CS3520 Programming in C++ Library Algorithms

Ken Baclawski
Fall 2016

Outline

- Library Algorithms
  - Overview of the algorithms
  - Naming conventions
- The rocket package
  - Use of library algorithms
  - Operator overloading
  - Enumerations
  - The typedef declaration
  - Tuples and pairs
- UML Class Diagrams
- Computational Complexity
- Assignment #5

© 2016 Ken Baclawski All Rights Reserved

Library Algorithms

The algorithms library

- Sequence Operations
  - Observers
  - Modifiers
- Partitioning
- Sorting
- Operating on sorted data
  - Searching
  - Merging
- Heap (priority queue)
- Extremes
- Permutations

© 2016 Ken Baclawski All Rights Reserved

Algorithms

- Designed for ranges of elements
  - Specified with iterators
- Main container object is unchanged
  - Elements of the container are rearranged
  - No change to size or storage allocation
- Optional output container
  - Output container size is changed
- Use of algorithms is not safe
  - Always check to ensure that parameters are valid

© 2016 Ken Baclawski All Rights Reserved

Inserters

- Inserters are iterators that insert additional objects into a container
- Inserters are used for output to a container in some of the functions in the algorithm library
- The most common inserter is `back_inserter` which uses `push_back` to insert objects into its container
- If a container c is to be used as the output container of a function, the argument should be `back_inserter(c)`

Predicates

- Many algorithms have an optional predicate parameter.
  - Sometimes there are two different function names: one with and one without a predicate parameter
  - Other times the function name is overloaded and the predicate is an optional parameter
- A *predicate* is a function such that
  - Returns boolean
  - Observer (i.e., has no side-effects)
  - No arguments are modified
- Well suited to lambda expressions

Algorithm Name Conventions

`_if` means that a predicate will be used rather than comparison with a value

`_copy` means that the input range will be left unchanged and the modified range is written to an output container

`stable_` means that a partition or sort operation maintains the original order when elements are equal

`_n` means the input range is specified by its size rather than specifying by the end of the input range

`is_` tests a sequence for a property

`_until` finds the first place in a sequence where a property does not hold

`set_` is used for set operations (e.g., union, intersection)

Specifying the range or ranges

- When there is a single range
  - Two iterator form: (begin iterator, end iterator)
  - Single iterator form: (begin iterator, size n)
    - Indicated with "`_n`" in the name
    - **Important**: Check that the container is large enough for n
- When there are two ranges of the same size
  - Only the first end is specified:

    (begin1 iterator, end1 iterator, begin2 iterator)
  - **Important**: Check that the two ranges are the same size! The algorithm does not check this. In fact, there is no way that it could check it.

Sequence Observing Algorithms

- Test a predicate for one range
  - `all_of` Universal Quantifier
  - `any_of` Existential Quantifier
  - `none_of` Negation of Existential Quantifier
- Test equality for two ranges
  - `equal` Pairwise Equality
  - `mismatch` Find first mismatch
  - `is_permutation` Permutation
- Search for subsequence
  - `search` Find first occurrence of subsequence
  - `find_end` Find last occurrence of subsequence
- Search for one element
  - `find` Find specified value
    - Variations: `find_if`, `find_if_not`
  - `search_n` Find n adjacent occurrences of a value
  - `find_first_of` Find value from a set
- Search for adjacent equal pair
  - `adjacent_find` Find first adjacent equal pair
- Apply function
  - `for_each` Same as for loop on the range
- Count occurrences
  - `count` Count elements equal to one value
  - `count_if` Count elements satisfying predicate

Sequence Modifying Algorithms

- Copy to output container

  copy, copy_n, copy_if, copy_backward
- Swap two values

  swap, iter_swap, swap_ranges
- Reverse values in range

  reverse, reverse_copy
- Move to another range

  move, move_backward
- Replace old value with new value

  replace, replace_if, replace_copy, replace_copy_if
- Fill with a value

  fill, fill_n
- Generate new values

  generate, generate_n
- Remove old value and shift

  remove, remove_if, remove_copy, remove_copy_if
- Remove adjacent duplicates

  unique, unique_copy
- Rotate left

  rotate, rotate_copy
- Random suffle
  - random_shuffle, suffle

© 2016 Ken Baclawski All Rights Reserved

Sequence Algorithms

- Most of the sequence algorithms are easily programmed with a `for` loop
- There are a few exceptions
  - Search for subsequence
  - `is_permutation` function
- The main advantage of the library functions is that they can be used in combination with each other

Partitioning

- Rearrange the container into two parts using a predicate

  `partition`, `partition_copy`, `stable_partition`

- Verify that a container is partitioned

  `is_partition`

- Find the location where the first group ends, assuming that the range has been partitioned

  `partition_point`

- More than two groups can be in the partition by calling a partition function multiple times
- Sorting may be regarded as form of partitioning with more than just two groups

© 2016 Ken Baclawski All Rights Reserved

Sorting

- By far the most important algorithms in the library
- Only the less than comparison is performed
  - Incomparable elements are considered equal
- Sorting is done in place by rearranging elements, unless copied
- Full sort

  `sort`, `stable_sort`
- Sort only elements less than a specified element

  `partial_sort`
- Sort to an output container (possibly smaller)

  `partial_sort_copy`
- Verify that a container is sorted

  `is_sorted`, `is_sorted_until`
- Order statistic

  `nth_element`

© 2016 Ken Baclawski All Rights Reserved

Operating on Sorted Data

- All of these algorithms assume that the data has been sorted or partitioned
- Bounds

  lower_bound, upper_bound
  - Useful for B-trees
- Search

  binary_search, equal_range
- Merge two sorted ranges

  merge, inplace_merge
- Set operations

  includes, set_union, set_intersection, set_difference, set_symmetric_difference
  - They all write to an output container, except the first which is boolean

© 2016 Ken Baclawski All Rights Reserved

Heap Operations

- Interface for classical heap structure
- Only useful for applications with large heaps
  - Uses less memory than a binary tree
  - Slower than a binary tree: takes about 1.5 to 2 times as long
- Interface is

  make_heap, push_heap, pop_heap
- Verify heap property

  is_heap, is_heap_until
- Sort a heap

  sort_heap

© 2016 Ken Baclawski All Rights Reserved

Extremes

- Computing extremes when not sorted
- Extreme value of two elements or list

  min, max, minmax
- Extreme element of a range

  min_element, max_element, minmax_element

Lexicographic Ordering

- Sequences are in *lexicographic order* when the first elements where they differ are in order
  - Also called alphabetical or dictionary order when applied to sequences of characters
- Verify lexicographic order

  lexicographical_compare
- Rearrange to be next or previous in lexicographic order

  next_permutation, prev_permutation

© 2016 Ken Baclawski All Rights Reserved

The rocket package

Requirements

- Store information about rockets
  - Name, payload, manufacturer
- Sort by any of the fields
- Test whether two lists are equal as lists or as multisets
- Determine the category of the rocket
- Print a table of the rockets in the list
  - One row per rocket
  - Optionally grouped by category

© 2016 Ken Baclawski All Rights Reserved

The files of the rocket package

- The Rocket class: `Rocket.h` and `Rocket.cpp`
  - An instance is one rocket
- The Rockets class: `Rockets.h` and `Rockets.cpp`
  - An instance is a list of rockets
- The main program with some test code: `Main.cpp`

© 2016 Ken Baclawski All Rights Reserved

Rocket Header

```
class Rocket {
public:
  Rocket(const std::string& name,
    double payload,
    const std::string& manufacturer);

  std::string getName() const;
  double getPayload() const;
  std::string getManufacturer() const;

...

private:
  std::string name_;
  double payload_ = 0.0;
  std::string manufacturer_;
};
```

This is essentially just a database record:
- constructor
- getters
- data members

There are no setters

Rocket Header

```
...
  bool operator==(const Rocket& other) const;

  enum class Category {
    /** Payload up to 2 Mg. */ Small,
    /** Payload up to 20 Mg. */ Medium,
    /** Payload up to 50 Mg. */ Heavy,
    /** Payload above 50 Mg. */ SuperHeavy };
  Category getCategory() const;

  std::string getHtmlTableEntries() const;
...
```

This is an overloaded operator

Example of a getter without a data member

Show the rocket information as a line in an HTML table

Enumerations are just named values. In this case, the values are sequential integers. However, the actual values are irrelevant. Using `enum class` rather than just `enum`, ensures type safety.

Overloading Operators

- The name of an operator as a method is `operator` op
- The equality operator is `operator==`

The expression `x == y` can also be written like this:

```
x.operator==(y)
```

Enumeration

- A distinct type with a restricted set of values
- The values can be explicitly specified or generated implicitly
- Values have names
- The underlying type is an integer type
- One should specify an enumeration with `enum class` rather than the older C notion of an `enum`.
  - The C notion is not type safe
  - The C++ notion is type safe

Rocket Source

```
#include <string>
#include "Rocket.h"
using namespace std;
using namespace rocket;

Rocket::Rocket(const string& name, double payload,
               const string& manufacturer)
  : name_(name), payload_(payload),
    manufacturer_(manufacturer) {}

string Rocket::getName() const {
  return name_;
}
double Rocket::getPayload() const {
  return payload_;
}
string Rocket::getManufacturer() const {
  return manufacturer_;
}
...
```

These are the traditional database record constructor and getters

© 2016 Ken Baclawski All Rights Reserved

Rocket Source

```
...
bool Rocket::operator==(const Rocket& otherRocket) const {
  return name_ == otherRocket.name_
    && payload_ == otherRocket.payload_
    && manufacturer_ == otherRocket.manufacturer_;
}
Rocket::Category Rocket::getCategory() const {
  if (payload_ < 2.0) {
    return Category::Small;
  } else if (payload_ < 20.0) {
    return Category::Medium;
  } else if (payload_ < 50.0) {
    return Category::Heavy;
  } else {
    return Category::SuperHeavy;
  }
}
string Rocket::getHtmlTableEntries() const {
  return "<td>" + name_ + "</td><td>"
    + to_string(payload_) + "</td><td>"
    + manufacturer_ + "</td>";
}
```

The equality operator must be explicitly defined since C++ does not generate it automatically

This is really the definition of rocket category.

© 2016 Ken Baclawski All Rights Reserved

Rockets Header

```
class Rockets {
public:
  void addRocket(const std::string& name,
         double payload,
         const std::string& manufacturer);

  void sortByName();
  void sortByPayload();
  void sortByManufacturer();

  bool operator==(const Rockets& other) const;
  bool isSameSet(const Rockets& other) const;

  std::string getHtmlTable() const;
  std::string getCategorizedHtmlTable();
```

The default constructor is the only constructor so it is generated automatically

The list is constructed programmatically

Each data member has its own sort method

There are two notions of equality:
1. equality of the lists with corresponding elements equal
2. equality of the set of rockets, possibly in a different order

There are two output formats:
1. An HTML table with one rocket per row
2. An HTML table with the rockets grouped by category

© 2016 Ken Baclawski All Rights Reserved

Rockets Header

```
...
private:
  std::vector<Rocket> rockets_;

  typedef std::tuple
    <std::string,
     std::vector<Rocket>::const_iterator,
     std::vector<Rocket>::const_iterator> Triple;

  std::vector<Triple> partitionByCategory();
};
```

The Rockets class adds functionality to a list of Rocket objects

The `typedef` declares a synonym for a type, reducing the effort to use it and improving readability

The `tuple` and `pair` templates allow one to define *ad hoc* structures more conveniently than defining a class for them.

Pairs and tuples

- A `pair` is an array of exactly two objects
  - Unlike an array, the objects can have different types
  - The two components are `first` and `second`
- A `tuple` is an array of any number of objects
  - The objects can have different types
  - Get a component with the `get<n>` function
- The `typedef` statement declares a synonym for a type
  - Useful for abbreviating complicated type names

Rockets Source

```
void Rockets::addRocket(const std::string& name, double payload,
                        const std::string& manufacturer) {
  rockets_.push_back(Rocket(name, payload, manufacturer));
}

bool Rockets::operator==(const Rockets& otherRockets) const {
  return rockets_ == otherRockets.rockets_;
}
bool Rockets::isSameSet(const Rockets& otherRockets) const {
  if (rockets_.size() != otherRockets.rockets_.size()) {
    return false;
  }
  return is_permutation(rockets_.begin(), rockets_.end(),
            otherRockets.rockets_.begin());
}
```

The equality operator for `vector` uses the `equal` algorithm of the standard library

When using one of the standard algorithms, make sure that the arguments are valid. The algorithms do not check validity.

© 2016 Ken Baclawski All Rights Reserved

Rockets Source

```
...
void Rockets::sortByName() {
  sort(rockets_.begin(), rockets_.end(),
       [](const Rocket& rocket1, const Rocket& rocket2)
       { return rocket1.getName() < rocket2.getName(); });
}
void Rockets::sortByPayload() {
  sort(rockets_.begin(), rockets_.end(),
       [](const Rocket& rocket1, const Rocket& rocket2)
       { return rocket1.getPayload() < rocket2.getPayload(); });
}
void Rockets::sortByManufacturer() {
  sort(rockets_.begin(), rockets_.end(),
       [](const Rocket& rocket1, const Rocket& rocket2)
       { return rocket1.getManufacturer() < rocket2.getManufacturer(); });
}
```

- The `sort` calls all use custom comparison functions
- The `sort` algorithm is not necessarily stable
- To get stable sorting one should use `stable_sort`

Rockets Source

```
...
vector<Rockets::Triple> Rockets::partitionByCategory() {
  vector<Rockets::Triple> partition;
  auto endSmall =
    stable_partition(rockets_.begin(), rockets_.end(),
             [](const Rocket& rocket)
             { return rocket.getCategory() == Rocket::Category::Small; });
  partition.push_back(Triple("Small", rockets_.begin(), endSmall));
  auto endMedium =
    stable_partition(endSmall, rockets_.end(),
             [](const Rocket& rocket)
             { return rocket.getCategory() == Rocket::Category::Medium; });
  partition.push_back(Triple("Medium", endSmall, endMedium));
  auto endHeavy =
    stable_partition(endMedium, rockets_.end(),
             [](const Rocket& rocket)
             { return rocket.getCategory() == Rocket::Category::Heavy; });
  partition.push_back(Triple("Heavy", endMedium, endHeavy));
  partition.push_back(Triple("Super Heavy", endHeavy, rockets_.end()));
  return partition;
}
```

© 2016 Ken Baclawski All Rights Reserved

Partitioning

Energia, 100
Falcon 9 v1.1, 13.15
Falcon Heavy, 54.4
Falcon 1, 0.42
Epsilon, 1.2
Falcon 9 v1.1FT, 22.8

The initial list

Falcon 1, 0.42
Epsilon, 1.2
Energia, 100 ← endSmall
Falcon 9 v1.1, 13.15
Falcon Heavy, 54.4
Falcon 9 v1.1FT, 22.8

After first partition

Falcon 1, 0.42
Epsilon, 1.2
Falcon 9 v1.1, 13.15
Energia, 100 ← endMedium
Falcon Heavy, 54.4
Falcon 9 v1.1FT, 22.8

After second partition

Falcon 1, 0.42
Epsilon, 1.2
Falcon 9 v1.1, 13.15
Falcon 9 v1.1FT, 22.8
Energia, 100 ← endHeavy
Falcon Heavy, 54.4

After third partition

Rockets Source

```
...
string Rockets::getHtmlTable() const {
  string table = "<table border='1'>\n"
    "<tr><th>Name</th><th>Payload</th><th>Manufacturer</th></tr>";
  for (const Rocket& rocket : rockets_) {
    table += "<tr>" + rocket.getHtmlTableEntries() + "</tr>\n";
  }
  table += "</table>\n";
  return table;
}
```

The newline characters `\n` are not necessary for HTML. They are only useful if one is looking at the HTML with a text editor rather than a browser. One can also look at the raw HTML on a browser when you ask to “View Page Source”.

© 2016 Ken Baclawski All Rights Reserved

Rockets Source

```
...
string Rockets::getCategorizedHtmlTable() {
  vector<Rockets::Triple> partition = partitionByCategory();
  string table = "<table border='1'>\n"
    "<tr><th>Category</th><th>Name</th>"
    "<th>Payload</th><th>Manufacturer</th></tr>";
  for (const Rockets::Triple& triple : partition) {
    int rocketCount = get<2>(triple) - get<1>(triple);
    if (rocketCount > 0) {
      for (auto iter = get<1>(triple); iter != get<2>(triple); ++iter) {
    if (iter == get<1>(triple)) {
      table += "<tr><td rowspan='" + to_string(rocketCount) + "'>"
        + get<0>(triple) + "</td>"
        + iter->getHtmlTableEntries() + "</tr>\n";
    } else {
      table += "<tr>" + iter->getHtmlTableEntries() + "</tr>\n";
    }
      }
    }
  }
  table += "</table>\n";
  return table;
}
```

© 2016 Ken Baclawski All Rights Reserved

Rockets Source

```
...
string Rockets::getCategorizedHtmlTable() {
  vector<Rockets::Triple> partition = partitionByCategory();
  string table = "<table border='1'>\n"
    "<tr><th>Category</th><th>Name</th>"
    "<th>Payload</th><th>Manufacturer</th></tr>";
  for (const Rockets::Triple& triple : partition) {
    int rocketCount = get<2>(triple) - get<1>(triple);
    if (rocketCount > 0) {
      for (auto iter = get<1>(triple); iter != get<2>(triple); ++iter) {
     if (iter == get<1>(triple)) {
       table += "<tr><td rowspan='" + to_string(rocketCount) + "'>"
         + get<0>(triple) + "</td>"
         + iter->getHtmlTableEntries() + "</tr>\n";
     } else {
       table += "<tr>" + iter->getHtmlTableEntries() + "</tr>\n";
     }
      }
    }
  }
  table += "</table>\n";
  return table;
}
```

Table begin tag

Column headers

Loop over categories

Loop over rockets in one category

Only the first row shows the category name

Table end tag

The components of the triple are the name of the category and the iterators defining the range. The difference between the two iterators is the number of elements in the range: `get<2>(triple) – get<1>(triple)`. The two iterators are then used to iterate over the rockets in one category.

Main Source

```
int main() {
  try {
    Rockets rockets;
    rockets.addRocket("Energia", 100, "NPO Energia");
    rockets.addRocket("Falcon 9 v1.1", 13.15, "SpaceX");
    rockets.addRocket("Falcon Heavy", 54.4, "SpaceX");
    rockets.addRocket("Falcon 1", 0.42, "SpaceX");
    rockets.addRocket("Epsilon", 1.2, "IHI Aerospace");
    rockets.addRocket("Falcon 9 v1.1FT", 22.8, "SpaceX");
    cout << rockets.getHtmlTable() << endl;
    rockets.sortByPayload();
    cout << rockets.getHtmlTable() << endl;
    rockets.sortByManufacturer();
    cout << rockets.getHtmlTable() << endl;
    rockets.sortByName();
    cout << rockets.getHtmlTable() << endl;
    cout << rockets.getCategorizedHtmlTable() << endl;
...
```

Sample data

Show the original list as well as the list sorted in every possible way

Main Source

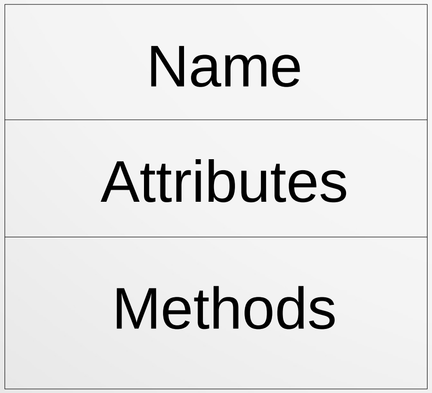
```
...
    Rockets originalRockets = rockets;
    rockets.sortByName();
    if (rockets == originalRockets) {
      cout << "Sorting did not change the list of rockets." << endl;
    } else {
      cout << "Sorting changed the list of rockets." << endl;
    }
    if (rockets.isSameSet(originalRockets)) {
      cout << "Sorting did not change the set of rockets." << endl;
    } else {
      cout << "Sorting changed the set of rockets." << endl;
    }
  } catch (const exception& e) {
    cerr << e.what() << endl;
    return 1;
  }
  return 0;
}
```

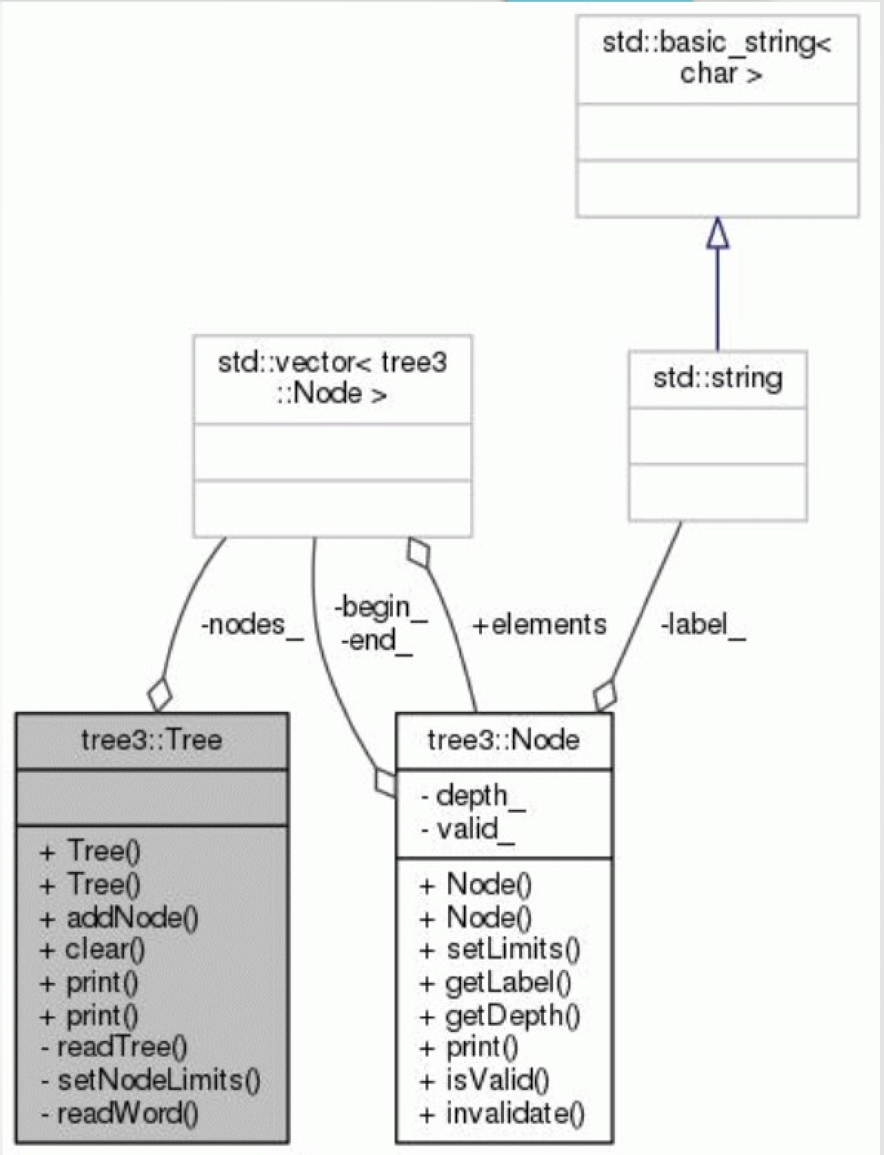
Determine whether sorting changes the list and/or changes the set.
It should change the list, but not the set.

UML Class Diagrams

Classes and Associations

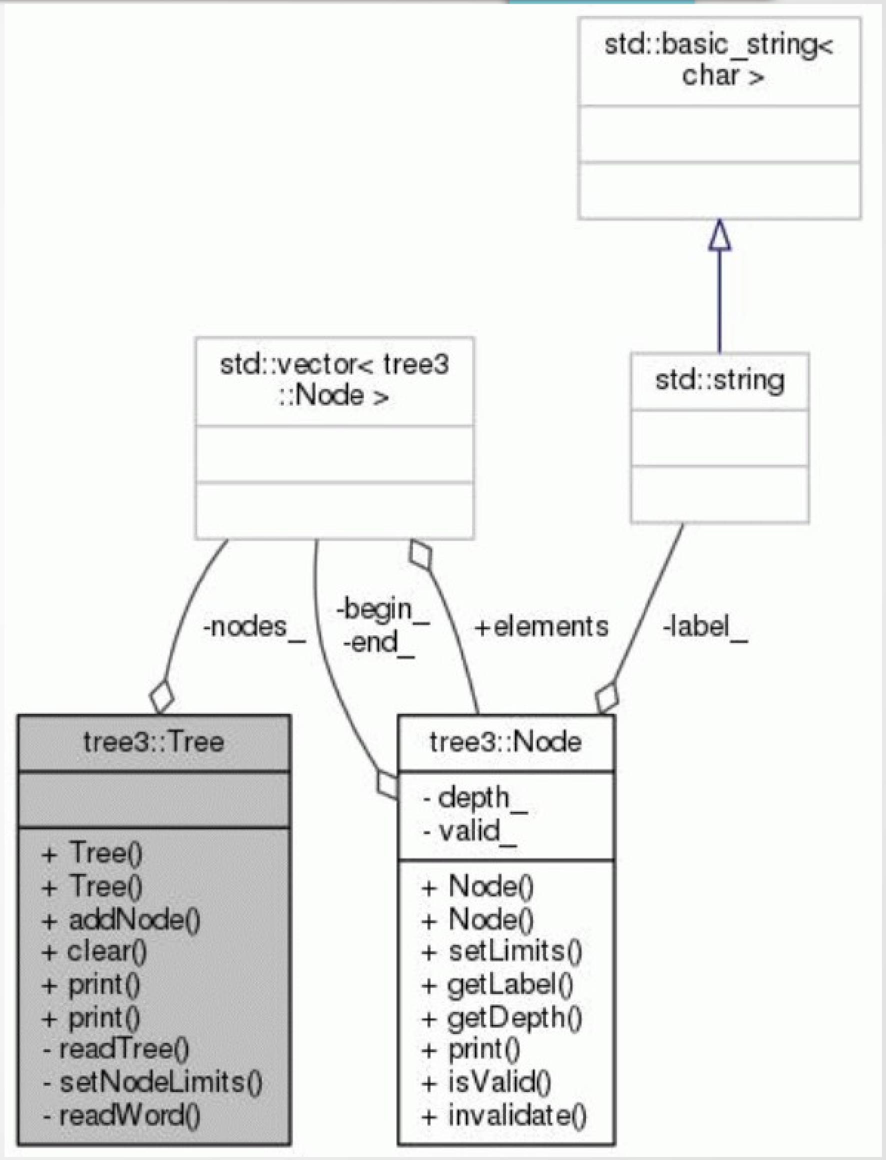
- The boxes are the *classes*
  - Each has three parts
- The lines between them are the *associations*
  - There are several kinds of association





Accessibility Restrictions

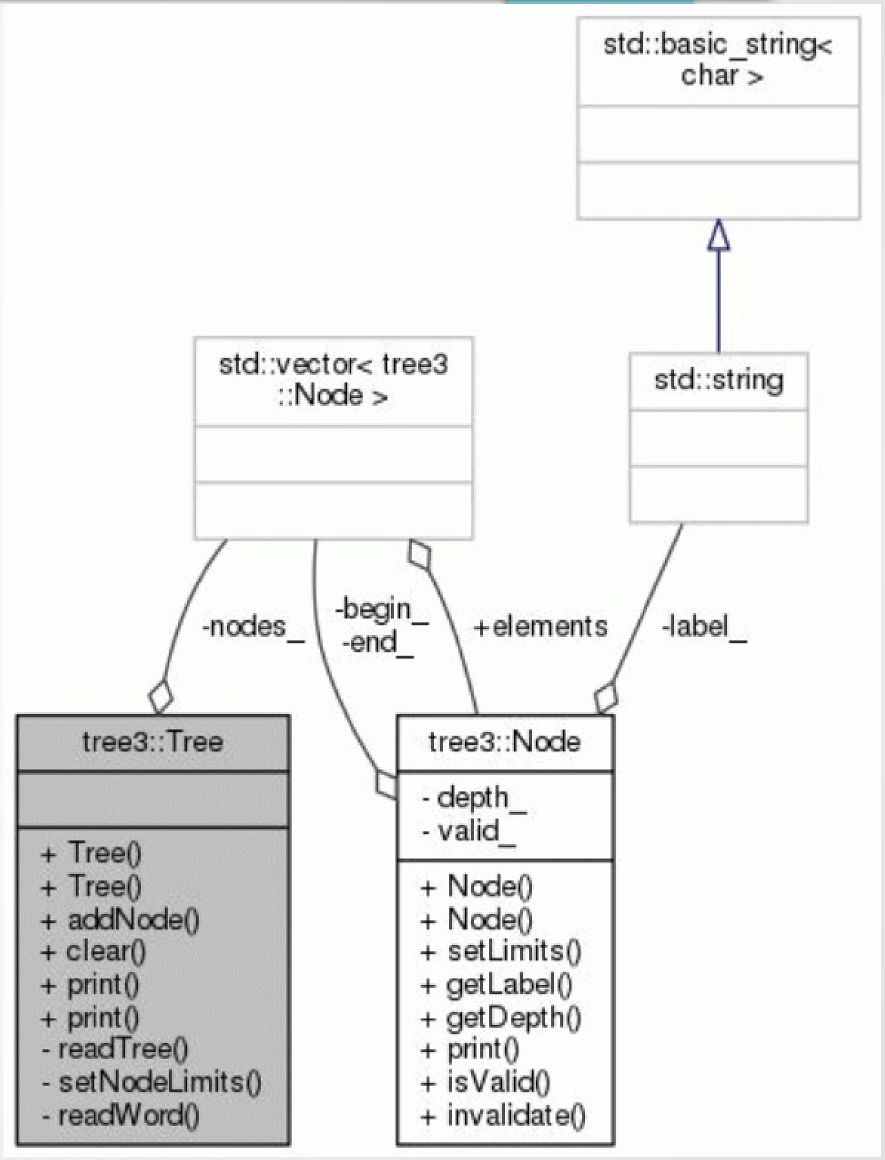
- The plus sign means public
  - +elements
  - +getDepth()
- The minus sign means private
  - -nodes_
  - -setNodeLimits()



© 2016 Ken Baclawski All Rights Reserved

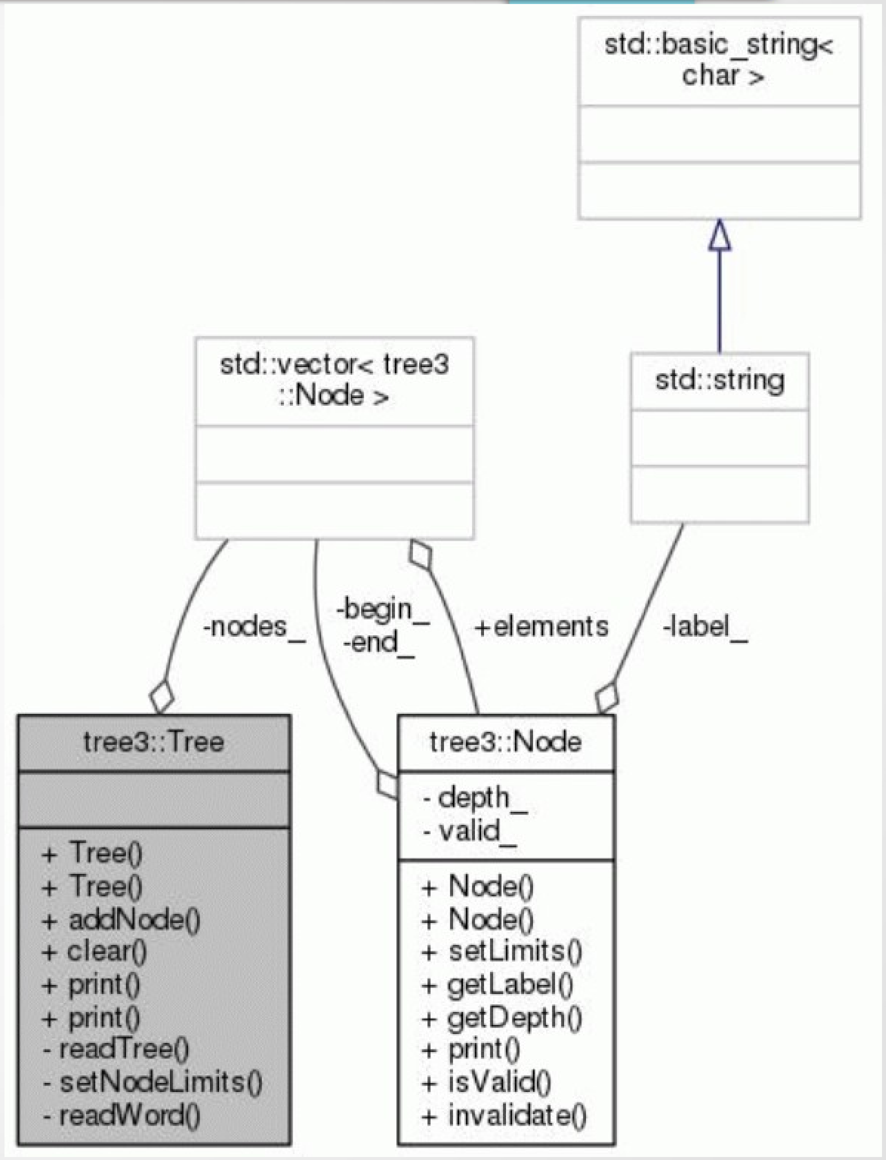
Kinds of Association

- Kinds of association
  - Ordinary association
  - Aggregation
  - Composition
  - Generalization (superclass)
- Associations can have multiplicities
  - Not shown by doxygen



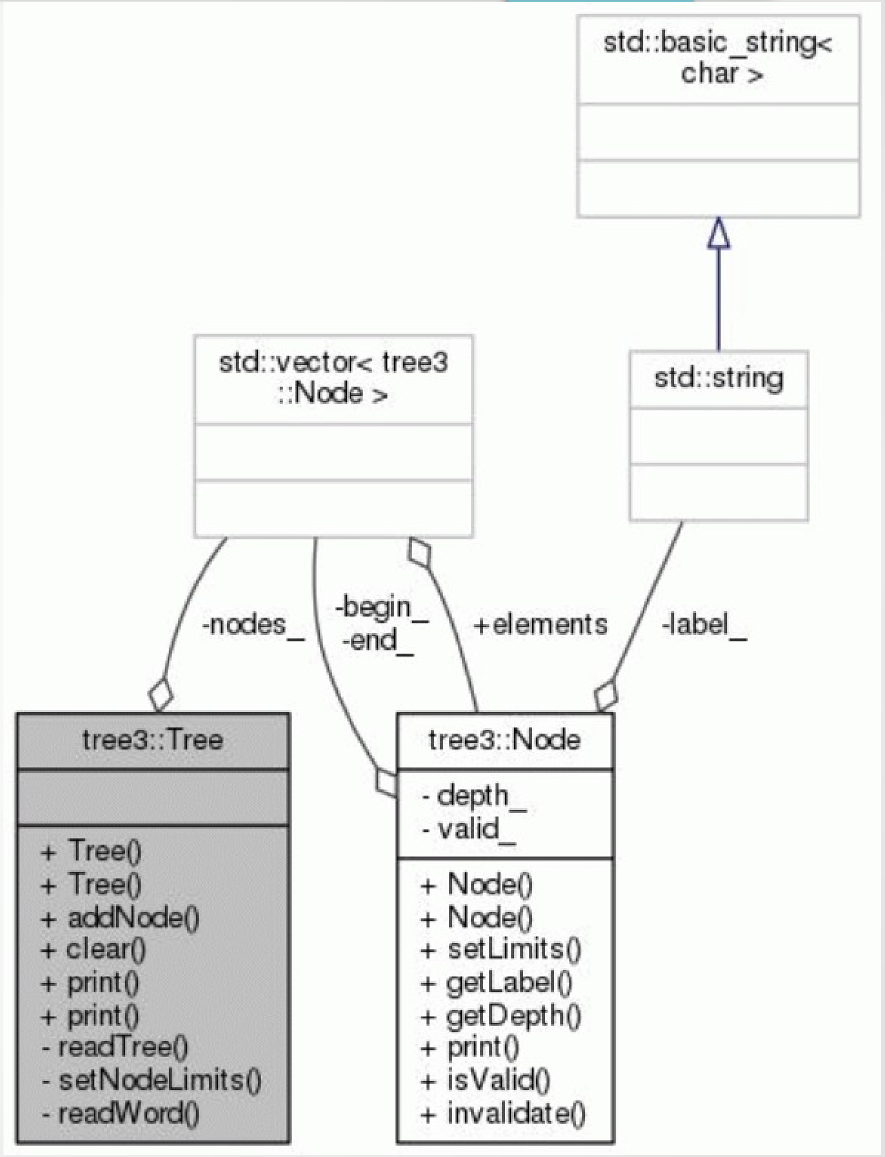
Aggregations

- Aggregation
  - Used for containers
  - The relationship is “part of”
  - Represented with an open diamond on the container side of the relationship
- Composition
  - Strong aggregation
  - The parts cannot exist without the whole
  - Represented with a closed diamond



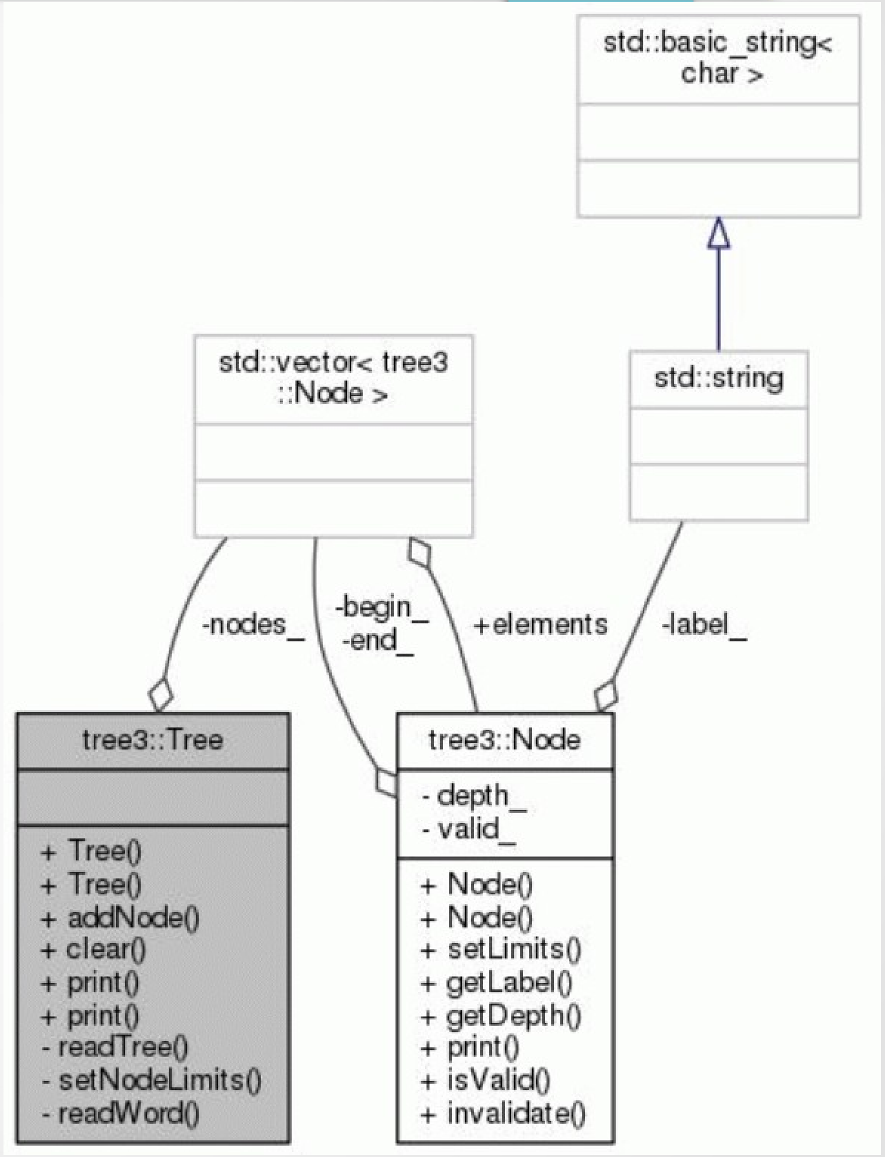
Generalizations and Associations

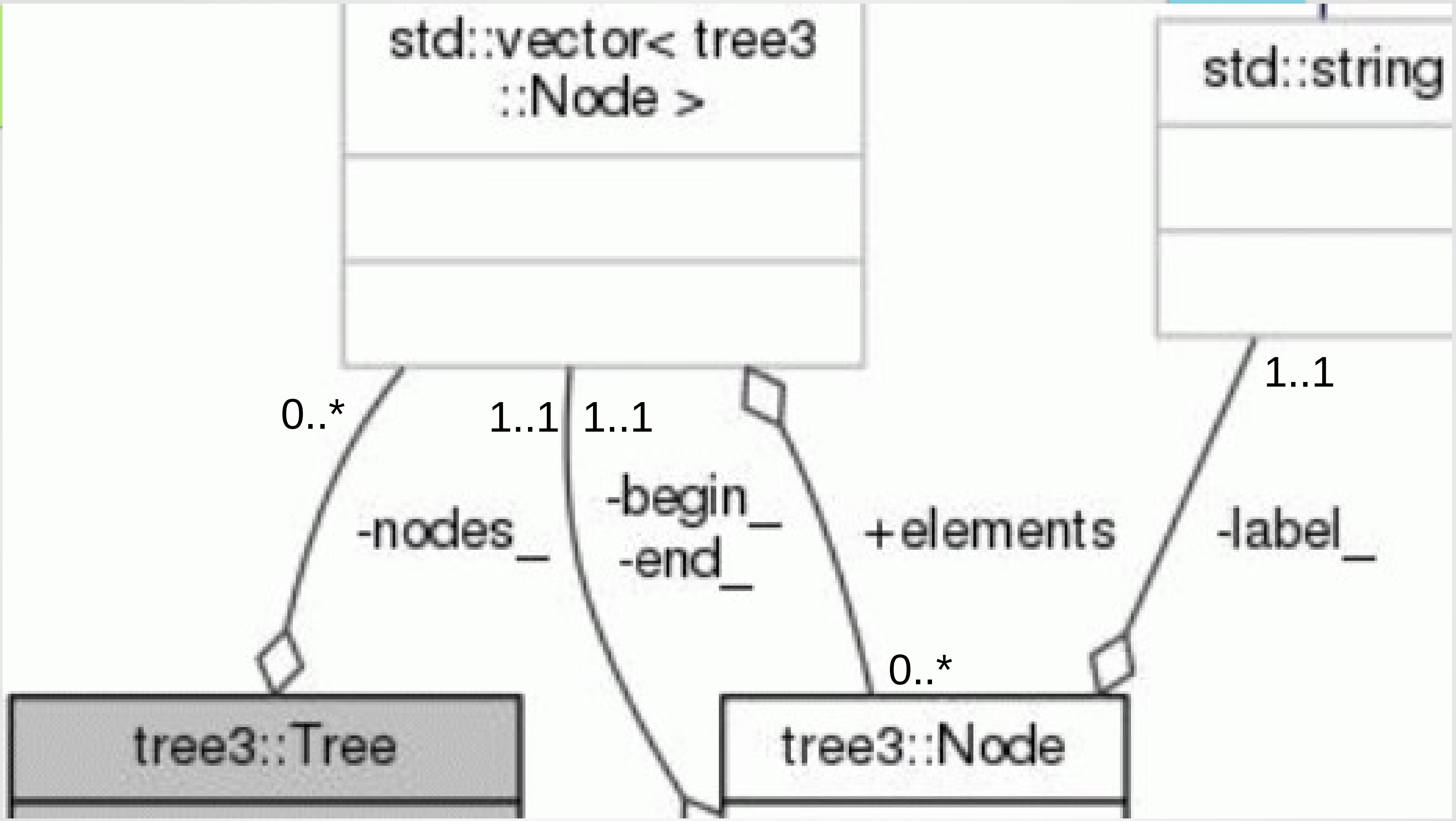
- Generalization
  - Used to show inheritance relationships
  - The relationship is a subset/superset relationship
  - Represented with an open triangle on the superclass side
- Association
  - General relationship between classes
  - Not used by doxygen



Multiplicities

- Multiplicities
  - [m..n] means at least m objects and at most n objects are related
  - The asterisk is used for unlimited upper bound
- [1..1] Exactly one
- [0..1] At most one (optional)
- [0..*] Any number





These are the missing multiplicities. Note that begin_ and end_ are two separate associations

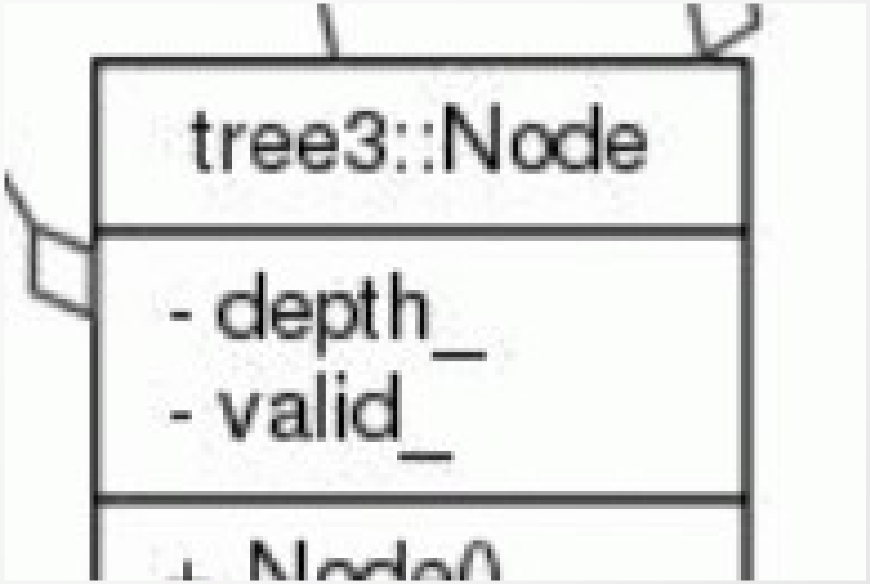
Attributes

- An attribute can have a type and multiplicity

  -depth_ [1..1] : int

  -valid_ [1..1] : bool

- Not shown by doxygen



Computational Complexity

Computational Complexity

- Measures the resources needed for operations
  - Time complexity
  - Space (memory) complexity
- Asymptotic complexity
  - Rate of increase as size of problem increases
  - Lower bound Ω
  - Upper bound O
  - Inaccurate for small problems

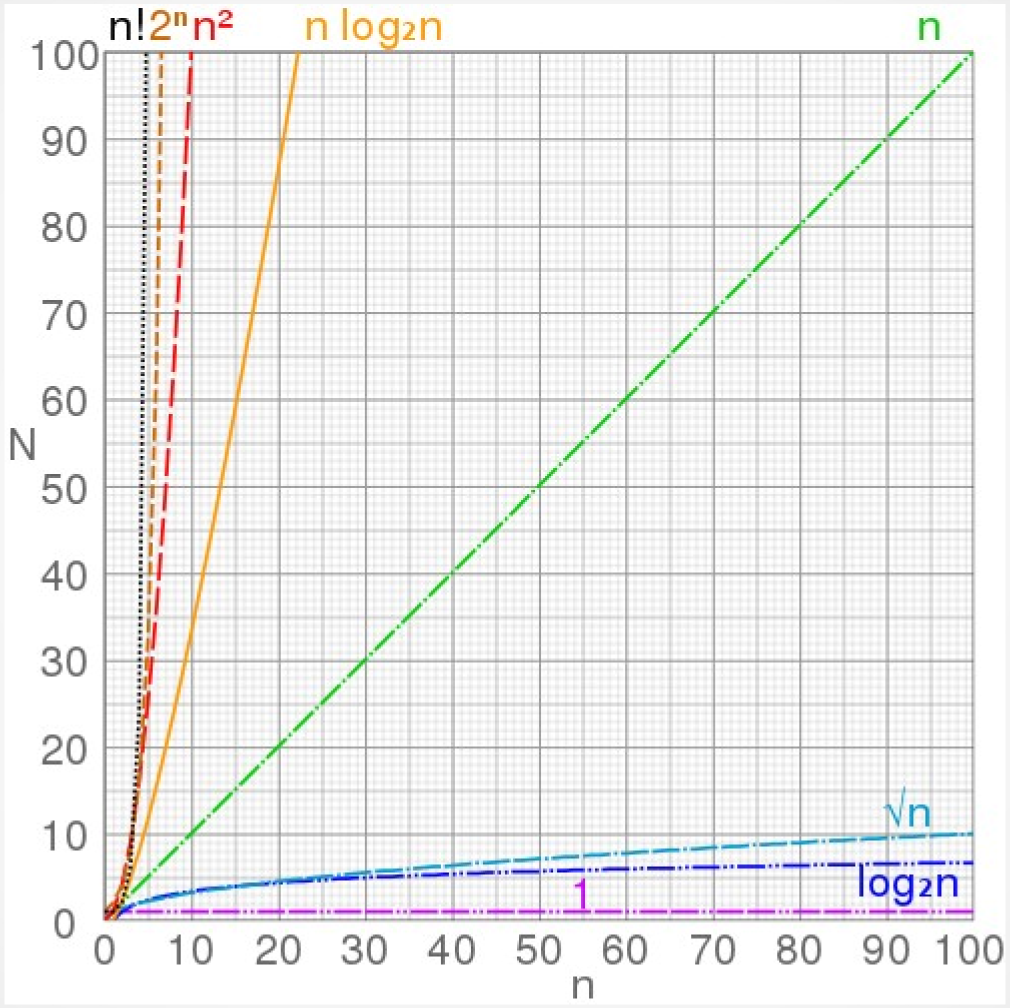
© 2016 Ken Baclawski All Rights Reserved

Some common complexity functions

Factorial n!

Exponential 2^n

Quadratic n^2



Linear n

Square Root √n

Logarithmic log n

Constant 1

Source: https://en.wikipedia.org/wiki/File:Comparison_computational_complexity.svg
License: Creative Commons Attribution-Share Alike 4.0 International license.

Mathematical Definition

- $f(n) = \mathcal{O}(g(n))$ means there exist numbers M and m such that $|f(n)| \leq M\,|g(n)|$ for all $n \geq m$
- $f(n) = \Omega(g(n))$ means there exist numbers M and m such that $|f(n)| \geq M\,|g(n)|$ for all $n \geq m$
- $f(n) = \Theta(g(n))$ means that both $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$

© 2016 Ken Baclawski All Rights Reserved

Properties of complexity measures

- Constant factors are ignored
  - n^2 and $1000n^2$ have the same complexity
  - The complexity is the “limit” as problem size increases
  - Nothing is said about small problem sizes
- Each polynomial is smaller than the next one
  - $O(n^p)$ is smaller than $O(n^q)$ for any $p < q$
- Polynomial is smaller than an exponential
  - $O(n^k)$ is smaller than $O(b^n)$ for any k and any $b > 1$
- Logarithmic is smaller than any polynomial
  - $O(\log n)$ is smaller than $O(n^k)$ for any k

Comparing complexities

- Two operations with different asymptotic complexity
  - One may be faster for small problem sizes and the other for larger problem sizes
  - If this happens, the problem size where they switch is called the crossover point
  - The algorithm that is better will depend on approximately how large the problem size will be
- Two operations with the same asymptotic complexity
  - The ratio between the complexities may tend to a constant
  - This allows one to select the better algorithm even though they have the “same” complexity

© 2016 Ken Baclawski All Rights Reserved

Assignment #5

Requirements

- Develop two classes: `MailingList` and `Person`
- A `MailingList` has a name and list of email addresses.
- A `Person` has a name, phone number and list of their email addresses.
- The main program should print an HTML table (matrix) showing the names and phone numbers of persons vertically, the mailing lists horizontally and whether a person subscribes to the mailing list at the intersections.
- Sorting of lists must be done only when necessary. A flag can be used to determine whether the list is currently sorted.
- The namespace for this assignment is `asst05`. You will lose points if your program has a different namespace than this.

© 2016 Ken Baclawski All Rights Reserved

MailingList Class Methods

- `getName` The getter for the mailing list name
- `subscribe` The method for subscribing to the mailing list. The email address is the only parameter. It is not necessary to eliminate duplicates.
- `getSubscriptionAddresses` Returns a list (`vector`) containing all of the email addresses of a specified person that are in the mailing list. The only parameter is a Person.
  - This method *must* use functions in the algorithm library to determine the email addresses to be returned
  - No loops may be used in this method

© 2016 Ken Baclawski All Rights Reserved

Person Methods

- Constructor of a person with no email addresses
- Getters for the name, phone number and list (`vector`) of all email addresses: `getName`, `getPhoneNumber` and `getEmailAddresses`.
  - The email address list that is returned must be in sorted order. Remember that sorting of lists must be done only when necessary. A flag can be used to determine whether the list is currently sorted.
- `addEmail` adds an email address to the list. It is not necessary to eliminate duplicates.

© 2016 Ken Baclawski All Rights Reserved

Main program

- Constructs mailing lists and persons
  - The information about them is posted and on the next slides
- Print the matrix of who is subscribed to which mailing lists
  - The result for the mailing lists and persons above is also posted and on the next slides.
- A different main program will be used to test your classes.

Mailing lists

The following are the 3 mailing lists (showing name and email addresses)

Dracula: erzebet@example.org cork@nobility.org

Irish Nobility: boyle@example.org lambart@example.org digby@example.org guinness@example.org

Hungarian Nobility: ecsed@nobility.org

© 2016 Ken Baclawski All Rights Reserved

Persons

The following are the 6 persons (showing name, phone number and email addresses)

John Boyle, 555-1000: boyle@example.org cork@nobility.org

Robert Lambart, 555-1001: lambart@example.org cavan@nobility.org

Edward Digby, 555-1002: digby@example.org digby@nobility.org

Gwendolen Guinness, 555-1003: guinness@example.org iveagh@nobility.org

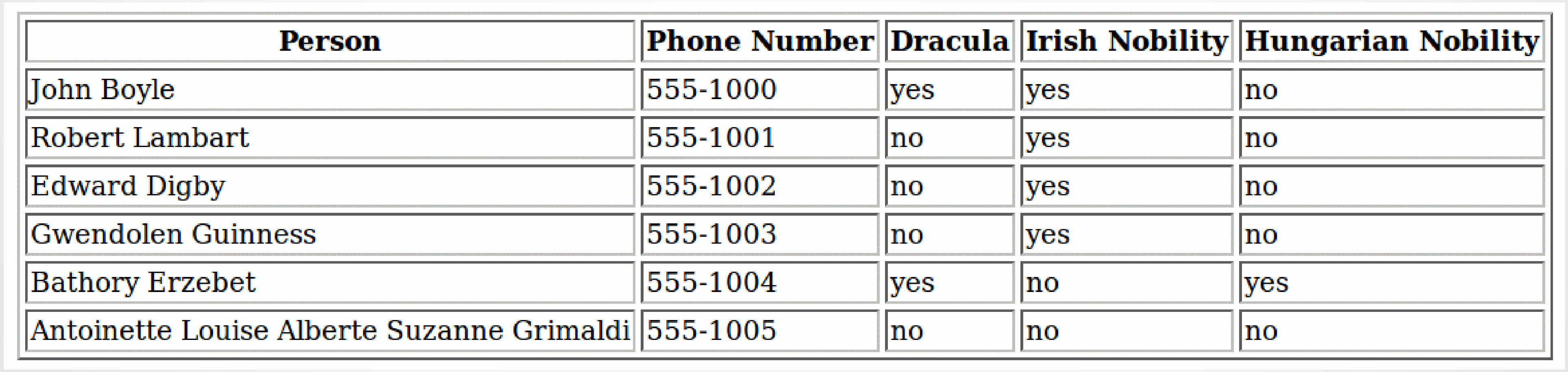
Bathory Erzebet, 555-1004: erzebet@example.org ecsed@nobility.org

Antoinette Louise Alberte Suzanne Grimaldi, 555-1005: grimaldi@example.org massy@nobility.org

© 2016 Ken Baclawski All Rights Reserved

Output

```
<table border='1'>
<tr><th>Person</th><th>Phone Number</th><th>Dracula</th><th>Irish Nobility</th><th>Hungarian Nobility</th></tr>
<tr><td>John Boyle</td><td>555-1000</td><td>yes</td><td>yes</td><td>no</td></tr>
<tr><td>Robert Lambart</td><td>555-1001</td><td>no</td><td>yes</td><td>no</td></tr>
<tr><td>Edward Digby</td><td>555-1002</td><td>no</td><td>yes</td><td>no</td></tr>
<tr><td>Gwendolen Guinness</td><td>555-1003</td><td>no</td><td>yes</td><td>no</td></tr>
<tr><td>Bathory Erzebet</td><td>555-1004</td><td>yes</td><td>no</td><td>yes</td></tr>
<tr><td>Antoinette Louise Alberte Suzanne Grimaldi</td><td>555-1005</td><td>no</td><td>no</td><td>no</td></tr>
</table>
```



Person	Phone Number	Dracula	Irish Nobility	Hungarian Nobility
John Boyle	555-1000	yes	yes	no
Robert Lambart	555-1001	no	yes	no
Edward Digby	555-1002	no	yes	no
Gwendolen Guinness	555-1003	no	yes	no
Bathory Erzebet	555-1004	yes	no	yes
Antoinette Louise Alberte Suzanne Grimaldi	555-1005	no	no	no

© 2016 Ken Baclawski All Rights Reserved

Next Class

- Associative Containers
- Random Number Generation

© 2016 Ken Baclawski All Rights Reserved