CS3520 Programming in C++
Iterators

Ken Baclawski
Fall 2016

Outline

- Solution to Assignment #1
- Solution to Assignment #2
- Assignment #4
- Iterators
- Sample program: tree3

Solution to Assignment #1

Solution to Assignment #1

```
#include <iostream>

/**
 * @namespace asst01 Solution to Assignment #1.
 * Read 5 lines from the standard input.
 * Concatenate the lines in reverse separated by
 * spaces.  Print the concatenated lines twice
 *
 * @author Ken Baclawski
 */

/**
 * Main program for the solution to Assignment #1.
 * @return The status code.  Normal status is 0.
 */
int main() {
```

Solution to Assignment #1

```
// Read 5 input lines.

std::string line1;
std::getline(std::cin, line1);
std::string line2;
std::getline(std::cin, line2);
std::string line3;
std::getline(std::cin, line3);
std::string line4;
std::getline(std::cin, line4);
std::string line5;
std::getline(std::cin, line5);

// Concatenate in reverse.

std::string result = line5 + " " + line4 +
  " " + line3 + " " + line2 + " " + line1;
```

Solution to Assignment #1

```
   // Print twice.

   std::cout << result << std::endl;
   std::cout << result << std::endl;

   // Return the status.

   return 0;
}
```

Solution to Assignment #2

Solution to Assignment #2

```
#include <vector>
#include <iostream>
using namespace std;

/**
 * @namespace asst02 Solution to Assignment #2.
 * Read words from the standard input.  Print the
 * longest word and the shortest word in the
 * input.  If there are no words in the input
 * print nothing and return status 1.  If more
 * than one word has the same length as the
 * shortest word, then print the first one.
 * Similarly for the longest word.
 *
 * @author Ken Baclawski
 */
```

Solution to Assignment #2

```
/**
 * Main program for the solution to Assignment #2.
 * @return The status code.  Normal status is 0.
 * If there is no input then return status 1.
 */
int main() {

  // Declare the word being read, and the longest
  // and shortest word.

  string word;
  string longestWord;
  string shortestWord;
```

Solution to Assignment #2

```
// Check that there is at least one word.

if (cin >> word) {
  longestWord = word;
  shortestWord = word;
} else {
  return 1;
}

// Compare all other words.

while (cin >> word) {
  if (word.length() > longestWord.length()) {
    longestWord = word;
  }
  if (word.length() < shortestWord.length()) {
    shortestWord = word;
  }
}
```

Solution to Assignment #2

```
   // Print the results.

   cout << "Shortest word is " << shortestWord << endl;
   cout << "Longest word is " << longestWord << endl;

   // Return the status.

   return 0;
}
```

Assignment #4

Requirements

- Design a class named `PositiveList` and a main program that tests it
- The class has three public methods:
  - `addPositive`
  - `removePositive`
  - `computeAverage`
- The constructor is the default constructor
  - You do not have to declare a default constructor
  - The compiler will generate it automatically

addPositive

- The method allows one to add a new positive double to your PositiveList object
- If the double being added is not positive, then the double is ignored and the method does not change the doubles stored in your PositiveList object and 0.0 is returned
- If the double being added is positive but is equal to one that is already stored in your PositiveList object, then the double is ignored and the method does not change the doubles stored in your PositiveList object and 0.0 is returned
- Otherwise, the double is stored in your PositiveList object and the double that was stored is returned
- How you store the doubles in PositiveList is up to you

removePositive

- The method allows one to remove a positive double from your PositiveList object
- If the double to be removed is not equal to one that is stored in your PositiveList object, then the double is ignored and the method does nothing except return 0.0
- Otherwise, the double that is currently stored and is equal to the double to be removed is removed from the doubles that are stored in your PositiveList object and the double is returned by the method
- How you remove a double from the doubles stored in PositiveList is up to you

`computeAverage`

- Compute the average value of all the positive doubles that are stored in your PositiveList object and return the average value
- If there are no positive doubles stored in your PositiveList object, then throw an exception with the message “There are no positive numbers”

Main program

- Your main program should test the following:
  - Create a `PositiveList` object
  - Call `addPositive` with both positive and negative numbers
    - Print what `addPositive` returns each time
  - Call `removePositive` with both positive and negative numbers
    - Print what `removePostitive` returns each time
  - Call `computeAverage`
    - Print what `computeAverage` returns
    - If `computeAverage` throws an exception, then print the exception message
- Your `PositiveList` program will be tested by the TA both with your main program and a different one

Assignment #4 Submitting and Grading

- Submit `PositiveList.h`, `PositiveList.cpp` and `Main.cpp`
- Grading:
  - Compile with no errors or warnings (20%)
  - Correct execution on test program (30%)
  - Documentation (20%)
  - Correct style (30%)

Iterators

Purpose of Iterators

- Fundamental Concept for the Standard Template Library
- Uniform mechanism for specifying:
  - A container
  - An element in a container
  - Operations that are independent of the container

Iterators vs Indexes

- Advantages of iterators
  - More efficient when used with STL algorithms
  - Fewer parameters than indexes
  - More opportunities for reusability
  - Can be used for containers that cannot be indexed
- Disadvantages of iterators
  - Unfamiliar syntax
  - Modifying the container requires reconstructing any iterators
  - No safer than indexes

Iterating over a container

```
for (int i = 0; i < container.size(); ++i) {
  ...
  // container[i] is the object being operated on
  ...
}
```

```
for (auto iter = container.begin();
     iter != container.end(); ++iter) {
  ...
  // (*iter) is the object being operated on
  ...
}
```

```
for (const auto& element : container) {
  ...
  // element is the object being operated on
  ...
}
```

Comparison of Iteration Mechanisms

	Available for all container types	Safety	STL Library Compatibility	Ease of Use
`for (int i = 0; i < container.size(); ++i) { ... // container[i] is the object being operated on ... }`	No	No	No	Yes
`for (auto iter = container.begin(); iter != container.end(); ++iter) { ... // (*iter) is the object being operated on ... }`	Yes	No	Yes	No
`for (const auto& element : container) { ... // element is the object being operated on ... }`	Yes	Yes	No	Yes

```
for (int i = 0; i < container.size(); ++i) {
  ...
  // container[i] is the object being operated on
  ...
}
```

```
for (auto iter = container.begin();
     iter != container.end(); ++iter) {
  ...
  // (*iter) is the object being operated on
  ...
}
```

```
for (const auto& element : container) {
  ...
  // element is the object being operated on
  ...

}
```

The tree3 Package

Requirements

- Non-Recursive Tree Structure.
- The tree structure can be constructed in several ways
  - Default tree
  - Input stream using lisp-style format
  - Programmatically
- The tree can be printed
- All subtrees with a specified label can be printed

Classes

- Tree class
  - Methods for construction, and printing
  - Data member is vector of Nodes
  - Storage and printing are not recursive
- Node class
  - Data members are a label, the depth, whether the node is valid and two iterators
  - Methods for printing, getting label and depth, testing validity, invalidating and setting the iterators

Tree Interface

Default constructor: Used when a Tree variable is defined like this: `Tree tree;`

Construct a tree from an input stream

Add another node to the tree

Delete all the nodes of the tree

Overloaded method name

```
class Tree {
public:
  Tree();

  Tree(std::istream& in);

  void addNode(const std::string& label, int depth);

  void clear();

  void print(std::ostream& out) const;

  void print(std::ostream& out, const std::string& label) const;
```

Default Constructor

- Used when a variable is declared with no parameters

```
Tree tree;
```

- The compiler generates a default constructor if no other constructors are declared in the interface
  - The generated default constructor initializes all data members with the initializations specified in the interface
- If there is a non-default constructor, one can reinstate the generated default constructor

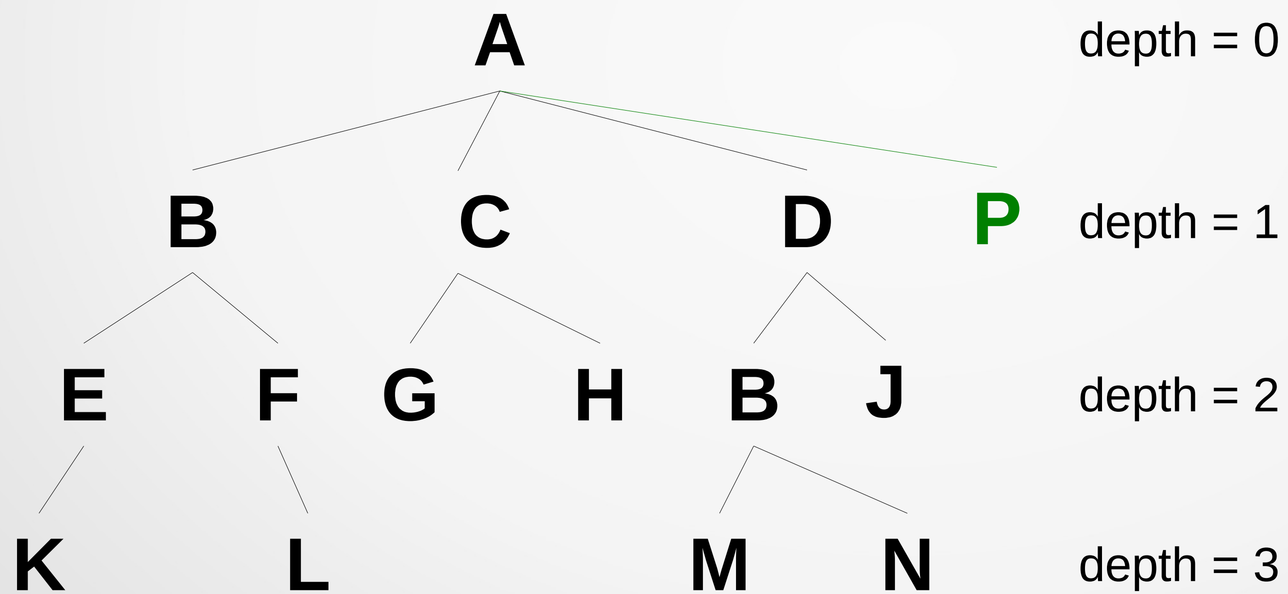
The default constructor for tree3

```
Tree tree;
```

root depth = 0

Adding a node to a tree

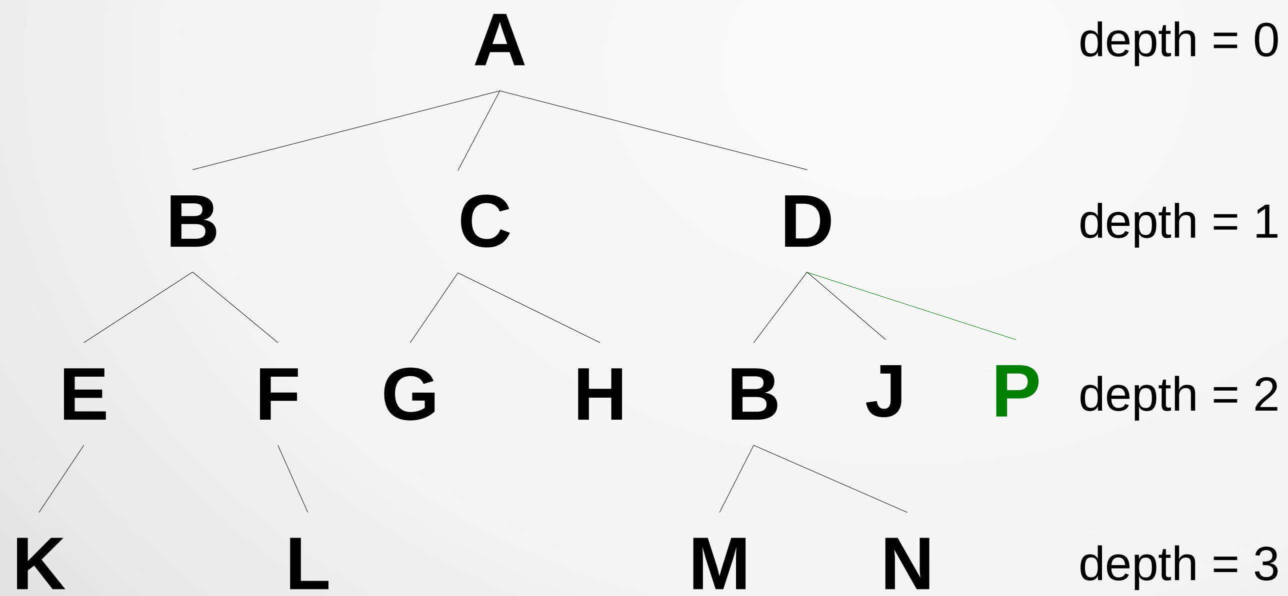
```
tree.addNode("P", 1);
```



The added node is shown in green

Adding a node to a tree

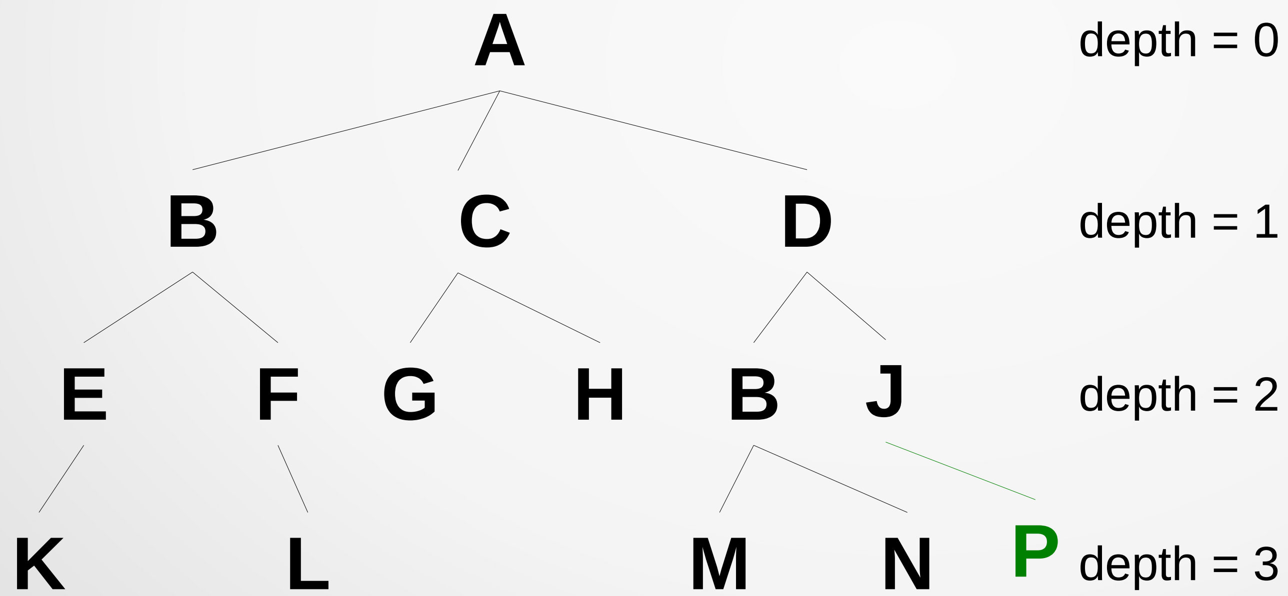
```
tree.addNode("P", 2);
```



The added node is shown in green

Adding a node to a tree

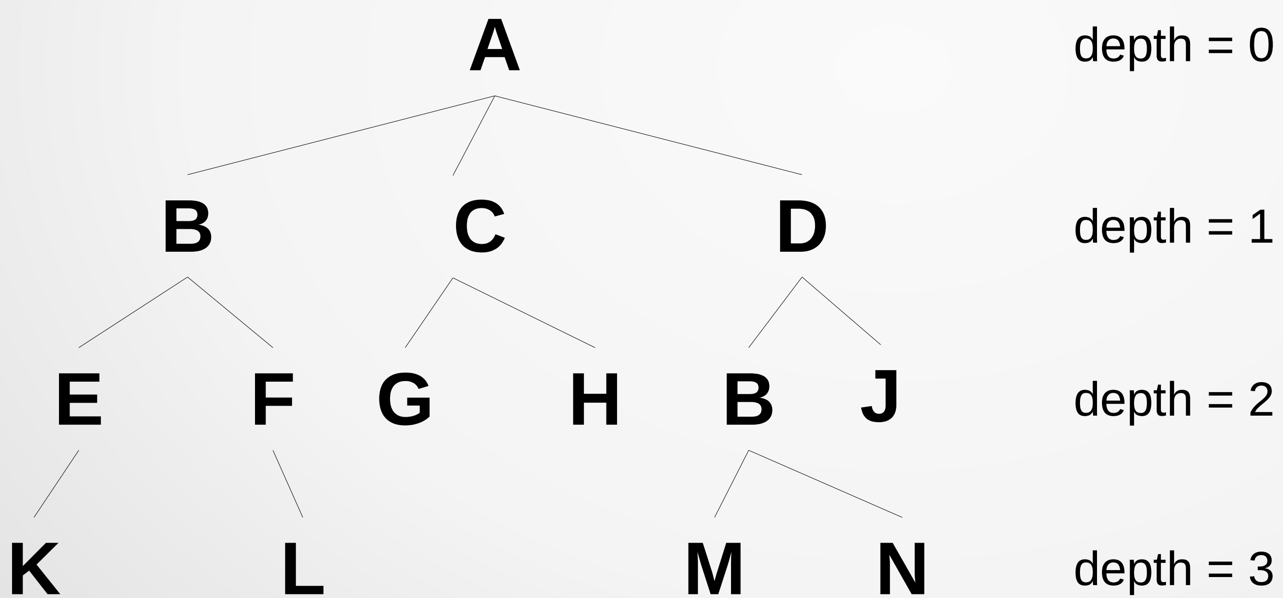
```
tree.addNode("P", 3);
```



The added node is shown in green

Adding a node to a tree

```
tree.addNode("P", 4);
```

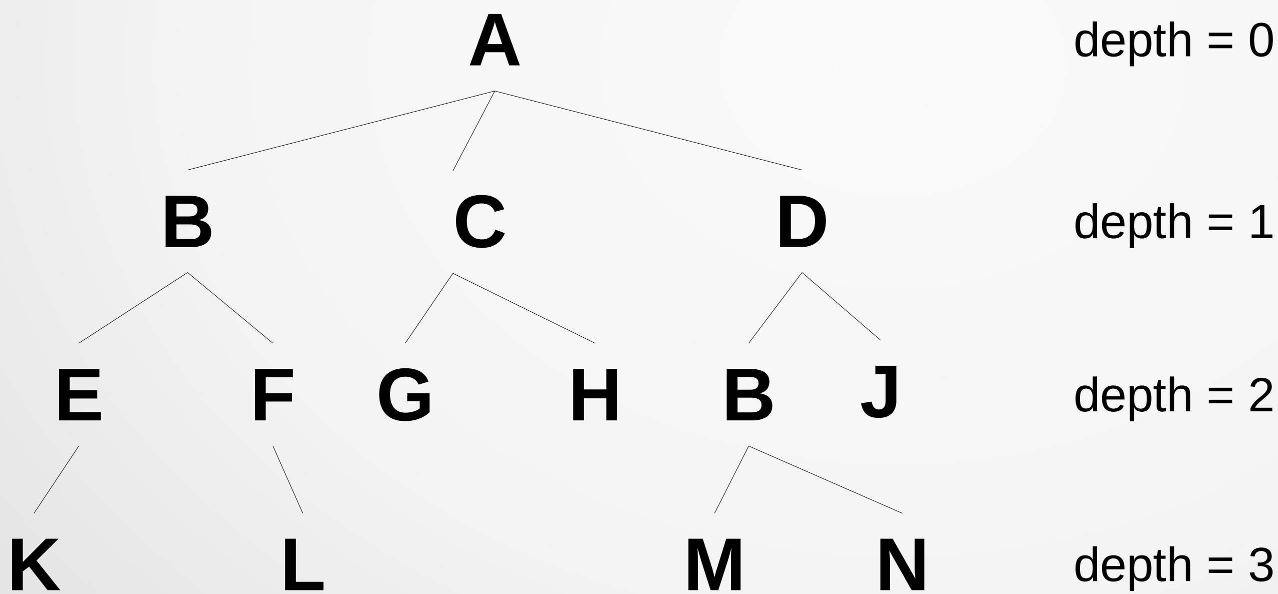


An exception is thrown

Printing the tree

```
tree.print(cout);
```

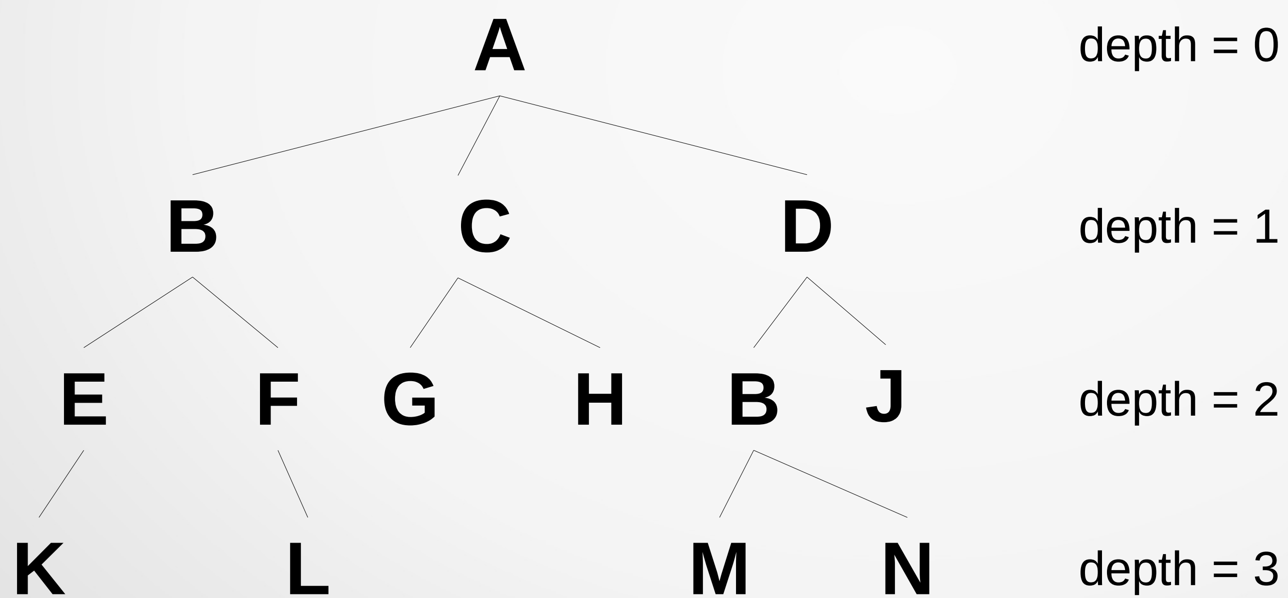
(A(B(E(K))(F(L)))(C(G)(H))(D(B(M)(N))(J)))



Printing subtrees

```
tree.print(cout, "B");
```

```
(B(E(K))(F(L)))
(B(M)(N))
```



Tree private members

```
class Tree {
...
private:
  std::vector<Node> nodes_;

  void readTree(std::istream& in);

  void setNodeLimits();

  static std::string readWord(std::istream& in);
}
```

Array list of tree nodes in depth-first search order

Read a tree written in lisp-style format

Set the two iterators in every node

Same as in tree1 and tree2

Node public interface

```
public:
  Node() = default;

  Node(const std::string& label, int depth);

  void setLimits(const std::vector<Node>::iterator& begin,
        const std::vector<Node>::iterator& end);

  std::string getLabel() const;

  int getDepth() const;

  bool isValid() const;

  void print(std::ostream& out) const;

  void invalidate();
```

Reinstate the default constructor

Construct a node

Set the iterators for use during the print methods

The getters for label, depth and valid

Non-recursive print method

Explicitly invalidate a node

Tree private members

```
private:
  const std::string label_;

  const int depth_ = 0;

  bool valid_ = false;

  std::vector<Node>::iterator begin_;

  std::vector<Node>::iterator end_;
};
```

Same as tree1

Flag for marking the node as invalid

Beginning of the subtree
Points to the node in the list of all nodes

End of the entire tree
Not the end of the subtree
This ensures that iterators do not exceed the end of the array list

The `bool` type

- A variable of type `bool` can only have values `true` or `false`.
- The result of a logical expression has type `bool`
  - Logical and uses `&&`
  - Logical or uses ||
  - Logical not uses `!`
- Binary logical operators are short-circuit operators
  - Overloaded logical operators are not short-circuit
- Comparison operators also produce values of type `bool`
- In an `if` or `while` condition, a `bool` variable *must* be used by itself.
  - Never compare a `bool` variable with `true` or `false`.

Tree constructors

```
Tree::Tree() {
  nodes_.push_back(Node("root", 0));
  setNodeLimits();
}
```

The iterators cannot be set until the entire tree has been constructed

```
Tree::Tree(istream& in) {
  readTree(in);
  setNodeLimits();
}
```

```
void Tree::clear() {
  nodes_.clear();
}
```

Most containers will have a clear method that removes all of the elements in the container

Construction and Destruction

- Sometimes initialization cannot be completed until the object is completely constructed
  - This was not necessary in the Tree class
  - When it is necessary, the technique that works best is the Factory design pattern to be discussed later
- The destruction of an object is done by a special method called the destructor
  - If not destructor is defined, then one is generated
  - So far the generated destructor was sufficient
- Common container methods
  - clear() removes all of the elements
  - reserve(n) is a hint to the container that there will be at least n elements
  - resize(n) removes some of the elements or adds default elements

Iteration for printing

```
void Tree::print(ostream& out, const string& label) const {
  for (vector<Node>::const_iterator iter =
         nodes_.begin();
       iter != nodes_.end(); ++iter) {
    if (iter->getLabel() == label) {
      iter->print(out);
      out << endl;
    }
  }
  /* Alternative iteration syntax
  for (const Node& node : nodes_) {
    if (node.getLabel() == label) {
      node.print(out);
      out << endl;
    }
  }
  */
}
```

Iteration for printing

```
void Tree::print(ostream& out, const string& label) const {
  for (vector<Node>::const_iterator iter =
         nodes_.begin();
       iter != nodes_.end(); ++iter) {
    if (iter->getLabel() == label) {
      iter->print(out);
      out << endl;
    }
  }
  /* Alternative iteration syntax
  for (const Node& node : nodes_) {
    if (node.getLabel() == label) {
      node.print(out);
      out << endl;
    }
  }
  */
}
```

- The alternative for loop syntax has the same meaning
  - Much more readable

- The full name of the iterator was used
  - One could use `auto` instead
- `const_iterator` ensures that the nodes are not altered
  - This is redundant since the whole method is `const`
- The node is `(*iter)`
  - This is called *dereferencing*
  - Parentheses are necessary because of the low precedence of the operator
  - `iter->getLabel()` is the same as `(*iter).getLabel()`
  - `iter->print(out)` is the same as `(*iter).print(out)`

Iteration in setNodeLimits

```
void Tree::setNodeLimits() {
  for (auto iter = nodes_.begin(); iter != nodes_.end(); ++iter) {
    iter->setLimits(iter, nodes_.end());
  }
}
```

- The alternative for loop syntax cannot be used
  - The iterator *itself* is being used as a parameter
- The iterator is like a file shortcut (or symbolic link)
  - The iterator object itself is `iter`
  - The object that is referenced by the iterator is `(*iter)`
  - The asterisk *dereferences* the iterator

Iterator Arithmetic

- `iter + n` references the object `n` positions later in the container
- `iter - n` references the object `n` positions earlier in the container
- `iter2 – iter1` is the difference of the locations of the iterators
  - The two iterators must be for the same container
- `iter[n]` is the same as `(*(iter + n))`
- The performance of these operations depends on the container type
  - Fast for an array list like the `<vector>` library
  - Slow for a linked list like the `<list>` library
- Iterator arithmetic is **not** safe
  - Using an iterator outside the range of the container is **undefined**

The addNode method

```
void Tree::addNode(const string& label, int depth) {
  if (nodes_.empty()) {
    // If the tree is empty, then the new node is the root.
    nodes_.push_back(Node(label, depth));
    setNodeLimits();
  } else {
    // If the tree is nonempty, then the depth
    // must be within limits for the depth to be
    // meaningful for the new node.
    if (depth <= nodes_.front().getDepth()) {
      throw domain_error("Attempt to add another root node to a tree");
    } else if (depth > nodes_.back().getDepth() + 1) {
      throw domain_error("Attempt to add a node with too great a depth");
    } else {
      // If the depth is okay, then add the new
      // node and set the node limits.
      nodes_.push_back(Node(label, depth));
      setNodeLimits();
    }
  }
}
```

The ends of a container

- The first and last elements of the container are `front()` and `back()`
  - Using these methods on an empty container is *undefined*
  - This is reason for the name `push_back`
- `begin()` returns an iterator referencing the first element
  - Meaningful even when the container is empty
- `front()` returns a direct reference to the first element
- `end()` returns an iterator that is one location after the last element
  - Does not reference an element in the container
- `back()` returns a direct reference to the last element

The `readTree` method

- The method is not recursive
  - Recursive methods can always be changed to non-recursive
  - Faster and requires less memory space
  - The required data in the execution stack can be stored in a vector or stack structure instead
  - The most subtle kind of data in the execution stack are the return address and return value
- Loop invariants are the key to programming the `readTree` method.

The `readTree` method

- Each node begins with a left parenthesis
- The outer loop invariant is: a left parenthesis has been read from the input stream
- The inner loop invariant is: the value of the `word` variable is a (left or right) parenthesis
- When the inner loop finishes the value of the `word` variable is a left parenthesis which proves the outer loop invariant

```
void Tree::readTree(istream& in) {
  ...
  // Read the first left parenthesis
  ...
  for (;;) {
    ...
    // Read the node label
    ...
    nodes_.push_back(Node(label, depth));
    while (word == ")") {
      ...
      // Get the next left parenthesis
      ...
    }
    ...
  }
}
```

The `readTree` method

- The depth is determined by the parentheses
  - Each left parenthesis increases the depth by 1
  - Each right parenthesis decreases the depth by 1
- When the depth is equal to the initial depth, the parentheses balance and the tree has been completed

```
void Tree::readTree(istream& in) {
  int depth = -1;
  ...
  // Read the first left parenthesis
  ...
  ++depth;
  for (;;) {
    ...
    // Read the node label
    ...
    nodes_.push_back(Node(label, depth));
    while (word == ")") {
      --depth;
      if (depth == -1) {
        return;
      }
      ...
      // Get the next left parenthesis
      ...
    }
    ++depth;
  }
}
```

Node methods

```
Node::Node(const string& label, int depth) :
  label_(label), depth_(depth) {}

void Node::setLimits(const vector<Node>::iterator& begin,
            const vector<Node>::iterator& end) {
  begin_ = begin;
  end_ = end;
  valid_ = true;
}
```

- If the `label` parameter were of type `string` rather than type `const string&` then the string would be copied twice: once to pass the parameter and again to initialize `label_`
- The assignment `begin_ = begin;` assigns the iterator, not the referenced objects. To assign the referenced objects one must dereference like this: `*begin_ = *begin;`

Getters

```
bool Node::isValid() const {
  return valid_;
}
string Node::getLabel() const {
  return label_;
}
int Node::getDepth() const {
  return depth_;
}
```

- Examples of getters
  - Special case of observer method
- Boolean getters usually begin with some word like “is”
  - Other languages end such a function with a question mark

Setters

```
void Node::invalidate() {
  valid_ = false;
}
```

- Example of a setter
- Setters normally have a parameter to allow setting the value to another value
  - The setter will check that the new value is acceptable

Print method

- Non-recursive
- This is where the `begin_` iterator is used
- The loop invariant is: the `depth` variable is equal to the depth of the node referenced by `iter`
- Note the use of iterator arithmetic

  `++iter`

  `iter + 1`

```
void Node::print(ostream& out) const {
  int depth = depth_ - 1;
  for (auto iter = begin_;; ++iter) {
    ...
    while (depth >= iter->depth_) {
      ...
      --depth;
      // Close any open expressions
      ...
    }
    while (depth < iter->depth_) {
      ...
      ++depth;
      // Open new expressions
      ...
    }
    // The depth is now equal to iter->depth_
    out << " " << iter->getLabel();
    if (iter + 1 == end_) {
      ...
      return;
    }
  }
}
```

Print method

- Check that begin_ and end_ are valid
- The subtree is complete when the depth returns to the initial value
  - No assumption that the root has depth 0
- The second while loop will only execute once
  - However, it would still work even if there were redundant parentheses like this:

    (A ((B)))
  - A has depth 0 and B has depth 2
- If the iterator reaches the end then close any open expressions
  - This only happens when the whole tree is printed
  - Important to ensure that the iterator is always valid because using an out of range iterator is *undefined*

```
void Node::print(ostream& out) const {
  if (!isValid()) {
    throw domain_error("Attempt to”
      “ print an invalid tree");
  }
  int depth = depth_ - 1;
  for (auto iter = begin_;; ++iter) {
    while (depth >= iter->depth_) {
      out << " )";
      --depth;
      if (depth < depth_) {
        return;
      }
    }
    while (depth < iter->depth_) {
      out << " (";
      ++depth;
    }
    out << " " << iter->getLabel();
    if (iter + 1 == end_) {
      while (depth >= depth_) {
        out << " )";
        --depth;
      }
      return;
    }
  }
}
```

The main program

- The main program illustrates some of the features of the Tree and Node classes
- The main program is not a complete test of either class

```
int main() {
  try {

    // Various tests go here...



  } catch (const exception& e) {
    cerr << e.what() << endl;
    return 1;
  }
  return 0;
}
```

First test of tree3

- Test of the default constructor
- Test of the first print method

```
// Create a default tree and print it.

Tree tree1;
tree1.print(cout);
```

Output: `( root )`

Second test of tree3

- Test of the clear method
- Test of programmatic construction of a tree

```
// Clear the tree and add some nodes.

tree1.clear();
tree1.addNode("another root", -4);
tree1.addNode("a child", -3);
tree1.addNode("another child", -3);
tree1.print(cout);
```

Output: `( another root ( a child ) ( another child ) )`

Third test of tree3

- Test of constructing from the standard input
- Test of second print method

```
Tree tree2(cin);
// Print the tree on the standard output
tree2.print(cout);
// Print subtrees with label B
tree2.print(cout, "B");
```

Output:

```
( A ( B ( E ( K ) ) ( F ( L ) ) ) ( C ( G ) ( H ) ) ( D ( B ( M ) ( N ) ) ( J ) ) )
( B ( E ( K ) ) ( F ( L ) ) )
( B ( M ) ( N ) )
```

Fourth test of tree3

- Using the standard input again should throw an exception because the input does not have another tree

```
// This should throw an exception
Tree tree3(cin);
```

Output on standard error: `Invalid expression`

Next Class

- Library Algorithms Part 1
- UML Class Diagrams