1 Claiming Lost Luggage [ / 23]

Write a function named `claim_bag` that takes in 2 arguments. The first argument is of type `UnclaimedBags` and contains the records of all unclaimed luggage at the airport. The second argument is of type `BagDescription` and contains a passenger’s description of their missing bag. The function should search through the bag records and have one of three behaviors: 1) Locate the unique bag matching that description and remove it from the data structure, 2) Determine that no bag matching that description exists in the collection, or 3) Find multiple bags matching the description and request additional information. Here are the relevant typedefs:

```cpp
typedef std::vector<std::pair<std::string, std::string>> BagDescription;
typedef std::list<BagDescription> UnclaimedBags;
```

Note that the description of each bag prepared by the airport’s unclaimed luggage office may be different from the description provided by the passenger. Some details may be omitted, and some details may be extra. For example, if the unclaimed bag office contains these 4 records:

- color:black  zipper:gold  size:small  wheels:wheels
- color:green  size:medium  wheels:no_wheels
- color:tan  zipper:gold  size:small  wheels:no_wheels
- color:brown  zipper:silver  size:large

And a passenger comes to the office looking for a bag with features:

- size=small, designer=gucci, and zipper=gold

they will be told:

ERROR! 2 possible matching bags found, provide additional details

If instead they describe their bag as being:

- size=small, designer=gucci, color=brown, and wheels=no_wheels

they will be told:

ERROR! bag not found

Finally, if they revise their description to be:

- zipper=gold, size=small, designer=gucci, and wheels=no_wheels

this message will be output:

bag returned! 3 unclaimed bags remain

1.1 Order Notation [ / 4]

Assuming the airport has \( n \) bags, and each bag is described by a maximum or average of \( a \) descriptive features by the airport office and \( p \) descriptive features by the passenger, what is the order notation of the `claim_bag` function? Write 2-3 concise and well written sentences justifying your answer. (Note: You may want to answer this after completing the implementation on the next page.)

Solution: The function must loop over all bags in the airport in a linear manner. For each bag, we must loop over all terms in both the airport database and passenger description and do an all-pairs comparison. The final answer is \( O(nap) \). We cannot assume that the descriptions are sorted or otherwise organized. But if we do sort both descriptions alphabetically by category (not required for full credit), then the comparisons will be faster: \( O(p \log p + a \log a + p) \).

1.2 claim_bag Implementation [ / 19]

Now implement the `claim_bag` function. You will be graded on code organization and clarity. Do not make your code more complex to achieve a faster running time. You may write and use a helper function.

Solution:

```cpp
bool match(const BagDescription &bag, const std::string &feature, const std::string &value) {
    for (int i = 0; i < bag.size(); i++) {
```
if (bag[i].first == feature && bag[i].second != value) { return false; } }
return true;

void claim_bag(UnclaimedBags &bags, const BagDescription &bag) {
std::vector<UnclaimedBags::iterator> possible;
for (UnclaimedBags::iterator itr = bags.begin(); itr != bags.end(); itr++) {
  bool ismatch = true;
  for (int i = 0; i < bag.size(); i++) {
    if (!match(*itr,bag[i].first,bag[i].second))
      ismatch = false;
  }
  if (ismatch == true) { possible.push_back(itr); }
}
if (possible.size() == 0) {
  std::cout << "ERROR! bag not found" << std::endl;
} else if (possible.size() > 1) {
  std::cout << "ERROR! " << possible.size() << " possible matching bags found, provide additional details" << std::endl;
} else {
  bags.erase(possible[0]);
  std::cout << "bag returned! " << bags.size() << " unclaimed bags remain" << std::endl;
}
}

2 Friends of Friends with Maps & Sets [ / 23]

For this problem you will write a function named suggest_friends that takes 2 arguments, the name of a person (as an STL string) and an STL map storing all current friendships defined with this typedef:

typedef std::map<std::string, std::set<std::string> > friend_map;

The function should return an STL set of STL strings containing the names of all possible new friends for the specified person. Each possible new friend should be the friend of an existing friend.

As an example, let’s study the variable of type friend_map named all_friends to the left. Note that friendships are one way and not necessarily mutual (alice lists chris as a friend but chris does not list alice as a friend). Let’s find all possible new friends for alice.

We start by looking through all friends of alice’s current friends (just one step away). In this example, we have just one new friend suggestion for alice: erin. We make this suggestion because alice lists chris as a friend and chris lists erin as a friend. However, even though alice lists bob and bob lists chris, we do not suggest chris as a new friend, because alice has already listed him as a friend.

We do not suggest alice because a person does not list him/herself as a friend. And finally, we do not suggest fred because that potential friendship relationship is more than one step away (alice lists chris, chris lists erin, and erin lists fred). If we ask for friend suggestions for a person who does not have any friendships recorded in the map (e.g., dave in this example), then we will print a message to std::cerr.

2.1 Expected Output [ / 4]

First, complete the expected return data for the example above. Hint: In the 4 boxes below you will list 8 total names.
suggest_friends("alice",all_friends); { erin}
suggest_friends("bob",all_friends);
Solution: { dave and erin }
suggest_friends("chris",all_friends);
Solution: { alice and fred }
suggest_friends("dave",all_friends);  ERROR: could not find dave in the friend map

suggest_friends("erin",all_friends);
Solution: { bob, chris, and dave }

suggest_friends("fred",all_friends);
Solution: { alice }

2.2 Order Notation  

Assuming the map holds friendship information for \( n \) people and each person lists an average or maximum of \( k \) other people as friends, what is the running time for the suggest_friends function? (Note: You may want to answer this after completing the implementation on the next page.)

Solution: \( O(\log n) \) to find the initial person, then we loop over \( k \) friends. We need to lookup each of them in the map, which is \( O(k \cdot \log n) \). Then we will loop over their friends \( k \cdot k \) friends-of-friends in worst case. We will add those friends-of-friends into a set, which takes \( O(k^2 \log (k^2)) \). Overall: \( O(\log n + k \cdot \log n + k^2 \cdot \log k^2)) \). Since log exponents simplify, more correctly: \( O(\log n + k \cdot \log n + k^2 \cdot \log k) \).

2.3 Implement suggest_friends  

Now, write the function suggest_friends. Be careful with syntax and make sure your function is efficient.

Solution:

```c++
std::set<std::string> suggest_friends(const std::string &person, const friend_map &all) {
    std::set<std::string> answer;
    friend_map::const_iterator p = all.find(person);
    if (p == all.end()) {
        std::cerr << "ERROR: could not find " << person << " in the friend map" << std::endl;
    } else {
        for (friend_map::const_iterator f = (*p).second.begin(); f != (*p).second.end(); f++) {
            friend_map::const_iterator fs = all.find( (*f) );
            if (fs != all.end()) {
                for (friend_map::const_iterator possible = (*fs).second.begin(); possible != (*fs).second.end(); possible++) {
                    if (*possible != person && (*p).second.find(*possible) == (*p).second.end()) {
                        answer.insert(*possible);
                    }
                }
            }
        }
    }
    return answer;
}
```

3 Potpourri  

3.1 Hash Tables / Hash Functions  

Which of the following statements are true about hash tables / hash functions?

A) A good hash function should run in \( O(1) \) time (expected).

B) A good hash function should use randomness to uniformly distribute the keys.

C) A linked list is commonly used to implement the separate chaining method of collision resolution.

D) When the amount of data stored in a hash table greatly exceeds the initial estimates, the table is resized and all data must be re-hashed.

E) According to Google, the hash table is the “single most important data structure known to mankind”.

F) First available in C++11, STL’s unordered_set provides a hash table container class.

G) Any program using STL map can be improved by replacing the map with a hash table.
3.2 Writing Simple String Hash Functions [ / 5] 
Study the distribution of keys in this hash table of size $n = 7$. First, add two new words to this table, following the established pattern:

<table>
<thead>
<tr>
<th>a</th>
<th>bulb</th>
<th>logic</th>
<th>fold</th>
<th>cake</th>
<th>chef</th>
<th>dog</th>
</tr>
</thead>
<tbody>
<tr>
<td>idea</td>
<td>i</td>
<td>lab</td>
<td>folk</td>
<td>pie</td>
<td>self</td>
<td>fern</td>
</tr>
<tr>
<td>sea</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now, write a simple hash function for STL strings that implements this pattern.

**Solution:**

```c++
int hash (const std::string& s, int n) {
    char last = s[s.size()-1];
    int val = last - 'a';
    return val % n;
}
```

3.3 Ternary Heaps [ / 11]

A ternary heap is the same as a balanced binary heap except each node has up to 3 children instead of 2. Given the values: 13 20 6 9 7 12 15 11 24, draw both a binary heap and a ternary heap containing these values. (Smaller values have higher priority and should be removed first.)

Draw a binary heap containing the values: Draw a ternary heap containing the values:

```
9
 / 
12 20
 / 
13 11
```

```
9
 / 
12 20
 / 
15 24
```

**Solution:** (one of many)

Draw the vector representation for the ternary heap structure you drew above.

```
6 9 7 13 12 20 15 11 24
```

**Solution:**

Now complete the implementation of the helper functions below that will be used by `percolate_up` and `percolate_down` to access the parent and child elements within a ternary heap represented as a vector. Each function takes in the integer index of an element in the heap and returns the index of the requested parent or child element. You may assume that the caller of these functions will correctly handle the corner cases in which the parent or child does not exist.

**Solution:**

```c++
int parentOf(int index) {
    return (index-1) / 3;
}
int leftChildOf(int index) {
    return index*3 + 1;
}
int centerChildOf(int index) {
    return index*3 + 2;
}
int rightChildOf(int index) {
    return index*3 + 3;
}
```
3.4 Memory Debugging

If your code is said to have a “memory leak”, which of the following statements are true?

A) Neither Valgrind nor Dr. Memory will identify the problem because these tools only detect memory corruption.
B) The problem can be corrected by adding `delete` statements at the end of the `main` function.
C) The program will crash with a segmentation fault.
D) Running this program will cause permanent damage to the RAM on your computer.
E) The program may appear to run bug-free for small test cases.
F) The program will produce different output on different hardware or different operating systems.
G) Rewriting the code to use the `auto_ptr` or `unique_ptr` will fix all memory leaks.

Solution: E

3.5 Diagramming Memory

Following the conventions from lecture, draw a diagram of the stack & heap at the end of these commands:

Please be neat!

```cpp
int a = 2;
int* b = new int;
int** c[2];
int* d;
c[1] = new int*;
c[0] = &b;
d = new int[a];
*c[1] = &d[0];
d[1] = a;
**c[1] = 0;
**c[0] = a*3;
(*c[1])++;
a--;```

Using all 4 variables, print the current year to `std::cout`.

Solution:

```cpp
std::cout << **c[1] << d[0] << a << *b << std::endl;
```

Now, write code to clean up all dynamically allocated memory (so we don’t have any memory leaks).

Solution:

```cpp
delete b;
delete [] d;
delete c[1];
```

4 Count Odd Find BST

Write a recursive function that takes in a pointer to the root node in a binary search tree, and a value to search for in the tree. The function should return -1 if the value is not present in the tree. If the value is found, it should return a count of the odd numbers on the path from root to value.

Solution:

```cpp
int count_odd_find(Node *root, int value) {
    if (root == NULL) return -1;
    if (root->value == value) return 0;
    int ret;
    if (root->value > value) {
        ret = count_odd_find(root->left,value);
    } else {
        ret = count_odd_find(root->right,value);
    }
    if (ret != -1 & root->value % 2 == 1) {
        ret += 1;
    }
}
```
What is the order notation for the running time of this function given a tree with \( n \) elements? Consider the best case, worst case, and average case. Write a concise and well-written 3-4 sentences explanation.

Solution: In the best case, the item we are searching for is at the root, or very close to the top and this code will stop quickly, \( O(1) \), even if the tree is huge. In the worst case, the tree is unbalanced, and the item we are searching for is all the way at the lowest level of the unbalanced tree, \( O(n) \). In the average case, the tree is well-balanced, with height \( \log n \), and we need to walk at least half the height of the tree, \( O(\log n) \).

5 Iterative Fruit Tree Post-Order Traversal

```cpp
class Node {
public:
  Node(const std::string &v) : value(v) {}  
  std::string value;
  std::vector<Node*> children;
};
```

What is the post-order traversal of the sample tree?

Solution:

```
fig banana grape orange peach strawberry kiwi cherry lime mango date apple
```

Now write a non-recursive function that takes in a pointer to the tree root and prints the contents in post-order. Your answer should work for any tree using this Node class (not just the sample).

Solution:

```cpp
void print_iterative(Node *root) {
  // handle an empty tree
  if (root == NULL) return;
  // start with the first child of the root of tree
  std::vector<std::pair<Node*, int>> todo;
  todo.push_back(std::make_pair(root, 0));
  while (todo.size() > 0) {
    std::pair<Node*, int> &current = todo.back();
    if (current.second < current.first->children.size()) {
      // before we can print this node, we must print all of its children
      todo.push_back(std::make_pair(current.first->children[current.second], 0));
    } else {
      // after printing all of this node's children, we can print this node
      std::cout << " " << current.first->value;
      todo.pop_back();
      if (todo.size() > 0) {
        // increment the parent's child counter
        todo.back().second++;
      }
    }
  }
}
```
In this problem, we will work with people who are either players or coaches. Some of the players are goalies. When a group of people get together, we say they form a team if they have at least one coach. If a team has at least one player who is a goalie, then we say it is a hockey team.

Here's some starter code:

```cpp
std::vector<Person*> people_a;
    people_a.push_back(new Player("goalie"));
    people_a.push_back(new Coach());
    people_a.push_back(new Player("center"));
    people_a.push_back(new Player("defense"));
std::vector<Person*> people_b;
    people_b.push_back(new Coach());
    people_b.push_back(new Coach());
    people_b.push_back(new Player("center"));
    people_b.push_back(new Player("defense"));
std::vector<Person*> people_c;
    people_c.push_back(new Player("goalie"));
    people_c.push_back(new Player("center"));
    people_c.push_back(new Player("forward"));
Team *team_a = CreateTeam(people_a);
PrintTeam("team_a",team_a);
Team *team_b = CreateTeam(people_b);
PrintTeam("team_b",team_b);
Team *team_c = CreateTeam(people_c);
PrintTeam("team_c",team_c);
```

Which produces this output:

```
team_a is a hockey team
WARNING! This group of people does not have a goalie!
team_b is a team
WARNING! This group of people does not have a coach!
WARNING! This group of people does not have a coach!
team_c is just a group of people
```

Here are few helper functions used by the code above:

```cpp
Team* CreateTeam(const std::vector<Person*>& people) {
    Team* answer = NULL;
    try {
        answer = new Hockey(people);
    } catch (const std::string& warning) {
        std::cerr << "WARNING! " << warning << std::endl;
        try {
            answer = new Team(people);
        } catch (const std::string& warning) {
            std::cerr << "WARNING! " << warning << std::endl;
        }
    }
    return answer;
}
void PrintTeam(const std::string &name, Team* team) {
    if (dynamic_cast<Hockey*>(team) != NULL) {
        std::cout << name << " is a hockey team" << std::endl;
    } else if (team != NULL) {
        std::cout << name << " is a team" << std::endl;
    } else {
        std::cout << name << " is just a group of people" << std::endl;
    }
}
```
6.1 Class Inheritance Diagram

First, draw the class inheritance hierarchy for the 5 classes used in this problem.

Solution:

```
Person
 /  \\
|    |
Coach  Player
   /   \
  Team  Hockey
```

6.2 Implementation

Now, let’s complete the implementation of the 5 classes used in the code above.

```cpp
class Person {
public:
    virtual ~Person() {}
};

class Coach : public Person {
};

class Player : public Person {
public:
    Player(const std::string &t) : type(t) {}
    bool isGoalie() const { return type == "goalie"; }
private:
    std::string type;
};

int numCoaches(const std::vector<Person*>& members) {
    int answer = 0;
    for (int i = 0; i < members.size(); i++)
        if (dynamic_cast<Coach*>(members[i]) != NULL) answer++;
    return answer;
}

int numGoalies(const std::vector<Person*>& members) {
    int answer = 0;
    for (int i = 0; i < members.size(); i++) {
        Player *player = dynamic_cast<Player*>(members[i]);
        if (player != NULL && player->isGoalie()) answer++;
    }
    return answer;
}

class Team {
public:
    Team(const std::vector<Person*>& members) {
        if (numCoaches(members) == 0)
            throw(std::string("This group of people does not have a coach!"));
    }
    virtual ~Team() {}
};

class Hockey : public Team {
public:
    Hockey(const std::vector<Person*>& members) : Team(members) {
        if (numGoalies(members) == 0)
            throw(std::string("This group of people does not have a goalie!"));
    }
};
```
## 7 Lightning Round Terminology

Place the letter for each of the following terms next to the correct definition below. Each letter should be used exactly once. Please write your answers clearly and neatly.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>A) object-oriented programming</td>
<td>should be used when classes have common member data and member functions</td>
</tr>
<tr>
<td>B) operator overloading</td>
<td>use of classes to store data and define associated functions</td>
</tr>
<tr>
<td>C) dynamic memory allocation</td>
<td>can cause deadlock</td>
</tr>
<tr>
<td>D) throw exception</td>
<td>a class with an overloaded function call operator</td>
</tr>
<tr>
<td>E) iterator</td>
<td>an alternative to subscripting for vectors</td>
</tr>
<tr>
<td>F) open addressing</td>
<td>hash table collision resolution method</td>
</tr>
<tr>
<td>G) red-black trees</td>
<td>should only be used when the meaning is intuitively clear</td>
</tr>
<tr>
<td>H) inheritance</td>
<td>something we haven’t taught, and you weren’t allowed to use</td>
</tr>
<tr>
<td>I) templated class</td>
<td>guarantees $O(\log n)$ performance of the STL map operations</td>
</tr>
<tr>
<td>J) recursion</td>
<td>this keyword facilitates multiple inheritance involving the diamond property</td>
</tr>
<tr>
<td>K) polymorphism</td>
<td>the only way a constructor can fail (other than crash)</td>
</tr>
<tr>
<td>L) smart pointers</td>
<td>useful for creating custom containers that can hold a type specified at compile time</td>
</tr>
<tr>
<td>M) functor</td>
<td>allows access to private and protected information</td>
</tr>
<tr>
<td>N) goto</td>
<td>instead of placing data on the stack, it is stored on the heap</td>
</tr>
<tr>
<td>O) breakpoint</td>
<td>allows us to store pointers to different types in the same vector</td>
</tr>
<tr>
<td>P) mutex</td>
<td>can be used to solve Paint-by-Pairs puzzles</td>
</tr>
<tr>
<td>Q) virtual</td>
<td>when you’re debugging, you might add one of these</td>
</tr>
<tr>
<td>R) friend</td>
<td>uses reference counting to automate object deletion</td>
</tr>
</tbody>
</table>

## 8 Short Answer

### 8.1 Comparing Vectors & Arrays

The statements below can be used to compare and contrast arrays and vectors. For each statement, specify “ARRAY” if it is only true for arrays, “VECTOR” if it is only true for vectors, “BOTH” if it is true for both types, and “NEITHER” if it is true for neither type.

**Solution: VECTOR**

- Knows how many elements it contains.

**Solution: BOTH**

- Can be used to store elements of any type.

**Solution: NEITHER**

- Prevents access of memory beyond its bounds.

**Solution: VECTOR**

- Is dynamically re-sizable.

**Solution: BOTH**

- Can be passed by reference.
8.2 Limited Looping [3]

**True or False** There are some algorithms that must be written using a **for** loop and **cannot** be written using a **while** or **do–while** loop.

Solution: False. All looping constructs are equivalent and any algorithm written using one looping construct can be re-written with the others.

9 Superhero Division [14]

In this problem you will add a new operator to the **Superhero** class from lab. Remember that a superhero has a name, a true identity, and a power, but we **cannot** access the true identity of a **Superhero** object from the public interface. Here is the basic **Superhero** class declaration:

```cpp
class Superhero {
public:
    // ACCESSORS
    const string& getName() const { return name; }
    const string& getPower() const { return power; }
    // INPUT STREAM OPERATOR
    friend istream& operator>>(istream &istr, Superhero &hero);
private:
    // REPRESENTATION
    string name;
    string true_identity;
    string power;
};
// OUTPUT STREAM OPERATOR
ostream& operator<<(ostream &ostr, const Superhero &hero);
```

And here is part of the **Superhero** class implementation:

```cpp
ostream& operator<<(ostream &ostr, const Superhero &hero) {
    if (hero.getPower() == "")
        ostr << hero.getName() << " has no power" << endl;
    else
        ostr << "Superhero " << hero.getName() << " has power " << hero.getPower() << endl;
    return ostr;
}
```

Now let’s define the `/=` operator on **Superhero**. This operator can be used to defeat a hero by dividing them from their true identity. If an attacker learns a hero’s true identity and uses it against them, the superhero loses his power. A superhero must carefully guard his true identity to prevent this attack. If the attacker does not know and just incorrectly guesses the superhero’s true identity, this `/=` operation does nothing. For example, suppose **elastigirl** is a **Superhero** object with name equal to “Elastigirl”, true identity equal to “Zoe”, and power equal to “Flexible”. Then the statement:

```cpp
cout << elastigirl;
```

would print this on the screen:

```
Superhero Elastigirl has power Flexible
```

But after executing the statement:

```cpp
elastigirl /= ("Zoe");
```

the output of the variable **elastigirl** would print on the screen as:

```
Elastigirl has no power
```
9.1 Implementation Choices

Name the three different ways we can implement operator overloading. Which of these three is the most appropriate choice for the \( /= \) operator described above? Why?

Solution: The three methods are non-member function, member function, and friend function. It is usually preferable to consider the methods in that order. In this case we cannot implement the \( /= \) operator as a non-member because it requires read access to \texttt{true_identity} and write access to \texttt{power}. We can implement it as a member function of the \texttt{Superhero} class because the first argument is of type \texttt{Superhero}.

9.2 \( /= \) operator implementation

Now implement the \( /= \) operator. Part of your job is to carefully define the prototype for this function. What should be added or changed in the \texttt{superhero.h} class declaration file? And what should be added or changed in the \texttt{superhero.cpp} class implementation file? Be specific.

Solution: The following prototype for the member function should be added to the \textit{public} portion of the \texttt{Superhero} class declaration:

\[
\texttt{Superhero& operator/=(const string &id);}
\]

And the function definition is added to the class implementation file:

\[
\begin{align*}
\text{Superhero& \\ 
\texttt{Superhero::operator/=(const string &id) \{}
\texttt{if \ (id == \texttt{true_identity}) \{}
\texttt{power = "";} \\
\texttt{return *this;} \\
\texttt{\}}
\end{align*}
\]

10 Valet Parking Maps

You have been asked to help with a valet parking system for a big city hotel. The hotel must keep track of all of the cars currently stored in their parking garage and the names of the owners of each car. Please read through the entire question before working on any of the subproblems. Here is the simple \texttt{Car} class they have created to store the basic information about a car:

\[
\begin{align*}
\text{class Car \{}
\texttt{public:\}
\texttt{ // CONSTRUCTOR}
\texttt{Car(const string &m, const string &c) : \texttt{maker}(m), \texttt{color}(c) \{\}}
\texttt{ // ACCESSORS}
\texttt{ const string& getMaker() const \{ return \texttt{maker}; \}}
\texttt{ const string& getColor() const \{ return \texttt{color}; \}}
\texttt{private:\}
\texttt{ // REPRESENTATION}
\texttt{ string \texttt{maker};}
\texttt{ string \texttt{color};}
\texttt{\}};
\end{align*}
\]

The hotel staff have decided to build their parking valet system using a map between the cars and the owners. This map data structure will allow quick lookup of the owners for all the cars of a particular color and maker (e.g., the owners of all of the silver Hondas in the garage). For example, here is their data structure and how it is initialized to store data about the six cars currently in the garage.

\[
\begin{align*}
\text{map<Car, vector<string>> \& cars;}
\texttt{cars[Car("Honda","blue")].push\_back("Cathy");}
\texttt{cars[Car("Honda","silver")].push\_back("Fred");}
\end{align*}
\]
The managers also need a function to create a report listing all of the cars in the garage. The statement:

```
print_cars(cars);
```

will result in this report being printed to the screen (std::cout):

```
People who drive a silver Audi:
  Dan
  Erin
People who drive a blue Honda:
  Cathy
People who drive a silver Honda:
  Fred
  Bob
People who drive a green Toyota:
  Alice
```

Note how the report is sorted alphabetically by maker, then by car color, and that the owners with similar cars are listed chronologically (the order in which they parked in the garage).

### 10.1 The Car class [ /6]

In order for the Car class to be used as the first part of a map data structure, what additional non-member function is necessary? Write that function. Carefully specify the function prototype (using const & reference as appropriate). Use the example above as a guide.

**Solution:** We must define `operator<` for Car objects so that we can sort the keys of the map.

```cpp
bool operator<(const Car &a, const Car &b) {
  return (a.getMaker() < b.getMaker() ||
          (a.getMaker() == b.getMaker() && a.getColor() < b.getColor()));
}
```

### 10.2 Data structure diagram [ /10]

Draw a picture of the map data structure stored by the cars variable in the example. As much as possible use the conventions from lecture for drawing these pictures. Please be neat when drawing the picture.

**Optional:** You may also write a few concise sentences to explain your picture.

**Solution:**

![Data structure diagram](image)

### 10.3 print_cars [ /9]

Write the `print_cars` function. Part of your job is to correctly specify the prototype for this function. Be sure to use const and pass by reference as appropriate.

**Solution:**

```cpp
void print_cars(const map<Car, vector<string> > &cars) {
  map<Car, vector<string> >::const_iterator itr = cars.begin();
  while (itr != cars.end()) {
    cout << "People who drive a " << itr->first.getMaker() << " " << itr->second[0] << ":
        ";
    for (int i = 1; i < itr->second.size(); i++)
      cout << itr->second[i] << " ";
    cout << endl;
    itr++; // Move to next iterator
  }
}
```
Car c = itr->first;
cout << "People who drive a " << c.getColor() << " " << c.getMaker() << ":" << endl;
vector<string>::const_iterator itr2 = itr->second.begin();
while (itr2 != itr->second.end()) {
    cout << " " << *itr2 << endl;
    itr2++;
}
itr++;
}

10.4 removeCars [ /13]

When guests pick up their cars from the garage, the data structure must be correctly updated to reflect this change. The remove_car function returns true if the specified car is present in the garage and false otherwise.

```cpp
bool success;
success = remove_car(cars, "Erin", "silver", "Audi");
assert (success == true);
success = remove_car(cars, "Cathy", "blue", "Honda");
assert (success == true);
success = remove_car(cars, "Sally", "green", "Toyota");
assert (success == false);
```

After executing the above statements the cars data structure will print out like this:

```
People who drive a silver Audi:
    Dan
People who drive a silver Honda:
    Fred
    Bob
People who drive a green Toyota:
    Alice
```

Note that once the only blue Honda stored in the garage has been removed, this color/maker combination is completely removed from the data structure.

Specify the prototype and implement the remove_car function.

Solution:

```cpp
bool remove_car(map<Car, vector<string> > &cars, const string &name, const string &color, const string &maker) {
    map<Car, vector<string> >::iterator itr = cars.find(Car(maker, color));
    if (itr == cars.end()) return false;
    if (itr->second.size() == 1 && itr->second[0] == name) {
        cars.erase(Car(maker, color));
        return true;
    }
    for (int i = 0; i < itr->second.size(); i++) {
        if (itr->second[i] == name) {
            itr->second.erase(itr->second.begin() + i);
            return true;
        }
    }
    return false;
}
```

11 Computational Desert Island [ / ]

Suppose that a monster is holding you captive on a computational desert island, and has a large file containing double precision numbers that he needs to have sorted. If you write correct code to sort his numbers he will release
you and when you return home will be allowed to move on to DSA. If you don’t write correct code, he will eventually release you, but only under the condition that you retake CS 1. The stakes indeed are high, but you are quietly confident — you know about the standard library sort function. (Remember, you are supposed to have forgotten all about bubble sort.) The monster startles you by reminding you that this is a computational desert island and because of this the only data structure you have to work with is a queue.

After panicking a bit (or a lot), you calm down and think about the problem. You realize that if you maintain the values in the queue in increasing order, and insert each value into the queue one at a time, then you can solve the rest of the problem easily. Therefore, you must write a function that takes a new double, stored in x, and stores it in the queue. Before the function is called, the values in the queue are in increasing order. After the function ends, the values in the queue must also be in increasing order, but the new value must also be among them.

Here is the function prototype:

```cpp
void insert_in_order(double x, queue<double>& q)
```

You may only use the public queue interface (member functions) as specified in lab. You may use a second queue as local variable scratch space or you may try to do it in a single queue (which is a bit harder). Give an “O” estimate of the number of operations required by this function.

**Solution:** Here is a version with a scratch queue:

```cpp
void insert_in_order(double x, queue<double>& q) {
    if (q.empty())
        q.push(x);
    else {
        queue<double> temp(q);            // copy q;
        while (!q.empty()) q.pop();        // empty q out
        while (!temp.empty() && x > temp.front()) {
            double item = temp.front();
            temp.pop();
            q.push(item);
        }
        q.push(x);                        // insert x in its proper position
        while (!temp.empty()) {
            double item = temp.front()
            temp.pop();
            q.push(item);
        }
    }
}
```

This function requires \(O(n)\) operations. Copying the queue initially and emptying the queue requires \(O(n)\) time each. The second and third while loops, combined, touch each entry in the queue and therefore require \(O(n)\) operations. Since none of these loops are nested we add the results and get \(O(n)\) time overall.

Here is a version without a scratch queue:

```cpp
void insert_in_order(int x, queue<double>& q) {
    int n = q.size();
    int position = 0;
    // Find the position for x in the queue, copying all values
    // less than x to the back of the queue
    while (position < n && x < q.front()) {
        q.push(q.front());     // copy the front to the back
        q.pop();               // remove the front
        ++position;
    }
    q.push(x);              // put x in position
```
The first n-position entries on the queue haven't been touched and are greater than or equal to the value stored in x. They need to move to the back of the queue.

```c++
int i = position;
while (i < n) {
    q.push(q.front()); // copy the front to the back
    q.pop(); // remove the front
    ++i;
}
```

The two while loops, combined touch each entry in the queue once and therefore require $O(n)$ operations. Since none of these loops are nested we add the results and get $O(n)$ time overall.

## 12 Operations on Lists

### 12.1 Reversing a dslist

Write a `dslist<T>` member function called `reverse` that reverses the order of the nodes in the list. The head pointer should point to what was the tail node and the tail pointer should point to what was the head node. All directions of pointers should be reversed. The function prototype is:

```c++
template <class T> void dslist<T>::reverse();
```

The function must NOT create ANY new nodes.

**Solution:**

```c++
template <class T>
void cs2list<T>::reverse() {
    // Handle empty or single node list
    if (head_ == tail_) return;
    // Swap pointers at each node of the list, using a temporary pointer q to remember where to go next.
    Node<T> *p = head_;
    while (p) {
        Node<T> *q = p->next_;
        p->next_ = p->prev_;
        p->prev_ = q;
        p = q;
    }
    // Swap head and tail pointers
    p = head_;
    head_ = tail_;
    tail_ = p;
}
```

### 12.2 Sublists

Write a function to create a new singly-linked list that is a copy of a sublist of an existing list. The prototype is:

```c++
Node<T> * Sublist(Node<T> * head, int low, int high)
```

The `Node` class is:

```c++
template <class T>
class Node {
public:
    T value;
    Node* next;
};
```
The new list will contain high-low+1 nodes, which are copies of the values in the nodes occupying positions low up through and including high of the list pointed to by head. The function should return the pointer to the first node in the new list. For example, in the following drawing the original list is shown on top and the new list created by the function when low=2 and high=4 is shown below.

Original list

New list

A pointer to the first node of this new list should be returned. (In the drawing this would be the value of nhead.) You may assume the original list contains at least low nodes. If it contains fewer than high nodes, then stop copying at the end of the original list.

Solution:

Node<T>* Sublist(Node<T>* head, int low, int high) {
    // Skip over the first low-1 nodes in the existing list
    Node<T>* p = head;
    int i;
    for (i=1; i<low; ++i) p = p->next;
    // Make the new head node and make a pointer to the last node in list
    Node<T>* new_head = new Node<T>;
    new_head->value = p->value;
    Node<T> * last = new_head;
    // Copy the remaining nodes, one at a time
    for (++i, p = p->next; i<=high && p; ++i, p = p->next) {
        last->next = new Node<T>;
        last->next->value = p->value;
        last = last->next;
    }
    last->next = 0;
}

12.3 Splicing into a cs2list[ ]

Write a cs2list<T> member function called splice that takes an iterator and a second cs2list<T> object and splices the entire contents of the second list between the node pointed to by the iterator and its successor node. The second list must be completely empty afterwards. The function prototype is:

```cpp
template <class T>
void cs2list<T>::splice(iterator itr, cs2list<T>& second);
```

No new nodes should be created by this function AND it should work in O(1) time (i.e. it should be independent of the size of either list).

Solution:

```cpp
template <class T>
void cs2list<T>::splice(iterator itr, cs2list<T>& second) {
    if (second.empty()) return;
    second.head_->prev_ = itr.ptr_;
    second.tail_->next_ = itr.ptr_->next_;
```
if (itr.ptr_->next_) {
    itr.ptr_->next_->prev_ = second.tail_;  
} else { // itr.ptr_ is the tail, so it must be reset
    this->tail_ = second.tail_;  
}
itr.ptr_->next_ = second.head_;  
this->size_ += second.size_;  
second.size_ = 0;  
second.head_ = second.tail_ = 0;

13   Concurrency and Asynchronous Computing [   /3]
Why might a group of dining philosophers starve?

A) Because it’s impossible to eat spaghetti with chopsticks.
B) Because they are all left-handed.
C) Because due to a bank error they didn’t have enough money in their joint account.
D) Because they didn’t all want to eat at the same time.

Solution: B

14   Garbage Collection  [   /12]
For each of the real world systems described below, choose the most appropriate memory management technique. Each technique should be used exactly once.

A) Explicit Memory Management (C++)  B) Reference Counting
C) Stop & Copy  D) Mark-Sweep

14.1   Student Registration System [   /3]
Must handle the allocation and shuffling of pointers as students register and transfer in and out of classes. Memory usage will not be a deciding factor. Fragmentation of data should be minimized.

Solution: C

14.2   Playing Chess  [   /3]
Implementation of a tree-based algorithm for searching the game space. Remember that a tree is a graph with no cycles.

Solution: B

14.3   Webserver  [   /3]
A collection of infrequently changing interconnected webpages. Any memory usage overhead should be low. Pauses in service are tolerable.

Solution: D

14.4   Hand Held Game (e.g., GameBoy or PSP, etc.) [   /3]
Performance critical application with extremely limited memory resources.
15 Short Answer [22/22]

15.1 Garbage Identification [7]
To which address in the memory below should the root variable point so that exactly 2 cells are garbage? Draw a box and pointer diagram to justify your answer and state which 2 cells are garbage.

<table>
<thead>
<tr>
<th>address</th>
<th>100</th>
<th>101</th>
<th>102</th>
<th>103</th>
<th>104</th>
<th>105</th>
<th>106</th>
<th>107</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
</tr>
<tr>
<td>left</td>
<td>106</td>
<td>106</td>
<td>107</td>
<td>100</td>
<td>102</td>
<td>101</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>right</td>
<td>103</td>
<td>105</td>
<td>105</td>
<td>0</td>
<td>105</td>
<td>101</td>
<td>0</td>
<td>101</td>
</tr>
</tbody>
</table>

Solution: The root should point to cell 104.
Cells 100 and 103 are garbage.

15.2 Stop and Copy Garbage Collection [4]
What is the purpose of the forwarding address in Stop and Copy garbage collection? What will go wrong if you neglect to record this value? Write 2 or 3 concise and well-written sentences.

Solution: When a non-garbage cell is copied from the old memory partition to the new memory partition, a forwarding address is left to indicate that the cell has been copied and to provide the address of the new location. All references to the old location will see the forwarding address and can then be updated as appropriate. If we don’t record this forwarding address cyclical data structures will not be copied correctly: the garbage collector will repeatedly copy the same cell resulting in an infinite loop!

15.3 Concurrency and Asynchronous Computing [4]
When programming with multiple threads or processes, the correct use of mutexes (locks) and condition variables will ensure that:

A) The program always returns the exact same answer.

B) The program returns an answer that was not possible if the program ran sequentially.

C) The entire program is atomic.

D) Each student in a large class will be able to successfully copy a complete set of lecture notes (with no repetitions), even if there are multiple professors.

E) Deadlock will be avoided if there are multiple mutexes, but may still happen in systems with a single lock.

Solution: D

15.4 Perfect Hashing for Image Compression [7]
For the last homework, you implemented a compression scheme for 2D images. What are the drawbacks of using this format as the underlying representation for an image editing program? What types of edits to the image are simple? What types of edits will be comparatively inefficient to process? Write 3-4 concise and well-written sentences.
Solution: The underlying representation is geared towards a static image because the computation of the perfect hash depends on the occupancy data. Simply changing the color of some or all of the non-white pixels is cheap. Removing non-white pixels (painting them white) or adding a non-white pixel that happens to hash to an empty spot in the hash data table is also cheap. Adding non-white pixels that collide with other pixels requires recomputation of the offset table, and possibly resizing of the hash data and/or offset tables.

16 Data Structures [ /18]

Indicate by letter the data structure(s) that have each characteristic listed below.

<table>
<thead>
<tr>
<th>A) vector</th>
<th>B) list</th>
<th>C) map</th>
<th>D) set</th>
<th>E) priority queue</th>
<th>F) hash table</th>
<th>G) leftist heap</th>
</tr>
</thead>
</table>

Solution: B C D allows efficient (sublinear) removal of the first and last elements (or the minimum and maximum elements)

A E F uses an array or vector as the underlying representation

B C D (F) G uses a network of nodes connected by pointers as the underlying representation

C D (E) F the underlying data structure must be “balanced” or well-distributed to achieve the targeted performance

C D E G requires definition of operator< or operator>

C D E F G entries cannot be modified after they are inserted (requires re-insertion or re-processing of position)

C D (F) duplicates are not allowed

B G allows sublinear merging of two of instances of this data structure

17 Order Notation [ /16]

Match the order notation with each fragment of code. Two of the letters will not be used.

<table>
<thead>
<tr>
<th>A) O(n)</th>
<th>B) O(1)</th>
<th>C) O(n^3)</th>
<th>D) O(n^2)</th>
<th>E) O(2^n)</th>
<th>F) O(log n)</th>
<th>G) O(n log n)</th>
<th>H) O(√n)</th>
</tr>
</thead>
</table>

vector<int> my_vector;
// my_vector is initialized with n entries
// do not include initialization in performance analysis
for (int i = 0; i < n; i++) {
    my_vector.erase(my_vector.begin());
}

Solution: D

map<string,int> my_map;
// my_map is initialized with n entries
// do not include initialization in performance analysis
my_map.find("hello");

Solution: F

int foo(int n) {
    if (n == 1 || n == 0) return 1;
    return foo(n-1) + foo(n-2);
}

Solution: E

int k = 0;
for (int i = 0; i < sqrt(n); i++) {
    for (int j = 0; j < sqrt(n); j++) {
        k += i*j;
    }
}

Solution: A
Solution: G

```cpp
set<string> my_set;
for (int i = 0; i < n; i++) {
    string s;
    cin >> s;
    my_set.insert(s);
}
```

Solution: B

```cpp
float* my_array = new float[n];
// do not include memory allocation in performance analysis
my_array[n/2] = sqrt(n);
```

18 Office Demolition [ /31]

In this problem we will explore a simple class to manage the assignment of people to offices and desks. Each Office object stores its name, the number of desks it can hold, and the names of the people assigned to those desks. An office also stores a reference to a master queue of all the people who still need to be assigned to desks. When an office is constructed, people are assigned to the office from the front of this master queue. When an office is demolished, the people who were assigned to that office should be added to the end of the queue while they wait for a new office assignment. Here is the partial declaration of the Office class:

```cpp
class Office {
public:
    Office(const string& name, int num_desks, queue<string> &unassigned);
    friend ostream& operator<<(ostream &ostr, const Office &office);
private:
    // representation
    string _name;
    int _num_desks;
    string* _desks;
    queue<string>& _unassigned; // a reference to the master queue
};
```

In the example below we create the master queue of people who need to be assigned to desks in offices, and create and delete several Office objects:

```cpp
queue<string> unassigned;
unassigned.push("Alice");
unassigned.push("Bob");
unassigned.push("Cathy");
unassigned.push("Dan");
unassigned.push("Erin");
unassigned.push("Fred");
unassigned.push("Ginny");

Office *red = new Office("red", 4, unassigned);
Office *green = new Office("green", 2, unassigned);
cout << *red << *green;

delete red;
cout << "After deleting the red office, "
    << unassigned.size() << " people are waiting for desks." << endl;

Office *blue = new Office("blue", 3, unassigned);
cout << *blue;

cout << "Before deleting the blue & green offices, "
    << unassigned.size() << " people are waiting for desks." << endl;
delete green;
delete blue;
cout << "After deleting all of the offices, "
    << unassigned.size() << " people are waiting for desks." << endl;
```

Here is the desired output from this example:

```plaintext
20
```
The red office has 4 desks:
  desk[0] = Alice
  desk[1] = Bob
  desk[2] = Cathy
  desk[3] = Dan
The green office has 2 desks:
  desk[0] = Erin
  desk[1] = Fred
After deleting the red office, 5 people are waiting for desks.
The blue office has 3 desks:
  desk[0] = Ginny
  desk[1] = Alice
  desk[2] = Bob
Before deleting the blue & green offices, 2 people are waiting for desks.
After deleting all of the offices, 7 people are waiting for desks.

Here is the implementation of the constructor, as it appears in the office.cpp file:

Office::Office(const string& name, int num_desks, queue<string> &unassigned)
  : _name(name), _num_desks(num_desks), _unassigned(unassigned) {
    _desks = new string[_num_desks]; // allocate the desk space
    for (int i = 0; i < _num_desks; i++) {
      if (_unassigned.size() > 0) { // assign from the master queue
        _desks[i] = _unassigned.front();
        _unassigned.pop();
      } else { // if there are no unassigned people, leave the desk empty
        _desks[i] = "";
      }
    }
  }

18.1 Classes and Memory Allocation [ /10]

Anytime you write a new class, especially those with dynamically allocated memory, it is very important to consider the member functions that the compiler will automatically generate and determine if this default behavior is appropriate. List these 4 important functions by their generic names, AND write their prototypes as they would appear within the Office class declaration.

Solution:

default constructor Office();
destructor ~Office();
copy constructor Office(const Office &office);
assignment operator Office& operator=(const Office &office);

18.2 Declaring a Destructor [ /3]

The Office class is incomplete and requires implementation of a custom destructor so that people assigned to demolished offices are returned to the master queue and memory is deallocated as appropriate to avoid memory leaks. What line needs to be added to the header file to declare the destructor? Be precise with syntax. Where should this line be added: within the public, protected, or private interface?

Solution: Anywhere in the public: interface,

  ~Office();

18.3 Implementing a Destructor [ /12]

Implement the destructor, as it would appear in the office.cpp file.

Solution:
Office::~Office() {
    for (int i = 0; i < _num_desks; i++) {
        if (_desks[i] != "") {
            _unassigned.push(_desks[i]);
        }
    }
    delete [] _desks;
}

18.4 Operator Overloading [ /6]

Here is the implementation of the << stream operator as it appears within the office.cpp file:

    ostream& operator<<(ostream &ostr, const Office &o) {
        ostr << "The " << o._name << " office has "
             << o._num_desks << " desks:" << endl;
        for (int i = 0; i < o._num_desks; i++) {
            ostr << " desk[" << i << "] = " << o._desks[i] << endl;
        }
        return ostr;
    }

There are three different ways to overload an operator: as a non-member function, as a member function, and as a friend function. Which method was selected for the Office object << stream operator? What are the reasons for this choice? Discuss why the other two methods are inappropriate or undesirable. Write 3 or 4 concise and thoughtful sentences.

Solution: This operator has been implemented as a friend function, which is necessary to gain access to the private member variables of the Office object. If we had implemented it as a non-member function the function wouldn’t have access to these variables and accessor functions would need to be added to the public interface, which is undesirable since that would expose the private representation unnecessarily. We cannot implement the stream operator as a member function of the Office class because the << syntax requires the ostream object to be the first argument of the operator. In order to be written as a member operator of a particular class, the first argument must be of that class type.

19 Dynamically-Allocated Arrays [ /17]

Write a function that takes an STL list of integers, finds the even numbers, and places them in a dynamically-allocated array. Only the space needed for the even numbers should be allocated, and no containers other than the given list and the newly-created array may be used. As an example, given a list containing the values:

    3 10 -1 5 6 9 13 14

the function should allocate an array of size 3 and store the values 10, 6 and 14 in it. It should return, via arguments, both the pointer to the start of the array and the number of values stored. No subscripting may be used — not even *(a+i) in place of a[i]. Here is the function prototype:

    void even_array(const list<int>& b, int* & a, int& n);

Solution:

    void even_array(const list<int>& b, int* & a, int& n) {
        n = 0; // count
        for (list<int>::const_iterator p = b.begin(); p!=b.end(); ++p)
            if (*p % 2 == 0) ++n;
        a = new int[n]; // allocate
        int* q = a; // store in array
        for (list<int>::const_iterator p = b.begin(); p!=b.end(); ++p)
if (*p % 2 == 0) {
    *q = *p;
    ++q;
}

20  Ternary Tree Recursion [ /17]
A ternary tree is similar to a binary tree except that each node has at most 3 children. Write a recursive function named EqualsChildrenSum that takes one argument, a pointer to the root of a ternary tree, and returns true if the value at each non-leaf node is the sum of the values of all of its children and false otherwise. In the examples below, the tree on the left will return true and the tree on the right will return false.

class Node {
public:
    int value;
    Node* left;
    Node* middle;
    Node* right;
};

Solution:

bool EqualsChildrenSum(Node *node) {
    if (node == NULL) return true;
    if (node->left == NULL && node->middle == NULL && node->right == NULL)
        return true;
    int sum = 0;
    if (node->left != NULL) sum += node->left->value;
    if (node->middle != NULL) sum += node->middle->value;
    if (node->right != NULL) sum += node->right->value;
    if (sum != node->value) return false;
    return
    EqualsChildrenSum(node->left) &&
    EqualsChildrenSum(node->middle) &&
    EqualsChildrenSum(node->right);
}

21  Priority Queues [ /16]

template <class T> class priority_queue {
public:
    // CONSTRUCTOR
    priority_queue() {} 
    // ACCESSORS
    int size() { return m_heap.size(); } 
    bool empty() { return m_heap.empty(); } 
    const T& top() const { assert(!m_heap.empty()); return m_heap[0]; } 
    // MODIFIERS
    void push(const T& entry) {
        m_heap.push_back(entry);
        this->percolate_up(int(m_heap.size()-1));
    }
    void pop() { // find and remove the element with the smallest value
        assert(!m_heap.empty());
        m_heap[0] = m_heap.back();
        m_heap.pop_back();
        this->percolate_down(0);
    }
    void pop_max() { /* YOU WILL IMPLEMENT THIS FUNCTION */ }
private:
    /* YOU WILL IMPLEMENT THIS FUNCTION */
}
21.1 Implementing `pop_max` [12]

Write the new priority queue member function named `pop_max` that finds and removes from the queue the element with the largest value. Carefully think about the efficiency of your implementation. Remember that a standard priority queue stores the smallest value element at the root.

Solution:

```cpp
void pop_max() {
    assert(!m_heap.empty());
    int tmp = (size()+1)/2;
    for (int i = tmp+1; i < size(); i++) {
        if (m_heap[tmp] < m_heap[i]) {
            tmp = i;
        }
    }
    m_heap[tmp] = m_heap.back();
    m_heap.pop_back();
    this->percolate_up(tmp);
}
```

21.2 Analysis [4]

If there are \( n \) elements in the priority queue, how many elements are visited by the `pop_max` function in the worst case? What is the order notation for the running time of this function?

Solution: \( n + \log n \) elements are visited. Running time is \( O(n) \).

22 Inheritance & Polymorphism [10]

What is the output of the following program?

```cpp
class A {
```
```cpp
public:
    virtual void f() { cout << "A::f\n"; }
    void g() { cout << "A::g\n"; }
};

class B : public A {
public:
    void g() { cout << "B::g\n"; }
};

class C : public B {
public:
    void f() { cout << "C::f\n"; }
    void g() { cout << "C::g\n"; }
};

int main() {
    A* a[3];
    a[0] = new A();
    a[1] = new B();
    a[2] = new C();

    for (int i = 0; i < 3; i++) {
        cout << i << endl;
        a[i]->f();
        B* b = dynamic_cast<B*>(a[i]);
        if (b) b->g();
    }
}
```

Solution:

```
0
A::f
1
A::f
B::g
2
C::f
B::g
```

23 Types & Values [ /15]

For the last expression in each fragment of code below, give the type (int, vector<double>, Foo*, etc.) and the value. If the value is a legal address in memory, write “memory address”. If the value hasn’t been properly initialized, write “uninitialized”. If there is an error in the code, write “error”. You may want to draw a picture to help you answer each question, but credit will only be given for what you’ve written in the boxes.

double a = 5.2;
double b = 7.5;
a+b

Solution: Type: double Value: 12.7

int *d;
int e[7] = { 15, 6, -7, 19, -1, 3, 22 };
d = e + e[5];
*d

Solution: Type: int Value: 19
bool *f = new bool;
*f = false;
f

Solution: Type: bool* Value: memory address

int g = 10;
int *h = new int[g];
h[0]

Solution: Type: int Value: uninitialized

map<string, int> m;
m.insert(make_pair(string("bob"),5551111));
m.insert(make_pair(string("dave"),5552222));
m.insert(make_pair(string("alice"),5553333));
m.insert(make_pair(string("chris"),5554444));
(;++m.find("bob")]->second

Solution: Type: int Value: 5554444