CS3520 Programming in C++
Lambda Expressions

Ken Baclawski
Fall 2016

Outline

- Valgrind
- Lambda Expressions
- The functional library
- Graphs
- The graph package
- Assignment #7

Testing for memory leaks

- Valgrind is an instrumentation framework for building dynamic analysis tools.
- The only tool we need is the one for detecting memory management bugs
- This is the default tool of valgrind

Using valgrind

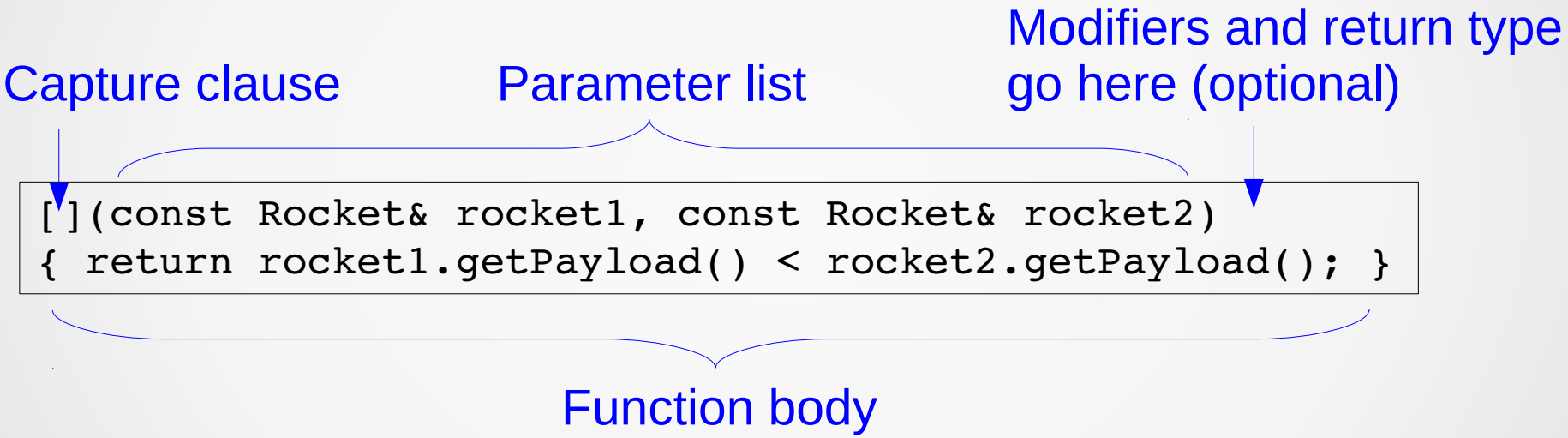
- Install `valgrind`
- Run `valgrind` on your program
- If your program is `main` with no command-line arguments, use this:

```
valgrind ./main
```

Lambda Expressions

Lambda Expression Syntax

Here is an example of a lambda expression from the rocket package:



Invoking a lambda expression

A lambda expression can be invoked as if it was a function

```
Rocket r1("Energia", 100, "NPO Energia");
Rocket r2("Falcon 9 v1.1", 13.15, "SpaceX");
auto lambda = [](const Rocket& rocket1, const Rocket& rocket2)
  { return rocket1.getPayload() < rocket2.getPayload(); };
bool comparison = lambda(r1, r2);
if (comparison) {
  cout << "r1 has a smaller payload" << endl;
} else {
  cout << "r1 does not have a smaller payload" << endl;
}
```

Output is: `r1 does not have a smaller payload`

What is a lambda expression?

- Short answer: a functor object
- Longer answer: a lambda expression defines:
  - A closure
  - An anonymous functor class
  - An overloaded `operator()` method
  - An object of the functor class

Capture Clause

- This is in brackets
- The formal name is *lambda-introducer*
- The capture clause specifies what variables are included in the closure
- Variables in the closure are passed by constant value or by reference

```
int x = 1;
int y = 5;
int z = [x,&y](){ ++y; return x + y; }();
cout << x << " " << y << " " << z << endl;
```

If x was incremented, instead of y, then the program would not compile

Output is: `1 6 7`

Capture Clause

- There are two “wildcards” that can be used in the capture clause
  - The equal sign causes all variables in the current scope to be passed by constant value
  - The ampersand causes all variables in the current scope to be passed by reference
- One can override the wildcard by explicitly specifying a variable in the capture clause

```
int x = 1;
int y = 5;
int z = [&](){ ++x; ++y; return x + y; }();
cout << x << " " << y << " " << z << endl;
```

Output is: `2 6 8`

Capture Clause

- In the context of a method, the class members are accessed using **this**
- For example, if `width_` is one of the data members, then it is accessed with `this->width_`
  - An example of a need for explicitly specifying `this`

Parameter List

- This is a parameter list like any other function
- If there are no parameters, then the parameter list can be omitted

Mutability and Exceptions

- Specify `mutable` to change “pass by constant value” to “pass by value”
  - This does not affect variables passed by reference
- One can also specify what types are thrown
  - This is deprecated so one should not use this feature

Return Type

- Unlike named functions, the type of a lambda expression is declared *after* the declaration
- It is not always required
  - The return type will be inferred if there is only one return statement
- Declaring the return type has some uses
  - To specify the intended return type so that it will be checked
  - To specify the return type when there are several return statements

Return Type

Here is how the previous example would explicitly specify its return type:

```
int x = 1;
int y = 5;
int z = [&]()->int{ ++x; ++y; return x + y; }();
cout << x << " " << y << " " << z << endl;
```

Output is: `2 6 8`

Function Body

- This is the same as any function body
- If there are several return statements, then the return type must be specified

The functional library

Purpose

- Provides classes for function objects
- Recall that a functor object is an object that overloads `operator()`
- A functor object can be used as if it is a function
- To make use of the library include `<functional>`

Class versus Object

- Every object is an *instance of* a class
  - The class is called the *type* of the object
  - The built-in types like `int` and `bool` are types but not classes
- If `MyClass` is a class, one can construct an object of the class in several ways
  - Anonymous object using default constructor: `MyClass()`
  - Anonymous object using constructor: `MyClass(10)`
  - Named object using default constructor: `MyClass m;`
  - Named object using constructor: `MyClass m(10);`
  - Named object that is a copy of another: `MyClass c = m;`

Class versus Object

- In every case, the class determines the type of the object
- Is the reverse possible?
  - Given an object or expression determine its type
  - This is *type inference*
- A named object that is a copy of another: `auto c = m;`
  - The type of `c` is *inferred* from the expression being copied
- More generally, the type of any object or expression is `decltype(`expression`)`
  - `auto c = m;` is a abbreviation of `decltype(m) c = m;`

Lambda Expression Type

- To declare a parameter or return type to be a lambda:
  - Include the `<functional>` header
  - Type is `std::function`<function type>
- For example, if your lambda should have two doubles and return a bool, then this is the type:

```
std::function<bool(double, double)>
```

- The argument can be a lambda expression or a functor
- It could also be a pointer to a function, but this should be avoided

Wrappers

- The `function` template is a *wrapper*
  - It wraps any kind of function object: functor object, lambda expression, or pointer to a function or method
- Other wrappers are:
  - `reference_wrapper` emulates a reference
  - `binary_negate` negates a binary boolean operator
    - For example if you need a `notEquals` operator and have an `equals` operator.
  - `unary_negate` negates a unary boolean operator

Operators

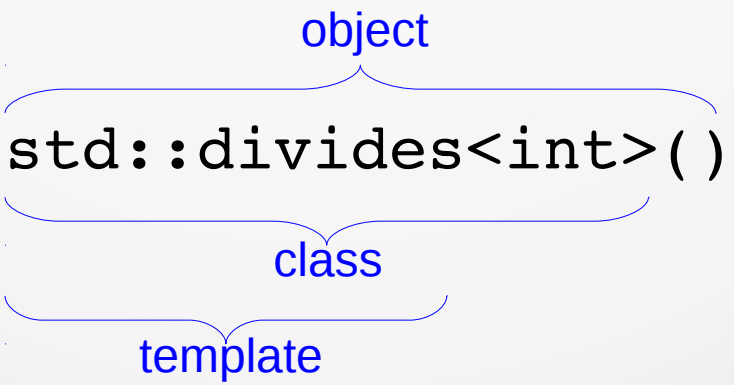
- The various operators all have templates

  `std::divides<int>()`

  is the same as

  `[](const int a, const int b){ return a/b; }`

- Note that `std::divides<int>` is a class while the lambda expression is an object



Hash functions

- The built-in types and string have hash functions

```
std::hash<string>()("Hello, world!")
```

  is the hash value of the string “Hello, world!”

- Pointers do not have built-in hash functions
- Here is how one would construct a hash set:

```
auto hasher = [](const Rocket* rocket){
  return reinterpret_cast<uintptr_t>(rocket); };
unordered_set<Rocket*, decltype(hasher)> rockets(16, hasher);
```

Alternatively, one can use this:

```
unordered_set<Rocket*, function<unsigned int(Rocket*)>>
  rockets(16, hasher);
```

Casting

- There are many kinds of conversions that will be discussed later
- The `reinterpret_cast` reinterprets a memory location as another type
  - In this case, the pointer is being reinterpreted as an unsigned integer
  - The `uintptr_t` type is an unsigned integer having the same size as a hardware address

Graphs

Definition

- There are many meanings for “graph” even in mathematics
- The notion here is a collection of nodes and edges such that each edge connects two nodes
- If the edge is a list of two nodes, the edge is *directed*
- If the edge is a set of two nodes, the edge is *undirected*
- If the two nodes of an edge are the same, the edge is a *loop*
- An *undirected graph* consists of undirected edges
- A *directed graph* consists of directed edges
- The nodes and edges of a graph are often *labeled* (with a string) or *weighted* (with a number)

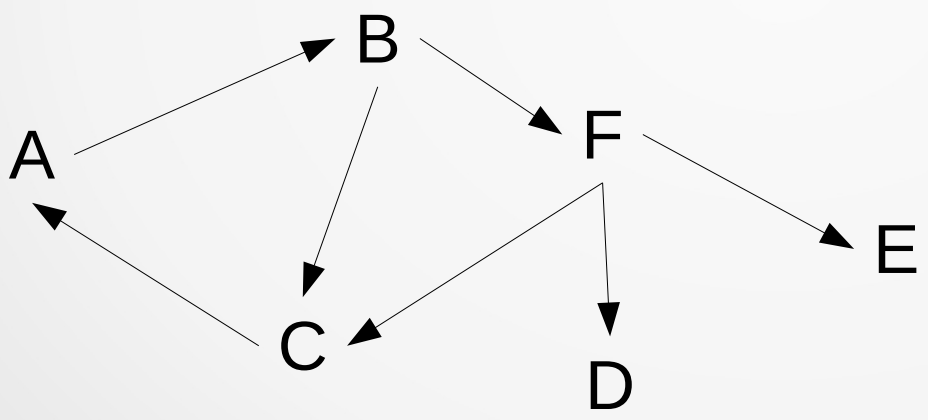
Terminology

- Nodes are also called *vertices* or *points*
- Edges are also called *arcs*
- If a directed edge has nodes A and B, in this order, then A is the *source* and B is the *target* (or *destination*)
- An undirected edge is sometimes regarded as being two directed edges, one in each direction
- A *path* is a collection of directed edges that are joined with each other
- A *cycle* is a path that returns to its starting node
- A directed graph is *acyclic* if it has no cycles

Representing a graph

The most common way to implement a graph is with adjacency lists

- Each node has a list of the nodes that are targets of edges at the node

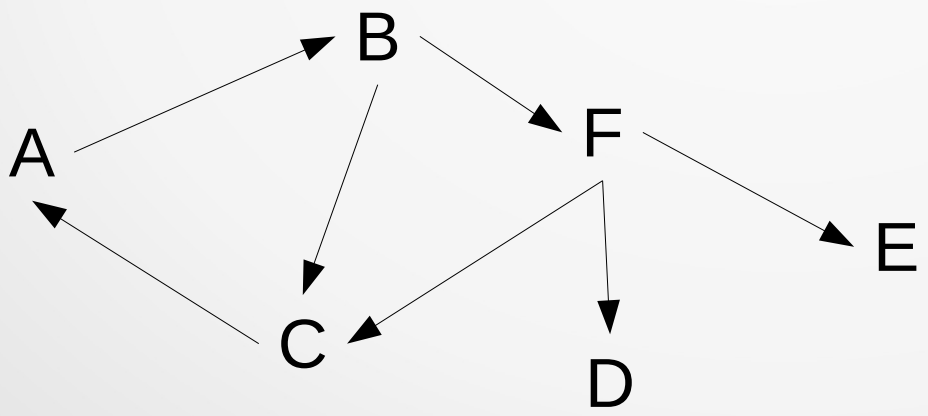



```
A: [B]
B: [C, F]
C: [A]
D: []
E: []
F: [C, D, E]
```

Representing a graph

If one uses vectors for the adjacency lists, then there is a problem:

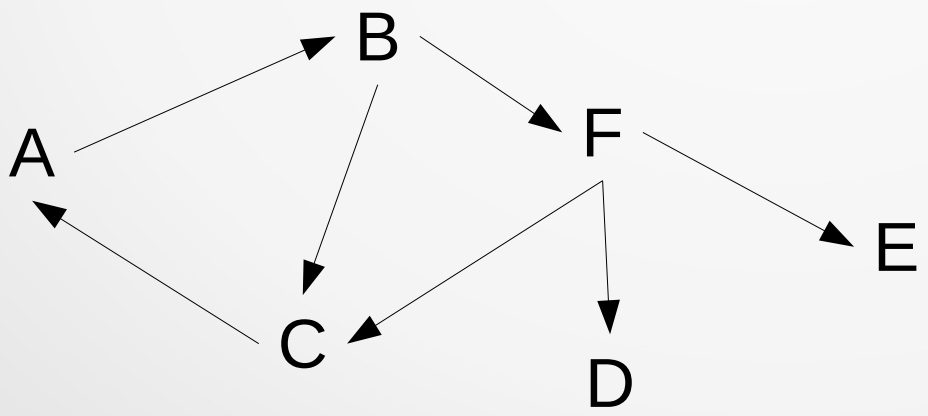
- The same node can occur more than once
- The vector cannot *contain* the adjacent nodes as with trees
- We could use references but this will not work if there is a cycle
- Even if there are no cycles, it is awkward to construct the graph




```
A: [B]
B: [C, F]
C: [A]
D: []
E: []
F: [C, D, E]
```

Representing a graph

- To resolve this problem, the adjacency lists use vectors of pointers
- The nodes are constructed in the free store
- The nodes must be explicitly deleted




```
A: [B]
B: [C, F]
C: [A]
D: []
E: []
F: [C, D, E]
```

The graph package

Requirements

- Read a graph specified with pairs of node labels
- Print the node labels and the adjacent node labels
- Perform the depth-first search algorithm applying functions at the previsit and postvisit times
- Depth-first search is not a single algorithm but rather a kind of “template” for many algorithms
  - Data can be added to the nodes and edges
  - Actions can be performed at different times during the traversal of the nodes
- The graph package supports all of these variations

Node Header

```
#ifndef GRAPH_NODE_H
#define GRAPH_NODE_H
#include <vector>
#include <functional>

namespace graph {

/**
 * One node in a graph.  Each node has a label,
 * adjacent nodes and a flag indicating whether it
 * has been visited.
 * @author Ken Baclawski
 */
class Node {
public:
  /** Construct a labeled node. */
  Node(/** The label of the node. */ std::string label);
```

Include the functional library

Node Header

```
/** Add an outgoing edge to this node.  */
void addEdge(/** The target of the edge. */
             Node* target) noexcept;

/**
 * Explore this node. This is the visit to the
 * node in the depth-first search algorithm.
 */
void explore(/** The previsit function. */
             std::function<void(const Node&)> preAction,
             /** The postvisit function. */
             std::function<void(const Node&)> postAction);
```

Pass a pointer by value (copy of the pointer)

The functions have one parameter that is a constant reference to a node and return nothing

Node Header

```
/**
 * Determine whether the node was visited.
 * @return true if and only if the node was visited.
 */
bool wasVisited() const noexcept;

/**
 * Show the node label.
 * @return The label of the node.
 */
std::string getLabel() const noexcept;

/**
 * Show the node and its adjacent node labels.
 * @return The string representation of this
 * node and its adjacent nodes.
 */
std::string getLabelAndAdjacentNodeLabels() const noexcept;
```

This is important for depth-first search to avoid an infinite recursion

These methods are getters for displaying a node and its edges

Node Header

```
private:
  /** The label of the node. */
  const std::string label_;

  /** The nodes for which there are
      outgoing edges from this node. */
  std::vector<Node*> adjacentNodes_;

  /** Whether the node has been visited. */
  bool visited_ = false;
};

}

#endif
```

This is a vector of pointers to nodes, not a vector of nodes. When the vector is deleted, the nodes are not deleted

Node Source

```
#include <string>
#include "Node.h"
#include "Graph.h"
using namespace std;
using namespace graph;

Node::Node(string label) : label_(label) {}

bool Node::wasVisited() const noexcept {
  return visited_;
}

void Node::addEdge(Node* target) noexcept {
  adjacentNodes_.push_back(target);
}
```

Adding a pointer to a vector copies the pointer but not what it points to.

Node Source

```
void Node::explore(function<void(const Node&)> preAction,
          function<void(const Node&)> postAction) {

  // Mark the node as having been visited
  visited_ = true;

  // Execute the previsit function
  preAction(*this);

  // Recursively search every adjacent node
  for (Node* target : adjacentNodes_) {
    if (!target->visited_) {
      target->explore(preAction, postAction);
    }
  }

  // Execute the postvisit action
  postAction(*this);
}
```

This ensures that every node is visited exactly once

The previsit action is performed before any adjacent nodes are scanned. Also called the *prefix order*.

The postvisit action is performed after all adjacent nodes were scanned. Also called the *postfix order*.

Node Source

```
string Node::getLabel() const noexcept {
  return label_;
}

string Node::getLabelAndAdjacentNodeLabels() const noexcept {
  string nodeString(label_);

  // If there are no outgoing edges, use the label
  if (adjacentNodes_.empty()) {
    return nodeString;
  }

  // If there are outgoing edges, show all of them
  nodeString += " ->";
  for (Node* target : adjacentNodes_) {
    nodeString += " " + target->label_;
  }
  return nodeString;
}
```

Graph Header

```
#ifndef GRAPH_GRAPH_H
#define GRAPH_GRAPH_H
#include <string>
#include <unordered_map>
#include <functional>
#include "Node.h"

namespace graph {

/**
 * A directed graph.  The graph is represented
 * using adjacency lists.  Each node has a unique
 * label.  One can get a node using its label.
 * One can read a graph, print it, and explore it
 * using depth-first search.
 * @author Ken Baclawski
 */
class Graph {
```

Graph Header

```
public:
  /** Destruct the graph object. */
  ~Graph();

  /**
   * Read a graph from a file.  The graph should
   * have a sequence of label pairs, one for each
   * directed edge in the graph.  Duplicates are
   * ignored.
   */
  void readGraph(/** The name of the file with the graph. */
                 const std::string& filename);
```

The destructor is responsible for deleting the nodes

Graph Header

```
/**
 * Perform the depth-first search algorithm.  One can
 * perform actions when each node is first encountered
 * and when the scan of its adjacent nodes is completed.
 */
void search(/** The previsit function. */
            std::function<void(const Node&)> preAction,
            /** The postvisit function. */
            std::function<void(const Node&)> postAction);
/**
 * Display the graph.
 * @return The graph as a collection of
 * adjacency lists.
 */
std::string toString() const noexcept;
```

Graph Header

A hash map is used for finding nodes quickly

```
private:
  /** The nodes of the graph being searched. */
  std::unordered_map<std::string, Node*> labelToNode_;

  /**
   * Get the node with a specified label.
   * @return The node with the specified label.
   * If there is no such node, then a new node is
   * constructed with this label.
   */
  Node* getNode(/** The node label. */
                const std::string& label) noexcept;
```

Graph Source

```
#include <unordered_map>
#include <fstream>
#include "Node.h"
#include "Graph.h"
using namespace std;
using namespace graph;

Graph::~Graph() {

  // Delete all of the nodes

  for (auto labelNode : labelToNode_) {
    delete labelNode.second;
  }
}
```

When a graph is deleted, all of its nodes are also deleted. This is not safe because there could still be pointers to the deleted nodes after their graph is deleted. We will solve this problem when we introduce smart pointers.

Graph Source

```
void Graph::readGraph(const string& filename) {

  // Open the file
  ifstream reader(filename);

  // Read pairs of words
  while (reader) {
    string sourceLabel;
    reader >> sourceLabel;
    if (reader) {
      string targetLabel;
      reader >> targetLabel;

      // Get the nodes and add an edge
      Node* source = getNode(sourceLabel);
      Node* target = getNode(targetLabel);
      source->addEdge(target);
    }
  }
}
```

The file stream will be deleted when the method completes. Deleting the file stream closes the file and releases system resources. This is an example of Resource Acquisition is Initialization (RAII).

Graph Source

```
void Graph::search(function<void(const Node&)> preAction,
                   function<void(const Node&)> postAction) {

  // Loop over all nodes

  for (auto labelNode : labelToNode_) {
    Node* node = labelNode.second;
    if (!node->wasVisited()) {
      node->explore(preAction, postAction);
    }
  }
}
```

This is the top-level of depth-first search. It ensures that every node is visited.

Graph Source

```
string Graph::toString() const noexcept {

  // Distinguish the empty graph case

  if (labelToNode_.empty()) {
    return "Empty graph";
  } else {

    // Display each node on a separate line

    string display;
    for (auto labelNode : labelToNode_) {
      display += labelNode.second->
        getLabelAndAdjacentNodeLabels() + "\n";
    }
    return display;
  }
}
```

Graph Source

```
Node* Graph::getNode(const string& label) noexcept {
  Node* node = nullptr;
  if (labelToNode_.count(label) == 0) {

    // The graph does not have a node with the label

    node = new Node(label);
    labelToNode_.insert({label, node});
  } else {

    // The graph has a node with the label

    node = labelToNode_[label];
  }
  return node;
}
```

This allocates space for the node in the free store and also calls the constructor

This assignment is safe because it is guaranteed that there is a node with this label

Main Source

```
#include <iostream>
#include "Graph.h"
using namespace std;
using namespace graph;

/**
 * @namespace graph Depth-First Search Algorithm.
 * Read a graph specified with pairs of node
 * labels and print it.  Then perform depth-first
 * search on the graph.
 * @author Ken Baclawski
 */
/**
 * Main program for the depth-first search
 * algorithm.  A graph is read from a file.
 * Depth-first search is performed.  The graph is
 * printed.
 * @return The status code.
 */
```

Main Source

```
int main(/** The number of command-line arguments. */
         int argc,
         /** The command-line arguments as C-style strings. */
         const char** args) {

  // Check that there is at least one argument on the
  // command-line.  Note that C counts the command itself
  // as the first command-line "argument".

  if (argc < 2) {
    cout << "Usage: " << args[0] << " filename" << endl;
    return 1;
  }
...
```

One must always check that the expected command-line arguments are provided. The “usage” report is a common response to missing command-line arguments.

Main Source

```
  try {

    // Read the graph

    Graph graph;
    cout << "The directed graph in the file "
         << args[1] << endl;
    graph.readGraph(args[1]);

    // Show the graph

    cout << "is the following:" << endl;
    cout << graph.toString() << endl;
...
```

Main Source

```
    // Perform depth-first search

    cout << "Performing depth-first search" << endl;
    int n = 0;
    graph.search([&n](const Node& node)
      { cout << n++ << " previsit to "
             << node.getLabel() << endl; },
                 [&n](const Node& node)
      { cout << n++ << " postvisit to "
             << node.getLabel() << endl; });

  } catch (const exception& e) {

    // Show the exception
    cerr << e.what() << endl;
    return 1;
  }
  return 0;
}
```

Capture the variable n by reference in both lambda expressions so that it will be incremented each time a previsit or postvisit occurs. These times are important for many applications of depth-first search.

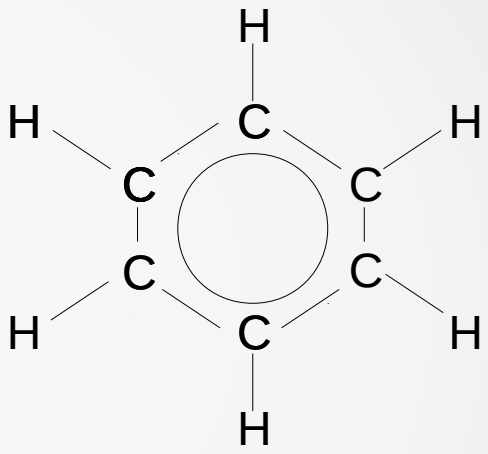
Assignment #7

Requirements

- Develop two classes: `Atom` and `Molecule`
- A molecule has a name and contains a set of bonds
- A bond is a set of atoms
  - A molecule contains a set of sets of atoms
- An atom has a symbol and is part of a molecule
- Use ordered (tree) sets, not unordered (hash) sets

Example Molecule

- The name of this molecule is benzene
- It has twelve atoms and seven bonds
- Six of the bonds are between C and H atoms
- The formula is C6H6
- The letters C and H are the symbols of the atoms
- The set of all six C's form one bond



Atom Class

- Constructor that constructs an atom with a symbol but not contained in any molecule
- No destructor is needed
- There is a getter for the symbol
- There is a setter for the molecule
  - The setter throws an exception if it tries to change the molecule to another one

Molecule Class

- Constructor that constructs a molecule with a name but no bonds
- The destructor deletes all of the atoms that it contains
- There is a getter for the name
- `addBond` adds a bond to the molecule
  - `addBond` also makes the atoms in the bond part of the molecule
- `getAtoms` gets the set of all atoms in the molecule
- `getFormula` computes the formula of the molecule

Main Program

- Constructs water and benzene
- Prints their formulas
- Catch an exception that is thrown and print its message
- Run your program and check that there are no memory leaks or other memory management problems
  - Use `valgrind`

Next Class

- Testing