

Introduction to Programming (in C++)

*Algorithms on sequences.
Reasoning about loops: Invariants.*

Jordi Cortadella, Ricard Gavalrà, Fernando Orejas
Dept. of Computer Science, UPC

Outline

- Algorithms on sequences
 - Treat-all algorithms
 - Search algorithms
- Reasoning about loops: invariants

Maximum of a sequence

- Write a program that tells the largest number in a non-empty sequence of integers.

```
// Pre:  a non-empty sequence of integers is  
//       ready to be read at cin  
// Post: the maximum number of the sequence has been  
//       written at the output
```

Assume the input sequence is: 23 12 -16 34 25

elem:	-	12	-16	34	25
m:	23	23	23	34	34

```
// Invariant: m is the largest number read  
//           from the sequence
```

Maximum of a sequence

```
int main() {  
    int m;  
  
    int elem;  
    cin >> m;  
    // Inv: m is the largest element read  
    //       from the sequence  
    while (cin >> elem) {  
        if (elem > m) m = elem;  
    }  
    cout << m << endl;  
}
```

Why is this necessary?

Checks for end-of-sequence and reads a new element.

Reading with `cin`

- The statement `cin >> n` can also be treated as a Boolean expression:
 - It returns *true* if the operation was successful
 - It returns *false* if the operation failed:
 - no more data were available (EOF condition) or
 - the data were not formatted correctly (e.g. trying to read a double when the input is a string)
- The statement:

`cin >> n`

can be used to detect the end of the sequence and read a new element simultaneously. If the end of the sequence is detected, `n` is not modified.

Finding a number greater than N

- Write a program that detects whether a sequence of integers contains a number greater than N.

```
// Pre:  at the input there is a non-empty sequence of
//        integers in which the first number is N.
// Post: writes a Boolean value that indicates whether
//        a number larger than N exists in the sequence.
```

Assume the input sequence is: 23 12 -16 24 25

num:	-	12	-16	24
N:	23	23	23	23
found:	false	false	false	true

```
// Invariant: “found” indicates that a value greater than
//            N has been found.
```

Finding a number greater than N

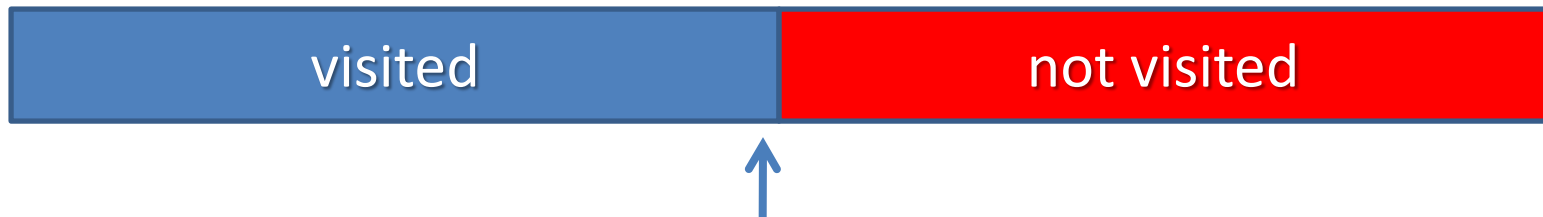
```
int main() {  
    int N, num;  
    cin >> N;  
    bool found = false;  
  
    // Inv: found indicates that a number  
    //       greater than N has been found  
    while (not found and cin >> num) {  
        found = num > N;  
    }  
    cout << found << endl;  
}
```

Algorithmic schemes on sequences

- The previous examples perform two different operations on a sequence of integers:
 - Finding the maximum number
 - Finding whether there is a number greater than N
- They have a distinctive property:
 - The former requires **all elements to be visited**
 - The latter requires **one element to be found**

Treat-all algorithms

- A classical scheme for algorithms that need to treat all the elements in a sequence



```
Initialize (the sequence and the treatment)
// Inv: The visited elements have been treated
while (not end of sequence) {
    Get a new element e;
    Treat e;
}
```

Search algorithms

- A classical scheme for algorithms that need to find an element with a certain property in a sequence

```
bool found = false;
Initialize;
// Inv: “found” indicates whether the element has been
//      found in the visited part of the sequence
while (not found and not end of sequence) {
    Get a new element e;
    if (Property(e)) found = true;
}
// “found” indicates whether the element has been found.
// “e” contains the element.
```

Longest repeated subsequence

- Assume we have a sequence of strings

cat dog bird cat bird bird cat cat cat dog mouse horse

- We want to calculate the length of the longest sequence of repetitions of the first string. Formally, if we have a sequence of strings

$$s_1, s_2, \dots, s_n$$

we want to calculate

$$\max\{j - i + 1 : 1 \leq i \leq j \leq n \wedge s_i = s_{i+1} = \dots = s_{j-1} = s_j = s_1\}.$$

Longest repeated subsequence

```
// Specification: see previous slide
// Variable to store the first string.
string first;
cin >> first;
string next; // Visited string in the sequence
// Length of the current and longest subsequences
int length = 1, longest = 1;

// Inv: "length" is the length of the current subsequence.
//      "longest" is the length of the longest subsequence
//      visited so far.
while (cin >> next) {
    if (first != next) length = 0; // New subsequence
    else {
        // The current one is longer
        length = length + 1;
        if (length > longest) longest = length;
    }
}

// "longest" has the length of the longest subsequence
```

Search in the dictionary

- Assume we have a sequence of strings representing words. The first string is a word that we want to find in the dictionary that is represented by the rest of the strings. The dictionary is ordered alphabetically.
- Examples:

dog ant bird cat cow dog eagle fox lion mouse pig rabbit shark whale yak

frog ant bird cat cow dog eagle fox lion mouse pig rabbit shark whale yak
- We want to write a program that tells us whether the first word is in the dictionary or not.

Search in the dictionary

```
// Specification: see previous slide
// First word in the sequence (to be sought).
string word;
cin >> word;

// A variable to detect the end of the search
// (when a word is found that is not smaller than "word").
bool found = false;

// Visited word in the dictionary (initialized as empty for
// the case in which the dictionary might be empty).
string next = "";

// Inv: not found => the visited words are smaller than "word"
while (not found and cin >> next) found = next >= word;
// "found" has detected that there is no need to read the rest of
// the dictionary
found = word == next;
// "found" indicates that the word was found.
```

Increasing number

- We have a natural number n . We want to know whether its representation in base 10 is a sequence of increasing digits.

- Examples:

134679	→ increasing
56688	→ increasing
3	→ increasing
1347 2 9	→ non-increasing

Increasing number (I)

```
// Pre:  n >= 0
// Post: returns true if the sequence of digits representing n (in base 10)
//       is increasing, and returns false otherwise

bool increasing(int n) {
    // The algorithm visits the digits from LSB to MSB.
    bool incr = true;
    int previous = 9; // Stores a previous “fake” digit

    // Inv: n contains the digits not yet treated, previous contains the
    //      last treated digit (that can never be greater than 9),
    //      incr implies all the treated digits form an increasing sequence
    while (incr and n > 0) {
        int next = n%10;
        incr = next <= previous;
        previous = next;
        n /= 10;
    }
    return incr;
}
```


Increasing number (II)

```
// Pre:  n >= 0
// Post: returns true if the sequence of digits representing n (in base 10)
//       is increasing, and returns false otherwise

bool increasing(int n) {
    // The algorithm visits the digits from LSB to MSB.
    int previous = 9; // Stores a previous “fake” digit

    // Inv: n contains the digits no yet treated, previous contains the
    //      last treated digit (that can never be greater than 9) and
    //      all the previously treated digits form an increasing sequence
    while (n > 0) {
        int next = n%10;
        if (next > previous) return false;
        previous = next;
        n /= 10;
    }
    return true;
}
```

Exercise: write the function `increasing(int n, int b)` with the same specification, but for a number representation in base `b`.

Insert a number in an ordered sequence

- Read a sequence of integers that are all in ascending order, except the first one. Write the same sequence with the first element in its correct position.
- Note: the sequence has at least one number. The output sequence must have a space between each pair of numbers, but not before the first one or after the last one.
- Example

Input: **15** 2 6 9 12 18 20 35 75

Output: 2 6 9 12 **15** 18 20 35 75

- The program can be designed with a combination of search and treat-all algorithms.

Insert a number in an ordered sequence

```
int first;
cin >> first;

bool found = false;           // controls the search of the location
int next;                     // the next element in the sequence

// Inv: All the read elements that are smaller than the first have been written
//       not found => no number greater than or equal to the first has been
//       found yet
while (not found and cin >> next) {
    if (next >= first) found = true;
    else cout << next << " ";
}

cout << first;

if (found) {
    cout << " " << next;
    // Inv: all the previous numbers have been written
    while (cin >> next) cout << " " << next;
}
cout << endl;
```

Counting words

- We have a sequence of characters representing a text that ends with ‘.’
- We want to calculate the number of words in the text.
- A word is a sequence of letters. Words are separated by characters that are not letters.
- There could be an undefined number of separators before the first word and after the last word.
- Example: the text

ii Today is Friday !! Alice, are we going out for dinner? .

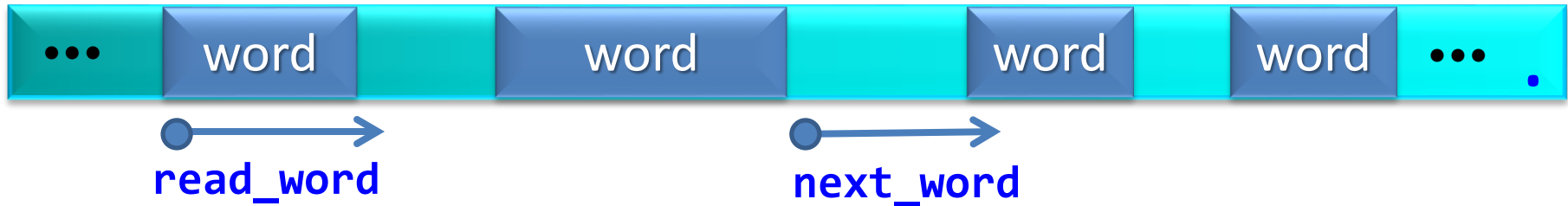
has 10 words.

Counting words

**// Pre: the input contains a sequence of
// characters that ends with ‘.’**

**// Post: the number of words in the
// sequence has been written at
// the output.**

Counting words



```
char read_word():
```

```
// Pre:  the last char from the input was a letter  
// Post: returns the next char that is not a letter.
```

```
char next_word(char c):
```

```
// Pre:  c is the last non-letter char read from the input.  
// Post: returns the next char that is a letter or a '.'
```

Note: next_word will return c if it is a '.'

Counting words

```
int main() {  
  
    char c = next_word(' ');    // A fake ' ' as parameter  
  
    int count = 0;  
  
    // Inv: count is the number of words in the treated part  
    //       of the input. c contains the first char in the  
    //       non-treated part  
    while (c != '.') {  
        count = count + 1;  
        c = read_word();  
        c = next_word(c);  
        // or: c = next_word(read_word());  
    }  
    cout << count << endl;  
}
```

Counting words

```
char next_word(char c) {  
    // Pre:  c is the last non-letter char read from the input.  
    // Post: returns the next char that is a letter or a '.'  
  
    bool found = false;  
  
    // Inv: found indicates that a letter has been found.  
    //      c is the last char read from the input  
    while (c != '.' and not found){  
        c = cin.get();  
        found = is_letter(c);  
        /* cin.get() returns the next char at the input.  
           (cin>>c skips spaces and other separating chars) */  
    }  
    return c;  
}
```


Counting words

```
char next_word(char c) { // a simpler solution
// Pre:  c is the last non-letter char read from the input.
// Post: returns the next char that is a letter or a '.'

    // Inv: c is the last char read from the input
    while (c != '.' and not is_letter(c)) c = cin.get();
    return c;
}
```

Counting words

```
char read_word() {  
    // Pre:  the last char from the input was a letter  
    // Post: returns the next char that is not a letter.  
  
    char c = cin.get(); // Next char after the first letter  
  
    // Inv: c is the last char read from the input  
    while (is_letter(c)) c = cin.get();  
    return c;  
}
```

Counting words

```
bool is_letter(char c) {  
  
    // Pre: -  
    // Post: indicates whether c is a letter  
  
    return ('a' <= c and c <= 'z') or  
           ('A' <= c and c <= 'Z');  
}
```

REASONING ABOUT LOOPS: INVARIANTS

Invariants

- Invariants help to ...
 - Define how variables must be initialized before a loop
 - Define the necessary condition to reach the post-condition
 - Define the body of the loop
 - Detect whether a loop terminates
- It is crucial, but not always easy, to choose a good invariant.
- Recommendation:
 - Use invariant-based reasoning for all loops (possibly in an informal way)
 - Use formal invariant-based reasoning for non-trivial loops

General reasoning for loops

Initialization;

```
// Invariant: a proposition that holds  
//      * at the beginning of the loop  
//      * at the beginning of each iteration  
//      * at the end of the loop
```

// Invariant

```
while (condition) {  
    // Invariant  $\wedge$  condition  
    Body of the loop;  
    // Invariant  
}
```

// Invariant $\wedge \neg$ condition

Example with invariants

- Given $n \geq 0$, calculate $n!$
- Definition of factorial:

$$n! = 1 * 2 * 3 * \dots * (n-1) * n$$

(particular case: $0! = 1$)

- Let's pick an invariant:
 - At each iteration we will calculate $f = i!$
 - We also know that $i \leq n$ at all iterations

Calculating $n!$

```
// Pre:   $n \geq 0$ 
// Post: returns  $n!$ 
int factorial(int n) {
    int i = 0;
    int f = 1;
    // Invariant:  $f = i!$  and  $i \leq n$ 
    while (i < n) {
        //  $f = i!$  and  $i < n$ 
        i = i + 1;
        f = f*i;
        //  $f = i!$  and  $i \leq n$ 
    }
    //  $f = i!$  and  $i \leq n$  and  $i \geq n$ 
    //  $f = n!$ 
    return f;
}
```


Reversing digits

- Write a function that reverses the digits of a number (representation in base 10)
- Examples:

35276 → 67253

19 → 91

3 → 3

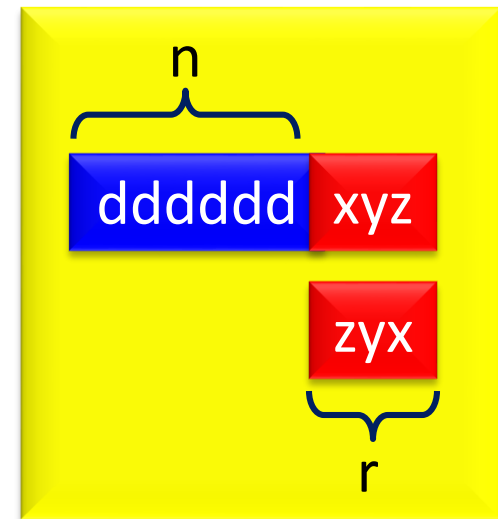
0 → 0

Reversing digits

// Pre: $n \geq 0$

// Post: returns n with reversed digits (base 10)

```
int reverse_digits(int n) {  
    int r;  
  
    r = 0;  
    // Invariant (graphical): →  
    while (n > 0) {  
        r = 10*r + n%10;  
        n = n/10;  
    }  
  
    return r;  
}
```



Calculating π

- π can be calculated using the following series:

$$\frac{\pi}{2} = 1 + \frac{1}{3} + \frac{1 \cdot 2}{3 \cdot 5} + \frac{1 \cdot 2 \cdot 3}{3 \cdot 5 \cdot 7} + \dots$$

- Since an infinite sum cannot be computed, it may often be sufficient to compute the sum with a finite number of terms.

Calculating π

```
// Pre:  nterms > 0
```

```
// Post: returns an estimation of  $\pi$  using nterms terms  
//       of the series
```

```
double Pi(int nterms) {  
    double sum = 1;           // Approximation of  $\pi/2$   
    double term = 1;          // Current term of the sum  
  
    // Inv: sum is an approximation of  $\pi/2$  with k terms,  
    //       term is the k-th term of the series.  
    for (int k = 1; k < nterms; ++k) {  
        term = term*k/(2.0*k + 1.0);  
        sum = sum + term;  
    }  
    return 2*sum;  
}
```

Calculating π

- $\pi = 3.14159265358979323846264338327950288\dots$
- The series approximation:

nterms	Pi(nterms)
1	2.000000
5	3.098413
10	3.140578
15	3.141566
20	3.141592
25	3.141593