Lecture 7: General Loops
(Chapter 7)
The Need for 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

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

Each execution of the body = **iteration**
The **WHILE** Statement

- 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.
WHILE Statement Flowchart

Previous statement in the Program

Condition

False

Body

True

Next statement in the Program
A Program That ??????

```c
main_program{
    int x=10;
    while(x > 0){
        cout << “Iterating” << endl;
    }
}
```
A Program that DOES halt

```c
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
And so on … till

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

- Again the condition is evaluated. For x with value 0, condition is now 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)

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

Equivalent to

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

OR

```plaintext
int i = 0;
while(i<n){any code; i++}
```
#include<simplecpp>

main_program{
    int n = 5;
    repeat(n) {
        cout << "in loop" << endl;
        n++;
    }
    while(n > 0){
        cout << "in 2nd loop" << endl;
        n++;
    }
}
Mark Averaging

Natural strategy

1. Read the next value
2. If it is negative, then go to step 5, if it is positive, continue to step 3
3. Add the value read to the sum of values read so far, Increment 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

- 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.

```c
int main() {
    int nextmark, sum = 0, count = 0;
    while (true) {
        cin >> nextmark;
        if (nextmark < 0) break;
        sum = sum + nextmark;
        count = count + 1;
        // Calculate and print average
    }
    // Next statement in the Program
    return 0;
}
```
Fixing the discrepancy

Original

```cpp
cin >> nextmark
nextmark >= 0
sum = sum + nextmark;
count = count + 1;
```

Modified

```cpp
cin >> nextmark
nextmark >= 0
sum = sum + nextmark;
count = count + 1;
```
main_program{
    float nextmark, sum = 0;
    int count = 0;
    cin >> nextmark;
    while(nextmark >= 0) {
        sum += nextmark; count++;
        cin >> nextmark;
    }
    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:

```c
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.
Nested WHILE Statements

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

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

What do you think this will print?
Back to Mark Averaging

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

```c++
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

  ```cpp
  int 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:

  ```cpp
  for(int i=1; i<= 100; i++)
    cout << i << ' ' << i*i*i << endl;
  ```
The **FOR** Statement

```plaintext
for(initialization; condition; update)
  body
  • initialization, update : Typically assignments
  • 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

Body

Update

Next statement in the Program
int i;
for (i=0; i < 0; i++)
    cout << "in loop" << endl;

What happens here?
int i;
for(i=0; i > 0; i++)
    cout << "in loop" << endl;

What happens here?
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

```cpp
main_program{
    int n; cin >> n;
    bool found = false;
    for(int i=2; i < n && !found; i++){
        if(n % i == 0){
            found = true;
        }
    }
    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
Euclid’s Algorithm

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

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

$\Rightarrow$ $m$, $n$, $m-n$ 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 $\text{GCD}(3977, 943) = \text{GCD}(3977 \% 943, 943)$
Example

Should we guess that \( \text{GCD}(m, n) = \text{GCD}(m \% n, n) \)?

This is not true if \( m \) or \( n = 0 \), 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 \( \text{GCD}(m, n) = n \). Otherwise \( \text{GCD}(m, n) = \text{GCD}(m \% n, n) \).
Example Continued

\[ \text{GCD}(3977,943) \]

\[ = \text{GCD}(3977 \% 943, 943) \]

\[ = \text{GCD}(205, 943) = \text{GCD}(205, 943 \% 205) \]

\[ = \text{GCD}(205,123) = \text{GCD}(205 \% 123,123) \]

\[ = \text{GCD}(82, 123) = \text{GCD}(82, 123 \% 82) \]

\[ = \text{GCD}(82, 41) \]

\[ = 41 \quad \text{because 41 divides 82} \]
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

```cpp
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;
}
```

We could have done the assignment with just one variable as follows

- `int r = m%n; m = n; n = r;`
Remark

It should be intuitively clear that in writing the program, we have followed the idea from Euclid’s theorem and it should therefore be correct.

However, having written the program, we should check this again.
Termination and Correctness

• The questions to ask are:
  – Does it terminate?
  – Does it give 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,

$$\text{GCD}(m,n) = \text{GCD}(M,N)$$

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

\[
\text{GCD}(3977, 943) \quad \quad \quad \quad m=M=3977, \quad n=N=943
\]

\[
= \text{GCD}(3977 \% 943, 943)
\]

\[
= \text{GCD}(205, 943) = \text{GCD}(205, 943\%205) \quad m=205, \quad n=943
\]

\[
= \text{GCD}(205, 943) = \text{GCD}(205\%123, 123) \quad m=205, \quad n=123
\]

\[
= \text{GCD}(205, 943) = \text{GCD}(205\%123, 123) \quad m=205, \quad n=123
\]

\[
= \text{GCD}(82, 123) = \text{GCD}(82, 123\%82) \quad m=205, \quad n=123
\]

\[
= \text{GCD}(82, 41) \quad m=82, \quad n=41
\]

\[
= 41 \quad \text{because 41 divides 82}
\]
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 before the execution of the loop begins

Condition

False

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

True

Body

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, \( \text{next}n = m \% n \) which must be smaller than \( n \)),

• But \( n \) must always be a positive integer in every iteration: (because we enter an iteration only if \( m \% n \neq 0 \), and then set \( \text{next}n = m \% n \))

• 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?

unsigned int x;
cin >> x;
int y = 0;
while (x != y)
    y++;

x >= y
What is the Loop Invariant Here?

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

- **0 <= i < 10**: NO
- **0 <= i <= 10**: Yes, but not precise (misses j)
- **i+j = 9**: Yes, but not precise
- **i+j=9, 0<=i<=10**: Yes, most precise
Is $i+j=9$ a Loop Invariant Here?

<table>
<thead>
<tr>
<th>Visit to the condition</th>
<th>Value of i</th>
<th>Value of j</th>
<th>Loop body executed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>8</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>7</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>0</td>
<td>No</td>
</tr>
</tbody>
</table>

```
Is i+j=9 a Loop Invariant Here?

<table>
<thead>
<tr>
<th>Visit to the condition</th>
<th>Value of i</th>
<th>Value of j</th>
<th>Loop body executed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>8</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>7</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>0</td>
<td>No</td>
</tr>
</tbody>
</table>
```
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 issue 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...