```cpp
#include <iostream>
#include <set>
#include <string>
#include "Ex02.h"


// ERROR in given code:
// namespace not specified
// Either use 'using' or std:: all over the code when needed.

void printSet(std::set<std::string> s) {
    if (s.size() == 0) {
        std::cout << " No nodes\n";
    }
    else {
        // UNNECESSARY: this iterator can be declared as
        // 'auto' inside the for loop
        std::set<std::string>::iterator it;
        for (it = s.begin() ; it != s.end() ; ++it)
            //cout << ' ' << *it << "\n";              // Two ERRORS
            std::cout << ' ' << *it << '\n';           // CORRECTED
    }
    std::cout << "\n";
}


////////////////////////////////////////////////////////////////////////////
// Exercise 2 (a) Check and correct if necessary
////////////////////////////////////////////////////////////////////////////

// PROBLEMS:
// - std:: is missing in various points (minor problem).
// - It calls itself even when it knows there are no children so there will
//   be at least 2 calls to the function with nullptr where the function
//   will try to call the left and right nodes of nullptr.

// GIVEN CODE:
// void computeParentNodes(Node* n, set<string> & parents) {
//   if (n->left != nullptr || n->right != nullptr)
//       parents.insert(n->name);
//   computeParentNodes(n->left, parents);
//   computeParentNodes(n->right, parents);
// }

// IMPROVED CODE:

using SETStr = std::set<std::string>;

void computeParentNodes(Node * n, SETStr & parents) {
    if (n->left != nullptr) {
        parents.insert(n->name);
        computeParentNodes(n->left, parents);
    }
    if (n->right != nullptr) {
        parents.insert(n->name);
        computeParentNodes(n->right, parents);
    }
}
```

```
59
60 ////////////////////////////////////////////////////////////////////////
61 // Exercise 2 (b) Implement this function
62 ////////////////////////////////////////////////////////////////////////
63
64 // Create a set with all of the names in a subtree given a starting node 'n'
65
66 // It's the same as the first exercise but now the nodes with no children
67 // are included too. It's actually simpler than the previous case.
68
69 // There's no point considering a specific order when reading the tree
70 // because std::set has its own internal ordering.
71
72 using SETStr = std::set<std::string>;
73
74 void computeMembersOfSubTree(Node * n, SETStr & members) {
75     members.insert(n->name);
76     if (n->left)
77         computeMembersOfSubTree(n->left, members);
78     if (n->right)
79         computeMembersOfSubTree(n->right, members);
80 }
81
82
83 ////////////////////////////////////////////////////////////////////////
84 ////////////////////////////////////////////////////////////////////////
85
86 // OBSERVATIONS:
87
88 // std::Set automatically avoids writing the same name multiple times,
89 // resulting in simple and clean code where attempts to insert the same node
90 // name have no effect.
91 // At the same time, this creates a problem though:
92 // if there are people in the family tree with the same name, that name will
93 // be saved only once, with loss of information.
94 // Switching to std::multiset solves the second problem but code must be more
95 // complex in the computation of a subtree as the code relies on the fact that
96 // repeated values (when passing through the same node while moving up and down
97 // the tree) are ignored.
98
99 // Another problem with sets and multisets is that these structures internally
100 // sort elements so when printing out, names will be presented in alphabetical
101 // order regardless of how data has been acquired.
102
103 // std::unordered_multiset doesn't follow an alphabetical order so it makes a
104 // bit more sense, but it groups together repeated values when printing out so
105 // results are a bit unpredictable.
106
107 // The best alternative is std:deque. It is a FIFO structure that makes possible
108 // to print out the tree in a way that is significant (although functions have
109 // to be more complex to acquire data correctly) and allows access to elements
110 // without having to pop them out (unlike simple std::queue).
111
112 // The function 'printSet' is exactly the same regardless the data container used
113 // (set, unordered_set, deque) so it can be templated.
```