C programming

Lecture 5: loops

Based on slides designed by Shay Artsi, Aythan Avior, Gitit Rockstein and Saher Ismir for the department of Computer Science, Technion, Israel Institute of Technology
Translated and updated by Anne Weill-Zrahia
Loops

- A code fragment which is executed several times is a loop
- Number of repetitions can be
  - Constant
  - Variable but determined before start of loop
  - Variable but determined during execution
  - Infinite.
- Each repetition is called an iteration
- A loop can be written in C in 3 modes:
  - While statement
  - do-while statement
  - For statement

Preferring one of the 3 modes will be determined by the specific Task there is to perform
**while**

- Syntax:
  ```
  while ( expression ) statement
  ```

- The value of `expression` is checked before execution of `statement`.

- As long as the value of `expression` stays TRUE, `statement` will continue to be executed.

- If the value of `expression` was false from the start, `statement` will never be executed.
Example 1: factorial

```c
int n;
int fact = 1;
int i = 1;
scanf("%d", &n);
while (i <= n) {
    fact *= i;
    i++;
}
printf("%d! = %d", n, fact);
```

- Factorial is defined as follows:

\[
\prod_{i=1}^{n} i = 1 \times 2 \times \ldots \times n
\]

- The following code will read the number n, compute the factorial, and print it.
Example 2: compute mean

- **Input**: series of non-negative numbers, terminated by 0
- **Output**: mean of the series (not including 0)
- **Assume**: input is valid

- The series only contains non-negative number
- There is at least one number (except for the final 0)

**Algorithm**

- Count numbers and sum them up through a loop
- At the end of the loop: compute sum divided by count ..
```c
#include <stdio.h>

int main()
{
    int value;
    int sum = 0;
    int num = 0;

    printf("Please insert a natural number: ");
    scanf("%d", &value);

    while ( value ) {
        sum += value;
        num ++;
        printf("Please insert a natural number: ");
        scanf("%d", &value);
    }

    printf("\nThe mean of the %d numbers is %.2f.\n", num, (double)sum / num);
    return 0;
}
```
**do-while**

- Syntax
  
do  
  statement  
  while (  
  expression  
  );

- The value of  
  expression  
  is checked after every occurrence of  
  statement  

- As long as  
  expression  
  remains  
  true,  
  statement  
  will continue being executed

- Even is  
  expression  
  has been  
  false from start,  
  statement  
  will execute (in other words, will execute at least once)
Example 2 – computing the mean—another implementation

```c
#include <stdio.h>

int main() {
    int value;
    int sum = 0;
    int num = 0;

    do {
        printf("Please insert a natural number: ");
        scanf("%d", &value);
        if ( value > 0 ) {
            sum += value;
            num ++;
        }
    } while ( value );

    printf("\nThe mean of the %d numbers is %.2f.\n", num, (double)sum / num);

    return 0;
}
```
for loop

- Syntax:
  \[ \text{for ( } \exp_1; \exp_2; \exp_3 \text{ ) statement} \]
- First \( \exp_1 \) is computed (once)
- Then the execution of the loop starts:
  - The value of \( \exp_2 \) is checked every time before executing \( \text{statement} \)
  - While the value of \( \exp_2 \) is true \( \text{statement} \) will be executed
  - If the value of \( \exp_2 \) is false from the beginning \( \text{statement} \) will not be executed
  - Expression \( \exp_3 \) will be computed after every execution of \( \text{statement} \)
- Every one of \( \exp_1, \exp_2 \) and \( \exp_3 \) can be empty
  - If \( \exp_1 \) or \( \exp_3 \) are empty, there is nothing to compute
  - If \( \exp_2 \) is empty, its value is true
For loop examples

```c
int n, i, fact;
scanf("%d", &n);
fact = 1;
for ( i = 1; i <= n; i++ )
    fact *= i;
printf("%d! = %d", n, fact);
```

```c
int sum, i;
sum = 0;
for ( i = 1; i <= n; i++ )
    sum += i * i;
```

\[ \sum_{i=1}^{n} i^2 \]
Example 3: finding minimum

- Find the minimum number in a series of n integers
- First try:

```c
int min = 0;
int i;
int num;

for ( i = 0; i < n; i++ ) {
    scanf("%d", &num);
    if ( min > num )
        min = num;  /* Found new minimum */
}
```

What is the problem?
Example 3: finding minimum-correction

- Variable `min` was initialized to 0
- If all the numbers are positive, minimum will be 0 instead of the smallest number
- Second try:

```c
int min;
int i;
int num;

scanf("%d", &min);

for ( i = 1;  i < n;  i++ ) {
    scanf("%d", &num);
    if (  min > num  )
        min = num;  /* Found new minimum */
}
```
Example 4: checking primeness

Definition: \( n \) is a prime iff it has only two divisors: 1 and itself.

We will use this definition to check if a number is prime.

\( n \) is prime if it is bigger than one and if none of 2, 3, ..., \( n - 1 \) divides \( n \) (without remainder).

```c
int is_prime = 1;
int n, i;

scanf("%d", &n);

if ( n == 1 ) {
    is_prime = 0;
}
else {
    for ( i = 2; i < n; i++ )
        if ( n % i == 0 )
            is_prime = 0;
}
```
More efficient solution

- We can only check divisors for \( \sqrt{n} \)
- If \( n \) is even it isn’t prime

```c
#include <math.h>
...
int is_prime = 1;
int n, i;
...
if ( n == 1 || ( n != 2 && n % 2 == 0 ) ) {
    is_prime = 0;
} else {
    int sqrt_n = (int) sqrt( n + 0.5 );
    for ( i = 3; i <= sqrt_n; i += 2 )
        if ( n % i == 0 )
            is_prime = 0;
    }
}
```

Without this line the program will be executed but not accurately.

Which inefficiency is there here?
Equivalence between loops

- All kinds of loops – while do-while for – are equivalent.
- They can be interchanged, for instance while can be changed to do-while as follows:

1. statement will be executed once
2. If cond is true it will be executed again, till it is no longer true.

Same as above

```
while (cond) {
    statement;
}
```

```
do {
    statement;
} while (cond);
```
Equivalence between loops

- This subject will be developed in the tutorial.
- Choice of the structure – depends on the problem!
  - If contents of the loop are to be executed at least once, go for the do-while
  - If contents of the loop will not be executed at all, prefer while
  - If the loop has a start, a continuation and a definite end point, better use for
break and continue

- Those are statements which change the course of events at run time.

- **Continue**
  - Stops execution of the current loop, and performs the next iteration
    - In a *for loop* $exp_3$ will be executed before going to the next iteration

- **Break**
  - Stops execution of the current loop and jumps outside of the loop to the next statement.
  - In a *switch* construction, there is a **break** statement after each of the multiple choices.
Continue – example

Goal

- Read 100 numbers from standard input
- Compute the sum of all positive numbers.

```c
for ( i = 0; i < 100; i++ ) {
    scanf("%d",&num);
    if ( num > 0 )
        sum += num;
}
```

Advantages of using `continue`

- When the loop contains many statements
- If there are reasons to add more continue statements
break - example

Goal
- Read 100 positive numbers from standard input (no negative numbers)
- Compute the sum of all positive numbers.

```
num = 0;
for ( i = 0;  i < 100 && num >= 0;  i++ ) {
    scanf("%d",&num);
    if ( num > 0 )
        sum += num;
}
```

```
for ( i = 0;  i < 100;  i++ ) {
    scanf("%d",&num);
    if ( num <= 0 )
        break;
    sum += num;
}
```

Advantages of using **break**
- Cleaner code
- Saving compute time,
Infinite loops

■ An infinite loop is a loop (of any kind) in which the stopping condition is always true.
■ In that case the program stays in the loop and never terminates.
■ An infinite loop results from a programming mistake, in which case execution of the code will be “brutally” stopped by:
  ■ The operating system (if possible)
  ■ Rebooting or shutting down the computer (in bad cases)
■ However creating such a loop is easy “

```c
while( 1 ) { ... }
```

```c
for( ; ; ) { ... }
```

An infinite loop is a loop (of any kind) in which the stopping condition is always true. In that case, the program stays in the loop and never terminates. An infinite loop results from a programming mistake, in which case execution of the code will be “brutally” stopped by: The operating system (if possible), rebooting or shutting down the computer (in bad cases). However, creating such a loop is easy.

```c
while( 1 ) { ... }
```

```c
for( ; ; ) { ... }
```
Some stuff from number theory ...

- **The division algorithm** says that: for each \( n, d \neq 0 \),
  \[ n = q \times d + r \]
  \[ 0 \leq r < d \]
  There is only 1 pair \( q,r \) verifying both above conditions.

- **definitions:**
  - \( n \) dividend
  - \( d \) divisor
  - \( q \) quotient
  - \( r \) remainder

- **Methods to find \( q \) and \( r \), given \( n \) and \( d \):**
  - **Method 1:**
    Subtract \( n \) from \( d \) as long as the remainder is nonnegative. The number of subtractions will be the quotient, :
  - **Method 2**
    Division till we get to the remainder.
  - Can be extended to negative numbers
...Some more number theory

Definitions:
- \( n/d \) has 0 remainder iff \( n \mid d \)
- Every natural number has at least 2 divisors: 1 and itself
- Every natural number >1 with only 2 divisors, is a prime number
- \( d \) is called a common divisor of \( m \) and \( n \) iff \( d \) is a divisor of both \( m \) and \( n \)

Theorem:
- Every 2 natural numbers \( m \) and \( n \) possess a greatest common divisor
- This greatest common divisor is named GCD (\( n,m \))
GCD – greatest common divisor

- GCD of 2 numbers can be useful in several cases:
  - Simplification of fractions
    \[
    \frac{n}{m} = \frac{\gcd(n, m)}{\gcd(n, m)}
    \]
  - Finding the lowest common multiplier (LCD) \(\text{lcm}(n, m)\),

\[
\text{lcm}(n,m) \times \gcd(n,m) = n \times m
\]

\[
\frac{a}{b} + \frac{c}{d} = \frac{a \times \text{lcm}(b,d)}{\text{lcm}(b,d)} + \frac{c \times \text{lcm}(b,d)}{\text{lcm}(b,d)}
\]
Finding the GCD

- **High school method:**
  - Reduce \( m \) and \( n \) to primary factors.
  - \( \text{Gcd}(n,m) \) is the product of common primary factors:

\[
\begin{align*}
  n &= 1350 = 2 \times 3^3 \times 5^2 \\
  m &= 700 = 2^2 \times 5^2 \times 7 \\
  \text{gcd}(1350, 700) &= 2 \times 5^2 = 50
\end{align*}
\]

- **Lcd : product of the union of common factors:**

\[
\begin{align*}
  \text{lcd}(1350, 700) &= 2^2 \times 3^3 \times 5^2 \times 7 = 18,900 \\
  \text{gcd}(1350, 700) \times \text{lcd}(1350, 700) &= 2^3 \times 3^3 \times 5^4 \times 7 = 945,000
\end{align*}
\]
Reduction to prime factors

- **Algorithm to reduce an integer n to its prime factors**
  - Start: remainder is n, factor list is empty
  - Do
    - Perform $n/p$ where $p=2..\sqrt{m}$ and $p$ is prime
    - If $m|p$ then :
      - Add $p$ to list of prime factors
      - $m=m/p$
  - Add remainder $m$ to factor list.

- **Example for $n = 90$:**
  $$90 = 2 \times 45 \quad \rightarrow \quad 45 = 3 \times 15 \quad \rightarrow \quad 15 = 3 \times 5 \quad \rightarrow \quad 5 = 5$$

- **This algorithm is not feasible for large n’s.** For instance, is $n$ is a 18-digit number we will have to go through 1,000,000,000 prime numbers. This is the base for modern cryptography.
Some more definitions:

For $n, m, k > 0$ it is known that:

- $d \mid n \Rightarrow d \mid n - d$
- $d \mid n \Rightarrow d \mid n + d$
- $d \mid n \Rightarrow d \mid n + k \cdot d$
- $d \mid n \Rightarrow d \mid n - k \cdot d$
- $d \mid n, d \mid m \Rightarrow d \mid n + m$
- $d \mid n, d \mid m \Rightarrow d \mid n - m$
- $d \mid n, d \mid m \Rightarrow d \mid n \% m$

\[
gcd(n, m) = gcd(n - m, m) = gcd(n \% m, m)
\]

\[
n > 0 \rightarrow gcd(n, n) = gcd(n, 0) = n
\]
Euclides algorithm for calculating the GCD

**Input:** n and m  ■

**Output:** \(\gcd(n,m)\).

**Method 1**

```
while m != n do:
    If m>n replace n by m-n

n=m therefore \(\gcd(n,m) = \gcd(n,n)\)
```

**Method 2**

```
While m>0 repeat {
    Exchange m and n%m
    Exchange n and m (so that \(n>m\))
}
```

```
m=0 therefore 
\(\gcd(n,m) = \gcd(n,0) = n.\)
```

- This algorithm is known since 370 B.C.
- Aristoteles (Ἀριστοτέλης) hints at it in 330 B.C.
- Euclides (Εὐκλείδης) presented it in his “Elements”, 300 B.C.
Euclid algorithm (method 2)

Example 1

\[ n = q \times m + r \]

\[
\begin{array}{c|c}
1350 & 700 \\
700 & 650 \\
650 & 50 \\
50 & 0 \\
\end{array}
\]

\[ \text{gcd}(1350, 700) = 50 \]

Example 2

<table>
<thead>
<tr>
<th>m</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>17</td>
</tr>
<tr>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ \text{gcd}(100, 17) = 1 \]

Two numbers without a common divisor are called co-primes.
#include <stdio.h>
int main()
{
    unsigned int   n, m;
    unsigned int t;

    scanf("%u%u", &n, &m);
    if    ( n == 0  &&  m == 0 )  { /* Error! */ }

    while ( n != m )
        if    ( n > m )
            n -= m;
        else
            m -= n;

    printf("The gcd is %u\n", n);
    return 0;
}

## Method 1

## Method 2 (division algorithm)