Introduction to Programming (in C++)

Loops

Jordi Cortadella, Ricard Gavaldà, Fernando Orejas
Dept. of Computer Science, UPC
• Assume the following specification:

**Input:** read a number $N > 0$
**Output:** write the sequence $1\ 2\ 3\ ...\ N$
(one number per line)

• This specification suggests some algorithm with a *repetitive* procedure.
The \textit{while} statement

• Syntax:

\texttt{while (\langle condition \rangle )} statement;

(the condition must return a Boolean value)

• Semantics:
  – Similar to the repetition of an \textit{if} statement
  – The condition is evaluated:
    • If \textit{true}, the statement is executed and the control returns to the while statement again.
    • If \textit{false}, the while statement terminates.
Write the numbers 1...N

// Input: read a number N > 0
// Output: write the numbers 1...N
// (one per line)

```cpp
int main() {
    int N;
    cin >> N;
    int i = 1;
    while (i <= N) {
        // The numbers 1..i-1 have been written
        cout << i << endl;
        i = i + 1;
    }
}
```
Product of two numbers

//Input: read two non-negative numbers x and y
//Output: write the product x*y

// Constraint: do not use the * operator

// The algorithm calculates the sum x+x+x+...+x (y times)

int main() {
    int x, y;
    cin >> x >> y; // Let x=A, y=B
    int p = 0;
    // Invariant: A*B = p + x*y
    while (y > 0) {
        p = p + x;
        y = y - 1;
    }
    cout << p << endl;
}
A quick algorithm for the product

• Let $p$ be the product $x \times y$

• Observation
  – If $y$ is even, $p = (x \times 2) \times (y/2)$
  – If $y$ is odd, $p = x \times (y-1) + x$ and $(y-1)$ becomes even

• Example: $17 \times 38 = 646$
A quick algorithm for the product

```c
int main() {
    int x, y;
    cin >> x >> y; // Let x=A, y=B
    int p = 0;
    // Invariant: A*B = p + x*y
    while (y > 0) {
        if (y % 2 == 0) {
            x = x*2;
            y = y/2;
        }
        else {
            p = p + x;
            y = y - 1;
        }
    }
    cout << p << endl;
}
```

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>38</td>
<td>0</td>
</tr>
<tr>
<td>34</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>34</td>
<td>18</td>
<td>34</td>
</tr>
<tr>
<td>68</td>
<td>9</td>
<td>34</td>
</tr>
<tr>
<td>68</td>
<td>8</td>
<td>102</td>
</tr>
<tr>
<td>136</td>
<td>4</td>
<td>102</td>
</tr>
<tr>
<td>272</td>
<td>2</td>
<td>102</td>
</tr>
<tr>
<td>544</td>
<td>1</td>
<td>102</td>
</tr>
<tr>
<td>544</td>
<td>0</td>
<td>646</td>
</tr>
</tbody>
</table>
Why is the quick product interesting?

• Most computers have a multiply instruction in their machine language.

• The operations $x \times 2$ and $y / 2$ can be implemented as 1-bit left and right shifts, respectively. So, the multiplication can be implemented with shift and add operations.

• The quick product algorithm is the basis for hardware implementations of multipliers and mimics the paper-and-pencil method learned at school (but using base 2).
Quick product in binary: example

\[77 \times 41 = 3157\]

\[
\begin{array}{c}
1001101 \\
x \ 0101001 \\
\hline
1001101 \\
1001101 \\
1001101 \\
\hline
110001010101
\end{array}
\]
Counting a’s

• We want to read a text represented as a sequence of characters that ends with ‘.’

• We want to calculate the number of occurrences of the letter ‘a’

• We can assume that the text always has at least one character (the last ‘.’)

• Example: the text

    Programming in C++ is amazingly easy!.

    has 4 a’s
Counting a’s

// Input: sequence of characters that ends with ‘.’
// Output: number of times ‘a’ appears in the
//         sequence

int main() {
    char c;
    cin >> c;
    int count = 0;
    // Inv: count is the number of a’s in the visited
    //      prefix of the sequence. c contains the next
    //      non-visited character
    while (c != '.') {
        if (c == 'a') count = count + 1;
        cin >> c;
    }

    cout << count << endl;
}
Counting digits

• We want to read a non-negative integer and count the number of digits (in radix 10) in its textual representation.

• Examples

  8713105 \(\rightarrow\) 7 digits
  156 \(\rightarrow\) 3 digits
  8 \(\rightarrow\) 1 digit
  0 \(\rightarrow\) 1 digit (note this special case)
Counting digits

// Input: a non-negative number N
// Output: number of digits in N (0 has 1 digit)

int main() {
    int N;
    cin >> N;
    int ndigits = 0;

    // Inv: ndigits contains the number of digits in the
    //      tail of the number, N contains the remaining
    //      part (head) of the number
    while (N > 9) {
        ndigits = ndigits + 1;
        N = N/10; // extracts one digit
    }

    cout << ndigits + 1 << endl;
}

Euclid’s algorithm for gcd

- Properties
  - gcd(a, a) = a
  - If a > b, then gcd(a, b) = gcd(a - b, b)

- Example

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>114</td>
<td>42</td>
</tr>
<tr>
<td>72</td>
<td>42</td>
</tr>
<tr>
<td>30</td>
<td>42</td>
</tr>
<tr>
<td>30</td>
<td>12</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
```

gcd (114, 42)
Euclid’s algorithm for gcd

// Input: read two positive numbers (a and b)
// Output: write gcd(a,b)

int main() {
    int a, b;
    cin >> a >> b; // Let a=A, b=B
    // gcd(A,B) = gcd(a,b)
    while (a != b) {
        if (a > b) a = a - b;
        else b = b - a;
    }
    cout << a << endl;
}

// Let A and B be positive integers.
Faster Euclid’s algorithm for gcd

• Properties
  – \( \text{gcd}(a, 0) = a \)
  – If \( b > 0 \) then \( \text{gcd}(a, b) = \text{gcd}(b, a \mod b) \)

• Example

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>114</td>
<td>42</td>
</tr>
<tr>
<td>42</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>
Faster Euclid’s algorithm for gcd

// Input: read two positive numbers (a and b)
// Output: write gcd(a,b)

```csharp
int main() {
    int a, b;
    cin >> a >> b; // Let a=A, b=B
    // gcd(A,B) = gcd(a,b)
    while (b != 0) {
        int r = a % b;
        a = b;
        b = r; // Guarantees b < a (loop termination)
    }
    cout << a << endl;
}
```
Efficiency of Euclid’s algorithm

• How many iterations will Euclid’s algorithm need to calculate gcd(a,b) in the worst case (assume a > b)?

  – Subtraction version: a iterations  
    (consider gcd(1000,1))

  – Modulo version: $\leq 5*d(b)$ iterations,  
    where d(b) is the number of digits of b represented in base 10 (proof by Gabriel Lamé, 1844)
Solving a problem several times

- In many cases, we might be interested in solving the same problem for several input data.

- Example: calculate the gcd of several pairs of natural numbers.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 56</td>
<td>4</td>
</tr>
<tr>
<td>30 30</td>
<td>30</td>
</tr>
<tr>
<td>1024 896</td>
<td>128</td>
</tr>
<tr>
<td>100 99</td>
<td>1</td>
</tr>
<tr>
<td>17 51</td>
<td>17</td>
</tr>
</tbody>
</table>
Solving a problem several times

// Input: several pairs of natural numbers at the input
// Output: the gcd of each pair of numbers written at the output

int main() {
    int a, b;
    // Inv: the gcd of all previous pairs have been calculated and written at the output
    while (cin >> a >> b) {
        // A new pair of numbers from the input
        int r = a % b;
        a = b;
        b = r;
        cout << a << endl;
    }
}

Calculate \texttt{gcd(a,b)} and write the result into \texttt{cout}
Solving a problem several times

// Input: several pairs of natural numbers at the input
// Output: the gcd of each pair of numbers written at the output

int main() {
    int a, b;
    // Inv: the gcd of all previous pairs have been
    // calculated and written at the output
    while (cin >> a >> b) {
        // A new pair of numbers from the input
        while (b != 0) {
            int r = a%b;
            a = b;
            b = r;
        }
        cout << a << endl;
    }
}
Prime number

• A **prime number** is a natural number that has exactly two **distinct** divisors: 1 and itself. (Comment: 1 is not prime)

• Write a program that reads a natural number (N) and tells whether it is prime or not.

• Algorithm: try all potential divisors from 2 to N-1 and check whether the remainder is zero.
// Input: read a natural number N>0
// Output: write “is prime” or “is not prime” depending on
//         the primality of the number

int main() {
    int N;
    cin >> N;

    int divisor = 2;
    bool is_prime = (N != 1);
    // 1 is not prime, 2 is prime, the rest enter the loop (assume prime)

    // is_prime is true while a divisor is not found
    // and becomes false as soon as the first divisor is found
    while (divisor < N) {
        if (N % divisor == 0) is_prime = false;
        divisor = divisor + 1;
    }

    if (is_prime) cout << “is prime” << endl;
    else cout << “is not prime” << endl;
}
Prime number

• Observation: as soon as a divisor is found, there is no need to check divisibility with the rest of the divisors.

• However, the algorithm tries all potential divisors from 2 to N-1.

• Improvement: stop the iteration when a divisor is found.
Prime number

// Input: read a natural number N>0
// Output: write “is prime” or “is not prime” depending on
// the primality of the number

int main() {
    int N;
    cin >> N;

    int divisor = 2;
    bool is_prime = (N != 1);

    while (is_prime and divisor < N) {
        is_prime = N % divisor != 0;
        divisor = divisor + 1;
    }

    if (is_prime) cout << “is prime” << endl;
    else cout << “is not prime” << endl;
}
Prime number: doing it faster

• If $N$ is not prime, we can find two numbers, $a$ and $b$, such that:
  
  $$N = a \times b, \quad \text{with } 1 < a \leq b < N$$

  and with the following property: $a \leq \sqrt{N}$

• There is no need to find divisors up to $N-1$. We can stop much earlier.

• Note: $a \leq \sqrt{N}$ is equivalent to $a^2 \leq N$
Prime number: doing it faster

// Input: read a natural number N > 0
// Output: write “is prime” or “is not prime” depending on the primality of the number

int main() {
    int N;
    cin >> N;

    int divisor = 2;
    bool is_prime = (N != 1);

    while (is_prime and divisor*divisor <= N) {
        is_prime = N%divisor != 0;
        divisor = divisor + 1;
    }

    if (is_prime) cout << “is prime” << endl;
    else cout << “is not prime” << endl;
}
Is there any real difference?

Number of iterations to detect a prime as a function of the number of significant bits of the number.
In real time (N= 2110454939)

```
> time prime_slow < number
is prime
10.984u 0.004s 0:11.10 98.9%
```

```
> time prime_fast < number
is prime
0.004u 0.000s 0:00.00 0.0%
```
The *for* statement

• Very often we encounter loops of the form:

```c
i = N;
while (i <= M) {
    do_something;
    i = i + 1;
}
```

• This can be rewritten as:

```c
for (i = N; i <= M; i = i + 1) {
    do_something;
}
```
The *for* statement

- In general

\[ \text{for} \ (\langle S_{\text{init}} \rangle; \langle \text{condition} \rangle; \langle S_{\text{iter}} \rangle) \ \langle S_{\text{body}} \rangle; \]

is equivalent to:

\[ S_{\text{init}}; \]
\[ \text{while} \ (\langle \text{condition} \rangle) \ {\}
  \langle S_{\text{body}} \rangle;
  \langle S_{\text{iter}} \rangle;
\]
Writing the numbers in an interval

// Input: read two integer numbers, N and M, such that N <= M.
// Output: write all the integer numbers in the interval [N,M]

int main() {
    int N, M;
    cin >> N >> M;

    for (int i = N; i <= M; ++i) cout << i << endl;
}

Variable declared within the scope of the loop

Autoincrement operator
Calculate $x^y$

// Input: read two integer numbers, x and y, such that y >= 0

// Output: write $x^y$

```cpp
int main() {
    int x, y;
    cin >> x >> y;
    int p = 1;
    for (int i = 0; i < y; ++i) p = p*x;
    cout << p << endl;
}
```
• Given a number \( n \) (e.g. \( n = 6 \)), we want to draw this triangle:

*  
**  
***  
****  
*****  
******  
*******
// Input: read a number n > 0
// Output: write a triangle of size n

int main() {
    int n;
    cin >> n;
    // Inv: the rows 1..i-1 have been written
    for (int i = 1; i <= n; ++i) {
        // Inv: ‘*’ written j-1 times in row i
        for (int j = 1; j <= i; ++j) cout << ‘*’;
        cout << endl;
    }
}

Perfect numbers

• A number $n > 0$ is perfect if it is equal to the sum of all its divisors except itself.

• Examples
  – 6 is a perfect number ($1+2+3 = 6$)
  – 12 is not a perfect number ($1+2+3+4+6 \neq 12$)

• Strategy
  – Keep adding divisors until the sum exceeds the number or there are no more divisors.
Perfect numbers

// Input: read a number n > 0
// Output: write a message indicating
//         whether it is perfect or not

int main() {
    int n;
    cin >> n;

    int sum = 0, d = 1;
    // Inv: sum is the sum of all divisors until d-1
    while (d <= n/2 and sum <= n) {
        if (n%d == 0) sum += d;
        d = d + 1;
    }

    if (sum == n) cout << "is perfect" << endl;
    else cout << "is not perfect" << endl;
}
Perfect numbers

• Would the program work using the following loop condition?

    while (d <= n/2 and sum < n)

• Can we design a more efficient version without checking all the divisors until n/2?
  – Clue: consider the most efficient version of the program to check whether a number is prime.