Introduction to Programming (in C++)

Vectors and strings

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

Outline

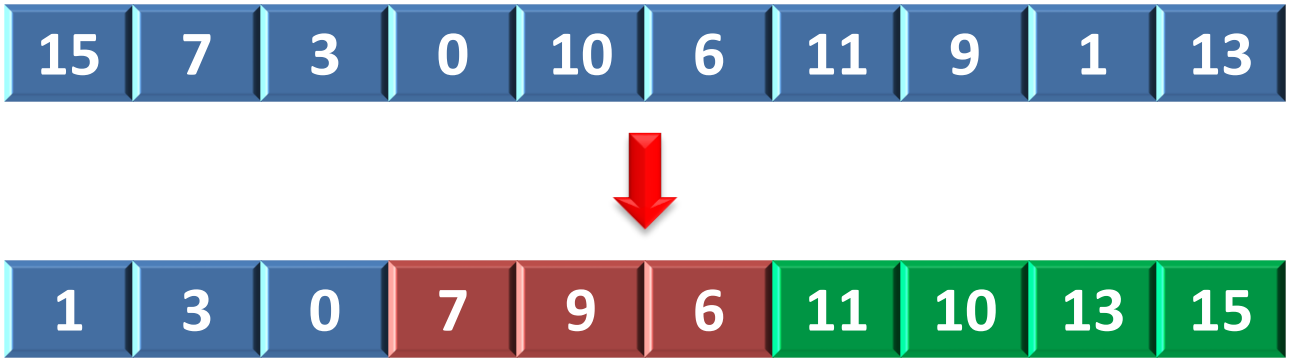
- More vector examples

- Strings

Classify elements

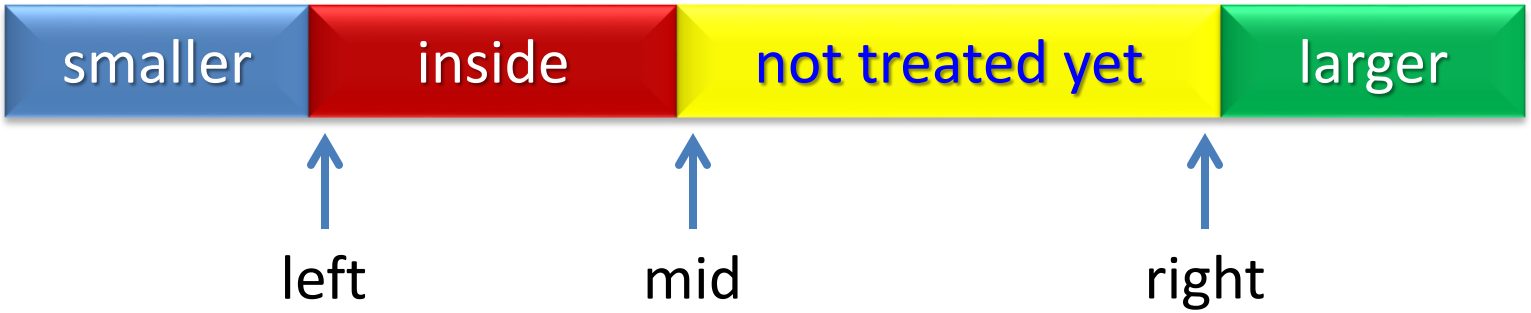
- We have a vector of elements V and an interval [x,y] (x ≤ y). Classify the elements of the vector by putting those smaller than x in the left part of the vector, those larger than y in the right part and those inside the interval in the middle. The elements do not need to be ordered.

- Example: interval [6,9]



Classify elements

- Invariant:



- At each iteration, we treat the element in the middle
  - If it is smaller, swap the elements in left and the middle (left→, mid→)
  - If larger, swap the elements in the middle and the right (←right)
  - If inside, do not move the element (mid→)

- End of classification: when mid > right.
  Termination is guaranteed since mid and right get closer at each iteration.

- Initially: left = mid = 0, right = size-1

Classify elements

```
// Pre:  x <= y
// Post: the elements of V have been classified moving those
//       smaller than x to the left, those larger than y to the
//       right and the rest in the middle.

void classify(vector<int>& V, int x, int y) {
    int left = 0;
    int mid = 0;
    int right = V.size() - 1;

    // Invariant: see the previous slide
    while (mid <= right) {
        if (V[mid] < x) {                // Put in the left part
            swap(V[mid], V[left]);
            left = left + 1;
            mid = mid + 1;
        } else if (V[mid] > y) {         // Put in the right part
            swap(V[mid], V[right]);
            right = right - 1;
        } else mid = mid + 1;            // Put in the middle
    }
}
```

Palindrome vector

- Design a function that checks whether a vector is a palindrome (the reverse of the vector is the same as the vector). For example:

is a palindrome.

Palindrome vector

```
bool palindrome(const vector<int>& A);

// Pre: --
// Returns true if A is a palindrome, and false otherwise.
```

Invariant:

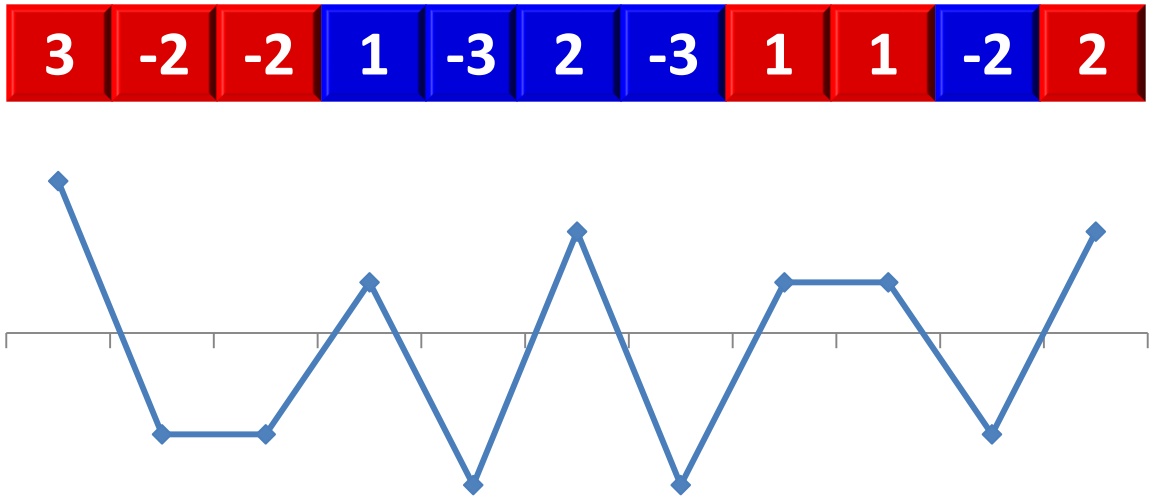
The fragments A[0..i-1] and A[k+1..A.size()-1] are reversed

Palindrome vector

```
// Pre: --
// Returns true if A is a palindrome,
// and false otherwise.

bool palindrome(const vector<int>& A) {
    int i = 0;
    int k = A.size() - 1;

    while (i < k) {
        if (A[i] != A[k]) return false;
        else {
            i = i + 1;
            k = k - 1;
        }
    }
    return true;
}
```

Peaks of a vector

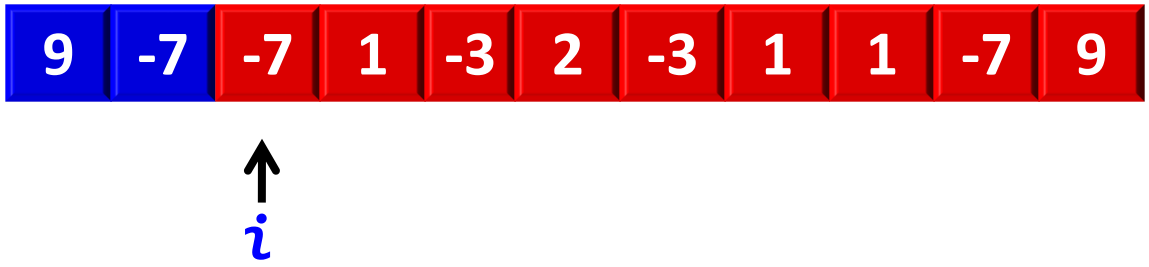
- Design a function that counts the number of peaks of a vector. A peak is the last element of a strictly increasing sequence and the first element of a strictly decreasing sequence, or vice versa. The extremes of a vector are not considered peaks. For example, the following vector has 5 peaks (in blue):



Peaks of a vector

```
// Pre:  --
// Returns the number of peaks of A.
int peaks(const vector<int>& A);
```

Invariant:



p is the number of peaks between locations 0 and i-1

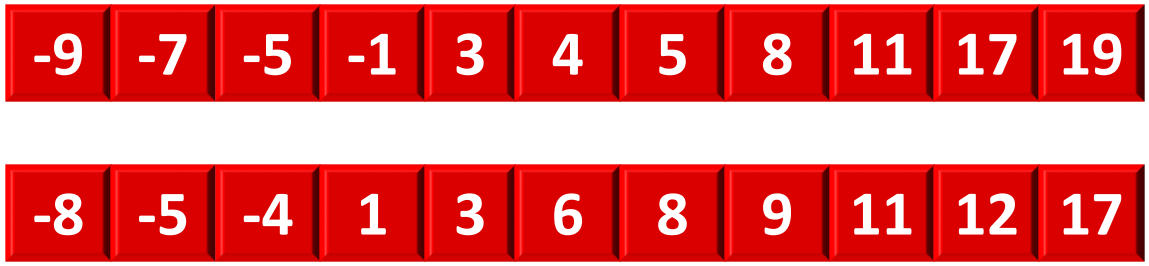
Peaks of a vector

```
// Pre: --
// Returns the number of peaks of A.

int peaks(const vector<int>& A) {
    int p = 0;
    int n = A.size();
    for (int i = 1; i < n - 1; ++i) {
        if (A[i - 1] < A[i] and A[i] > A[i + 1]) or
           (A[i - 1] > A[i] and A[i] < A[i + 1]) {
            p = p + 1;
        }
    }
    return p;
}
```

Common elements

- Design a function that counts the number of common elements of two vectors sorted in strict ascending order (the vectors cannot contain repetitions).

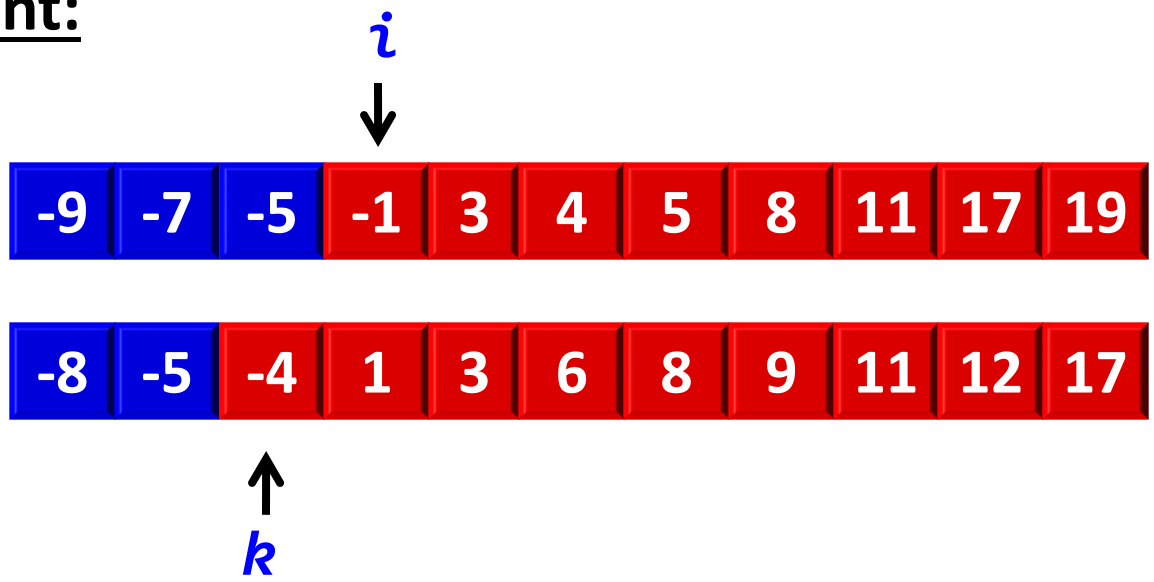
- Example: the two vectors below have 5 common elements.



Common elements

```
// Pre: A and B are two vectors sorted in strict ascending order.
// Returns the number of common elements of A and B.
int common(const vector<int>& A, const vector<int>& B);
```

Invariant:



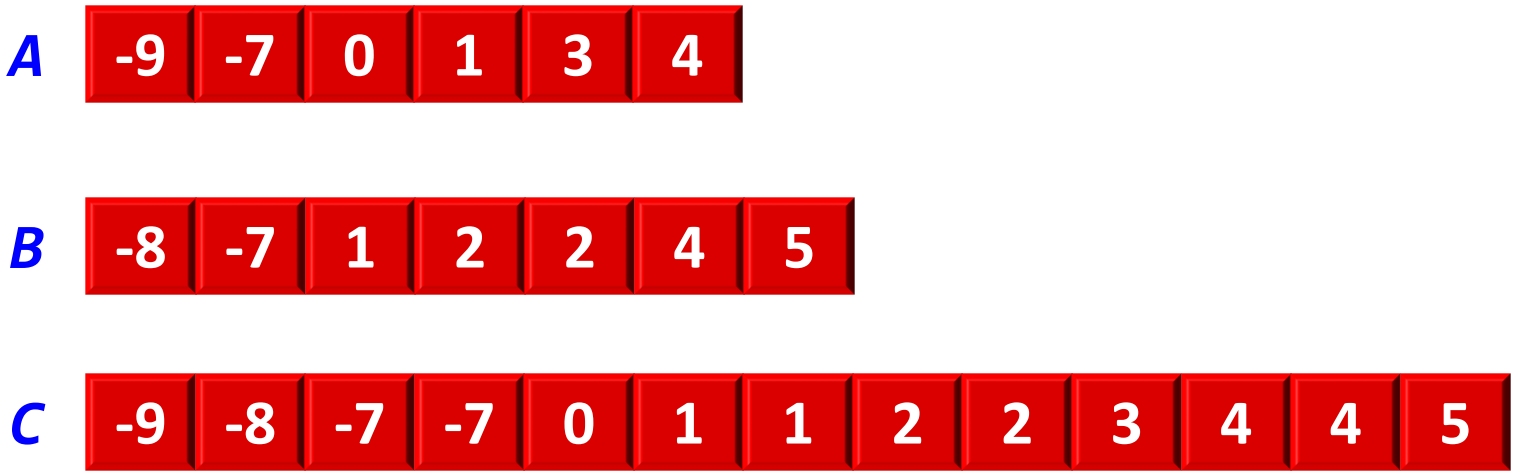
- **n is the number of common elements of A[0..i-1] and B[0..k-1].**
- **All the visited elements are smaller than the non-visited ones.**

Common elements

```
// Pre: A and B are two vectors sorted in strict ascending order.
// Returns the number of common elements of A and B.

int common(const vector<int>& A, const vector<int>& B) {
    int i, k, n;
    i = k = n = 0;
    while (i < A.size() and k < B.size()) {
        if (A[i] < B[k]) i = i + 1;
        else if (A[i] > B[k]) k = k + 1;
        else {
            i = i + 1;
            k = k + 1;
            n = n + 1;
        }
    }
    return n;
}
```

Vector fusion

- Design a function that returns the fusion of two ordered vectors. The returned vector must also be ordered. For example, C is the fusion of A and B:



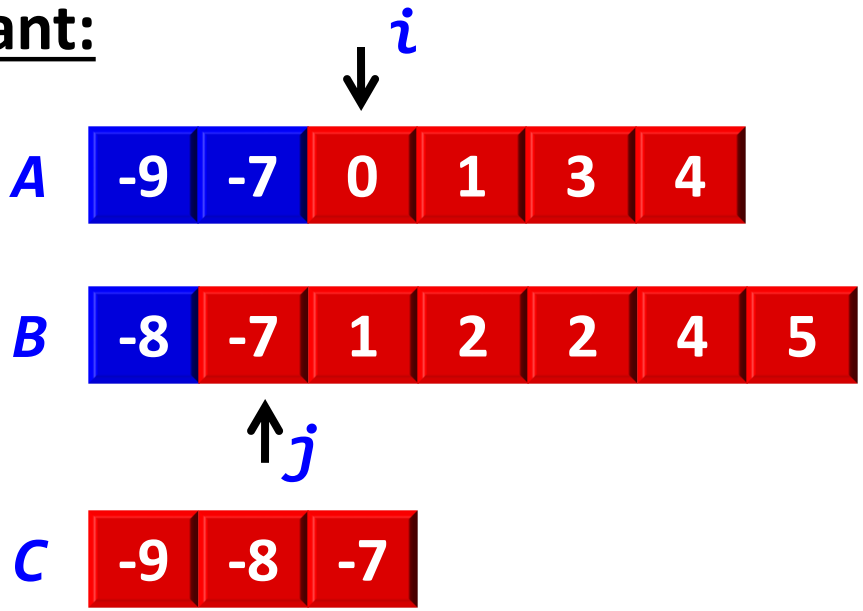
push_back and pop_back operations

```
vector<int> a;    // a.size()=0

a.push_back(3);
a.push_back(5);
a.push_back(8);

// a = [3, 5, 8]; a.size()=3

a.pop_back();

// a = [3, 5]; a.size()=2
```

Vector fusion

```
// Pre:  A and B are sorted in ascending order.
// Returns the sorted fusion of A and B.
vector<int> fusion(const vector<int>& A, const vector<int>& B);
```

Invariant:



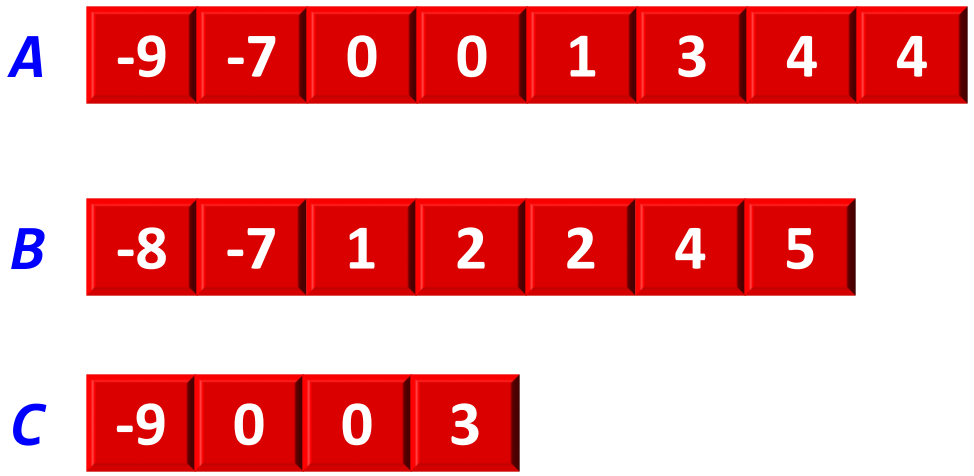
- **C contains the fusion of A[0..i-1] and B[0..j-1]**
- **All the visited elements are smaller than or equal to the non-visited ones.**

Vector fusion

```
// Pre: A and B are sorted in ascending order.
// Returns the sorted fusion of A and B.

vector<int> fusion(const vector<int>& A, const vector<int>& B) {
    vector<int> C;
    int i = 0, j = 0;
    while (i < A.size() and j < B.size()) {
        if (A[i] <= B[j]) {
            C.push_back(A[i]);
            i = i + 1;
        } else {
            C.push_back(B[j]);
            j = j + 1;
        }
    }

    while (i < A.size()) {
        C.push_back(A[i]);
        i = i + 1;
    }
    while (j < B.size()) {
        C.push_back(B[j]);
        j = j + 1;
    }
    return C;
}
```

Difference between two vectors

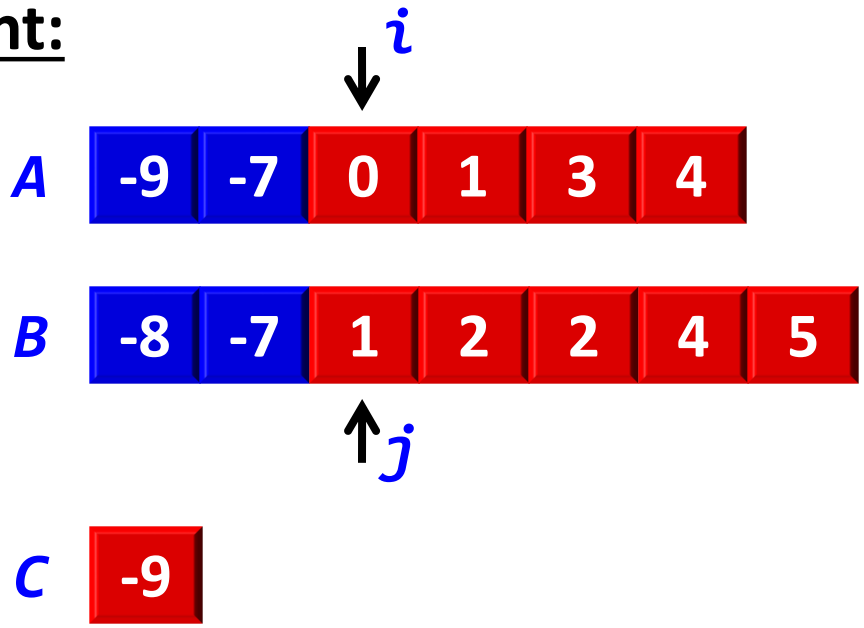
- Design a function that returns an ordered vector C containing the difference between two ordered vectors, A and B (all the elements in A that are not in B).
- Example:



Difference between two vectors

```
// Pre: A and B are sorted in ascending order.
// Returns the sorted difference between A and B.
vector<int> diff(const vector<int>& A, const vector<int>& B);
```

Invariant:



- **C is the difference between A[0..i-1] and B[0..j-1]**
- **All the visited elements are smaller than or equal to the non-visited ones.**

Difference between two vectors

```
// Pre: A and B are sorted in ascending order.
// Returns the sorted difference between A and B.
vector<int> diff(const vector<int>& A, const vector<int>& B) {
    vector<int> C;
    int i = 0, j = 0;
    while (i < A.size() and j < B.size()) {
        if (A[i] == B[j]) i = i + 1;
        else if (A[i] > B[j]) j = j + 1;
        else {
            C.push_back(A[i]);
            i = i + 1;
        }
    }

    while (i < A.size()) {
        C.push_back(A[i]);
        i = i + 1;
    }
    return C;
}
```

STRINGS

Strings

- Strings can be treated as vectors of characters.

- Variables can be declared as follows:
  - string s1;
  - string s2 = “abc”;
  - string s3(10,'x');

- Do not forget to **#include <string>** in the header of your program.

Strings

- Examples of the operations we can do on strings:
  - Comparisons: `==`, `!=`, `<`, `>`, `<=`, `>=`
    Order relation assuming lexicographical order.
  - Access to an element of the string: `s3[i]`
  - Length of a string: `s.length()`

Hybrid animals

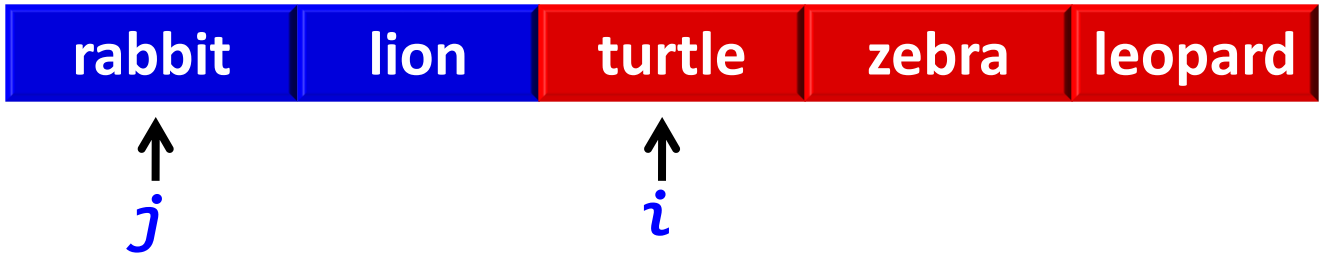
- Given the name of two animals, a hybrid can be formed if the last two letters of one of them coincide with the first two letters of the other one. In this case, the name of the hybrid animal is created by concatenating the two names, except for the first two letters of the second animal.

- Design a procedure that writes the names all possible hybrid animals from a vector of names of animals. We will assume that an animal cannot form a hybrid with itself.

Hybrid animals

```
// Pre:  --
// Post: all the possible hybrid animals from A
//       have been written.
void hybrid_animals(const vector<string>& A);
```

Solution 1:



**Index i traverses the names of the first animal,
whereas index j traverses the names of the second animal.**

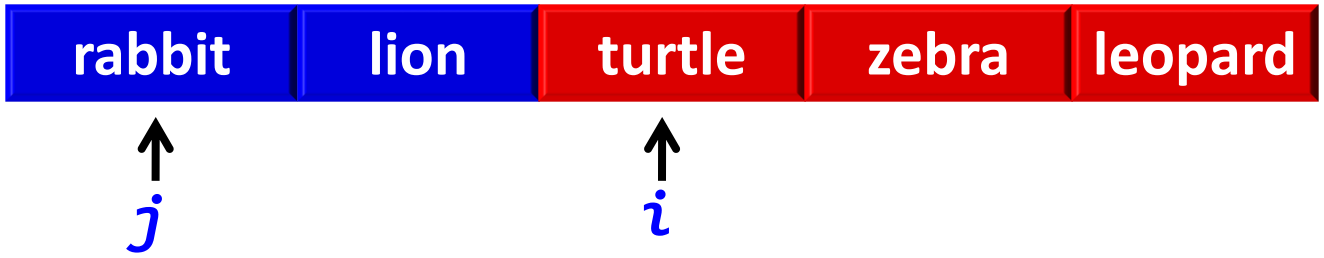
Hybrid animals

```
// Pre:  --
// Post: all possible hybrid animals from A have been written.

void hybrid_animals(const vector<string>& A) { // Solution 1

    // i traverses the names of the first animal
    // j traverses the names of the second animal
    for (int i = 0; i < A.size(); ++i) {
        for (int j = 0; j < A.size(); ++j) {
            // there_is_hybrid checks whether a hybrid
            // between A[i] and A[j] is possible
            if (i != j and there_is_hybrid(A[i], A[j])) {
               // hybrid returns the string with the hybrid name
               cout << hybrid(A[i], A[j]) << endl;
            }
        }
    }
}
```

Hybrid animals

```
// Pre:  --
// Post: all the possible hybrid animals from A have been written.
void hybrid_animals(const vector<string>& A);
```

Solution 2:



Indices i and j traverse the names of the animals and check for hybrids between them in both orders.

Hybrid animals

```
// Pre: --
// Post: all possible hybrid animals from A have been written.

void hybrid_animals(const vector<string>& A) { // Solution 2

    // i and j traverse all pairs of animals (without repetition)
    for (int i = 1; i < A.size(); ++i) {
        for (int j = 0; j < i; ++j) {
            if (there_is_hybrid(A[i], A[j])) {
                cout << hybrid(A[i], A[j]) << endl;
            }
            if (there_is_hybrid(A[j], A[i])) {
               cout << hybrid(A[j], A[i]) << endl;
            }
        }
    }
}
```

Hybrid animals

```
// Pre: --
// Returns true if s1 and s2 can form a hybrid name,
// and false otherwise
bool there_is_hybrid(string s1, string s2) {
    int ls1 = s1.length();
    int ls2 = s2.length();
    if (ls1 < 2 or ls2 < 2) return false;
    else return (s1[ls1-2] == s2[0]) and (s1[ls1-1] == s2[1]);
}

// Pre: s1 and s2 can form a hybrid.
// Returns the hybrid of s1 and s2.
string hybrid(string s1, string s2) {
    string s3(s1.length() + s2.length() - 2, ' ');
    for (int i = 0; i < s1.length(); ++i) s3[i] = s1[i];
    for (int i = 2; i < s2.length(); ++i) {
        s3[i + s1.length() - 2] = s2[i];
    }
    return s3;
}
```

Finding a substring in a string

- String x appears as a substring of string y at position i if y[i...i+x.size()-1] = x

- Example: “tree” is the substring of “the tree there” at position 4.

- Problem: given x and y, return the smallest i such that x is the substring of y at position i. Return -1 if x does not appear in y.

Finding a substring in a string

- Solution: search for such i
- For every i, check whether x = y[i..i+x.size()-1]
- In turn, this is a search for a possible mismatch between x and y: a position j where x[j] ≠ y[i+j]
- If there is no mismatch, we have found the desired i. As soon as a mismatch is found, we proceed to the next i.

Finding a substring in a string

```
// Pre: --
// Returns the smallest i such that x=y[i..i+x.size()-1].
// Returns -1 if x does not appear in y.

int substring(const string& x, const string& y);
```

Finding a substring in a string

```
int substring(const string& x, const string& y) {

    int i = 0;

    // Inv: x is not a substring of y at positions 0..i-1
    while (i + x.size() <= y.size()) {
        int j = 0;
        bool matches = true;
        // Inv: … and x[0..j-1] == y[i..i+j-1]
        while (matches and j < x.size()) {
            matches = (x[j] == y[i + j]);
            ++j;
        }
        if (matches) return i;
        else ++i;
    }

    return -1;
}
```

Finding a substring in a string

- When a mismatch is found (x[j] ≠ y[i+j]) we must proceed to i+1, not to i+j.

- Example:

  X = "papageno", Y = "papapageno"

  - Consider i=0.
  - X[0..3] matches Y[0..3], mismatch at X[4]≠Y[4]
  - If we continue searching at i=4, we miss the occurrence of X in Y.

Finding a substring in a string

- A more compact, but more cryptic solution, with one loop.

```
int substring(const string& x, const string& y) {
    int i, j;
    i = j = 0;
    // Inv: x[0..j-1] == y[i..i+j-1]
    //      and x does not appear in y in positions 0..i-1
    while (i + x.size() <= y.size() and j < x.size()) {
        if (x[j] == y[i + j]) ++j;
        else {
            j = 0;
            ++i;
        }
    }
    if (j == x.size()) return i;
    else return -1;
}
```

Anagrams

- An anagram is a pair of sentences (or words) that contain exactly the same letters, even though they may appear in a different order.

- Example:

  AVE MARIA, GRATIA PLENA, DOMINUS TECUM

  VIRGO SERENA, PIA, MUNDA ET IMMACULATA

- Design a program that reads two sentences that end in '.' and tells whether they are an anagram.

```
// Pre:  the input contains two sentences that end in '.'
// Post: the output tells whether they are an anagram.
```

Anagrams

- A possible strategy for solving the problem could be as follows:
  - First, we read the first sentence and count the number of occurrences of each letter. The occurrences can be stored in a vector.
  - Next, we read the second sentence and discount the appearance of each letter.
  - If a counter becomes negative, the sentences are not an anagram.
  - At the end, all occurrences must be zero.

Anagrams

```
int main() {

    const int N = int('z') - int('a') + 1;
    vector<int> count(N, 0);
    char c;
    cin >> c;

    // Read the first sentence
    while (c != '.') {
        if (c >= 'a' and c <= 'z') ++count[int(c)-int('a')];
        else if (c >= 'A' and c <= 'Z') ++count[int(c)-int('A')];
        cin >> c;
    }
```

Anagrams

```
    // Read the second sentence
    cin >> c;
    bool is_anagram = true;
    while (is_anagram and c != '.') {
        if (c >= 'a' and c <= 'z') c = c - int('a') + int('A');
        if (c >= 'A' and c <= 'Z') { // Discount if it is a letter
            int i = int(c) - int('A');
            --count[i];
            is_anagram = count[i] >= 0;
        }
        cin >> c;
    }

    // Check that the two sentences are an anagram
    int i = 0;
    while (is_anagram and i < N) {
        is_anagram = count[i] == 0;
        i = i + 1;
    }
    cout << is_anagram;
}
```