we have specified the loop condition as true, but have put a break inside
- The statements in the loop will repeatedly execute; however when a negative number is read, the loop will be exited immediately, without even finishing the current iteration
- The break statement is of course useful in general

The CONTINUE Statement

- continue is another single word statement
- If it is encountered in execution, the control directly goes to the beginning of the loop for the next iteration, skipping the statements from the continue statement to the end of the loop body

Example

Mark averaging with an additional condition :

- if a number > 100 is read, discard it (say because marks can only be at most 100) and continue with the next number. As before stop and print the average only when a negative number is read

Code For New Mark Averaging

```
main_program{
  float nextmark, sum = 0;
  int count = 0;
  while (true){
         cin >> nextmark;
         if(nextmark > 100)
         continue;
         if(nextmark < 0)
              break;
         sum += nextmark;
         count++;
  }
  cout << sum/count << endl;
}
```

If executed, the control goes back to condition evaluation

The DO-WHILE Statement

Not very common

Discussed in the book

The FOR Statement: Motivation

- Example: Write a program to print a table of cubes of numbers from 1 to 100

  ```
  nt i = 1;
  repeat(100){
   cout << i <<‘ ‘<< i*i*i << endl;
   i++;
  }
  ```

- This idiom: do something for every number between x and y occurs very commonly
- The for statement makes it easy to express this idiom, as follows:

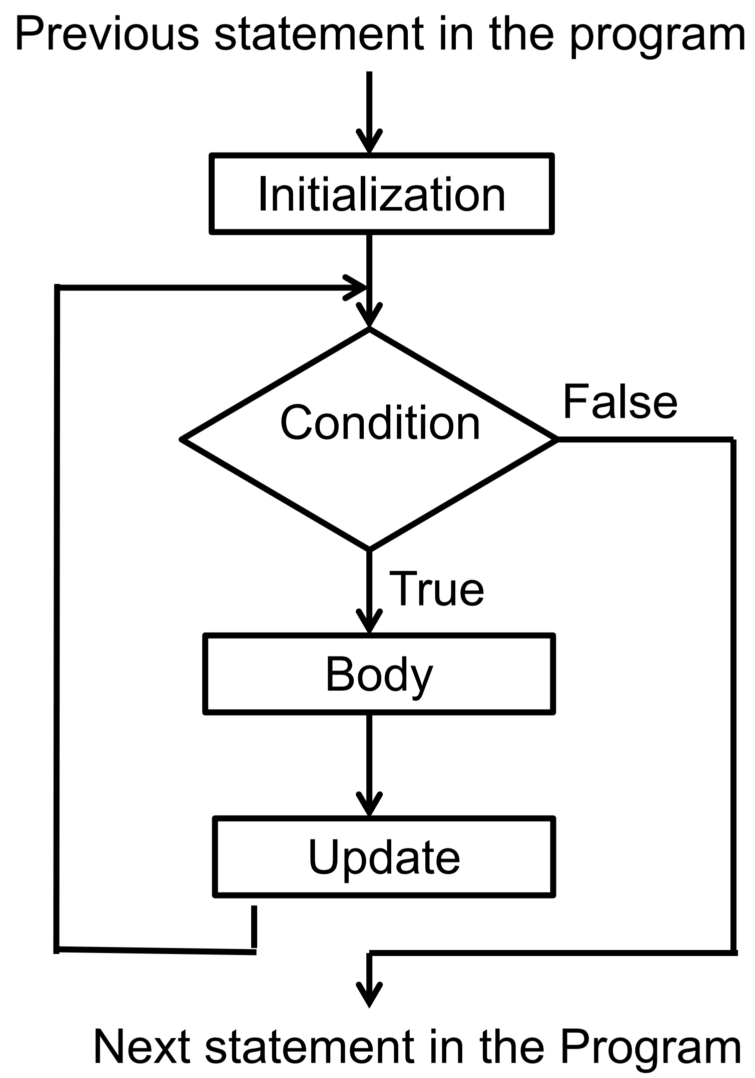
  ```
  for(int i=1; i<= 100; i++)
     cout << i <<‘ ‘<< i*i*i << endl;
  ```

The FOR Statement

for(initialization; condition; update)
  body

- initialization, update : Typically assignments (without semi-colon)
- condition : boolean expression
- Before the first iteration of the loop the initialization is executed
- Within each iteration the condition is first tested. If it fails, the loop execution ends. If the condition succeeds, then the body is executed. After that the update is executed. Then the next iteration begins

Flowchart for FOR Statement


Previous statement in the program
Initialization
Condition
False
True
Body
Update
Next statement in the Program

Definition of Repeat

```
repeat(n)
```

is same as

```
for (int _iterator_i = 0, _iterator_limit = n;
     _iterator_i < _iterator_limit;
     _iterator_i ++)
```

Hence changing n in the loop will have no effect in the number of iterations

Determining whether a number is prime

```
main_program{
  int n; cin >> n;
  bool found = false;
  for(int i=2; i < n; i++){
    if(n % i == 0){
      found = true;
      break;
    }
  }
  if(found) cout << "Composite.\n";
  else cout << "Prime.\n";
}
```

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

Algorithm Our GCD 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

Termination and Correctness

- We wrote the program based on Euclid’s theorem, but are we sure that it
  - Terminates?
  - Gives the correct answer?
- For any program, it is essential to argue both these.
- This is done by defining
  - Invariants
  - “Potential”

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;
}
```

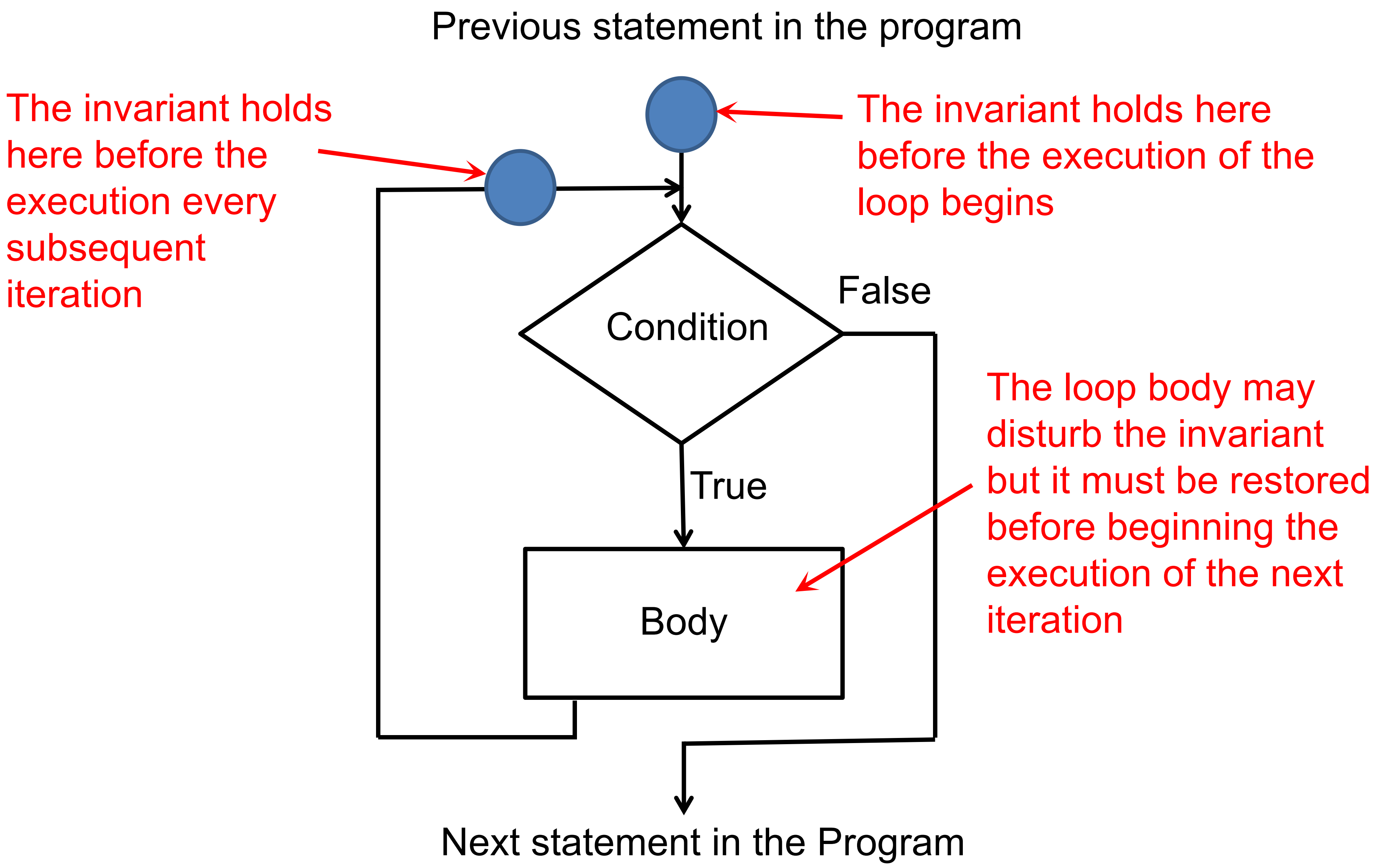
Loop Invariant for GCD

GCD(3977,943)	m=M=3977, n=N=943
= GCD(3977 % 943, 943)	
= GCD(205, 943) = GCD(205, 943%205)	m=205, n=943
= GCD(205,123) = GCD(205%123,123)	m=205, n=123
= GCD(82, 123) = GCD(82, 123%82)	m=205, n=123
= 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
Condition
False
True
Body
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

But the above argument applies to all iterations

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 (Analogy: Potential energy drops as system becomes less active) You have to creatively choose the potential

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!
- Potential: value of `i` : it must increase in every step, but cannot increase beyond 100.
- For programs so simple, writing invariants seems to make simple things unnecessarily complex. But invariants are very useful when programs are themselves complex/clever.

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

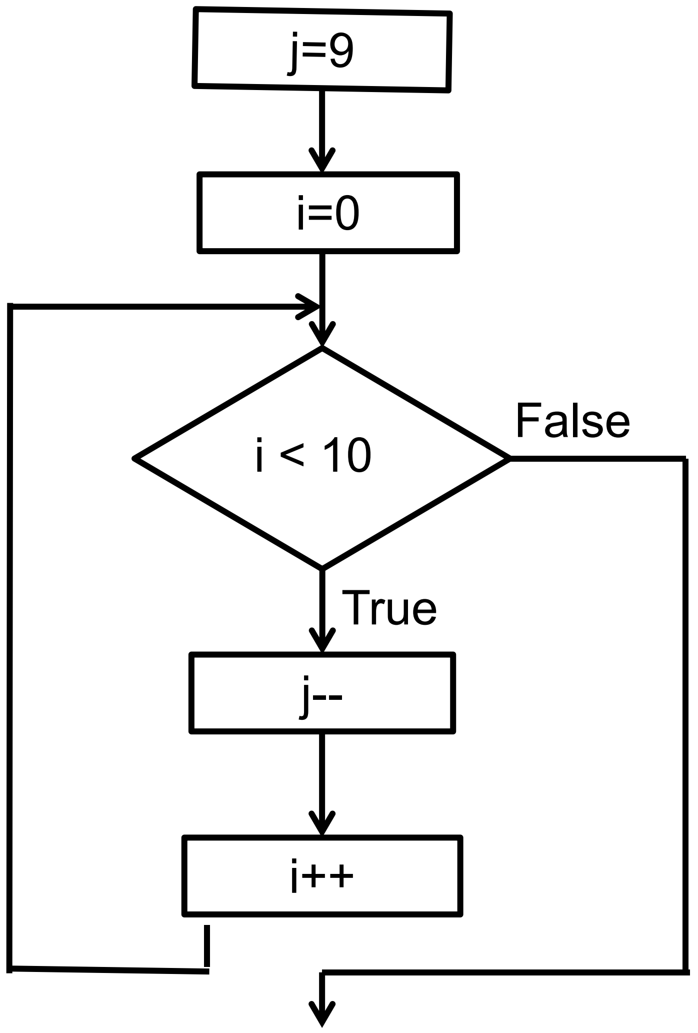
What is the Loop Invariant Here?

```
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



Remarks

- while, do while, for are the C++ statements that allow you to write loops
- repeat allows you to write a loop, but it is not a part of C++ It is a part of simplecpp; it was introduced because it is very easy to understand.
- Now that you know while, do while, for, you should stop using repeat

Remarks

An important issues in writing a loop is how to break out of the loop. You may not necessarily wish to break at the beginning of the repeated portion. In which case you can either duplicate code, or use `break`