

Project 5

● Graded

Student

Devin Chen

Total Points

95 / 100 pts

Autograder Score

80.0 / 80.0

Passed Tests

Test compiles (5/5)

Testing Ingredient struct (5/5)

Testing Pantry default and parameterized constructors (10/10)

Tests contains, addIngredient (10/10)

Tests canCreate() (15/15)

Tests calculatePantryValue() (5/5)

Tests printIngredient() (10/10)

Tests ingredientQuery() (10/10)

Tests pantryList() (10/10)

Question 2

Style & Documentation

15 / 20 pts

✓ + 5 pts Style

+ 5 pts Indicates name and date in comment preamble

✓ + 5 pts Has inline comments where appropriate

✓ + 5 pts Has function preambles with @pre, @post, @param, @return where appropriate

+ 20 pts No-Compile Adjustment

+ 10 pts Partial No-Compile

+ 0 pts Insufficient submission

- 100 pts Academic integrity

- 90 pts Academic integrity

- 40 pts Academic integrity

- 65 pts Academic integrity

- 30 pts Academic integrity

Autograder Results

Test compiles (5/5)

Your program compiles!

Testing Ingredient struct (5/5)

Your program passed this test.

Testing Pantry default and parameterized constructors (10/10)

Your program passed this test.

Tests contains, addIngredient (10/10)

Your program passed this test.

Tests canCreate() (15/15)

Your program passed this test.

Tests calculatePantryValue() (5/5)

Your program passed this test.

Tests printIngredient() (10/10)

Your program passed this test.

Tests ingredientQuery() (10/10)

Your program passed this test.

Tests pantryList() (10/10)

Your program passed this test.

Submitted Files

▼ .gitignore		Download
1	.DS_Store	
2	.vscode	
3	*.log	
4		
5		

```
1 // Created by Frank M. Carrano and Timothy M. Henry.
2 // Copyright (c) 2017 Pearson Education, Hoboken, New Jersey.
3 //
4 // Modified by Tiziana Ligorio for Hunter College CSCI 235
5 // modified position s.t. 0 <= position < item_count_
6 // some style modification, mainly variable names
7 // added getHeadNode() for grading purposes
8
9 /** ADT list: Singly linked list implementation.
10
11 Implementation file for the class LinkedList.
12 @file LinkedList.cpp */
13
14 #include "LinkedList.hpp" // Header file
15 #include <cassert>
16
17 // constructor
18 template<class T>
19 LinkedList<T>::LinkedList() : head_ptr_(nullptr), item_count_(0)
20 {
21 } // end default constructor
22
23
24 // copy constructor
25 template<class T>
26 LinkedList<T>::LinkedList(const LinkedList<T>& a_list) : item_count_(a_list.item_count_)
27 {
28     Node<T>* orig_chain_pointer = a_list.head_ptr_; // Points to nodes in original chain
29
30     if (orig_chain_pointer == nullptr)
31         head_ptr_ = nullptr; // Original list is empty
32     else
33     {
34         // Copy first node
35         head_ptr_ = new Node<T>();
36         head_ptr_->setItem(orig_chain_pointer->getItem());
37
38         // Copy remaining nodes
39         Node<T>* new_chain_ptr = head_ptr_; // Points to last node in new chain
40         orig_chain_pointer = orig_chain_pointer->getNext(); // Advance original-chain pointer
41         while (orig_chain_pointer != nullptr)
42         {
43             // Get next item from original chain
44             T next_item = orig_chain_pointer->getItem();
45
46             // Create a new node containing the next item
47             Node<T>* new_node_ptr = new Node<T>(next_item);
48
49             // Link new node to end of new chain
```

```

50     new_chain_ptr->setNext(new_node_ptr);
51
52     // Advance pointer to new last node
53     new_chain_ptr = new_chain_ptr->getNext();
54
55     // Advance original-chain pointer
56     orig_chain_pointer = orig_chain_pointer->getNext();
57 } // end while
58
59     new_chain_ptr->setNext(nullptr);          // Flag end of chain
60 } // end if
61 } // end copy constructor
62
63
64 // destructor
65 template<class T>
66 LinkedList<T>::~LinkedList()
67 {
68     clear();
69 } // end destructor
70
71
72
73 /**@return true if list is empty - item_count_ == 0 */
74 template<class T>
75 bool LinkedList<T>::isEmpty() const
76 {
77     return item_count_ == 0;
78 } // end isEmpty
79
80
81 /**@return the number of items in the list - item_count_ */
82 template<class T>
83 int LinkedList<T>::getLength() const
84 {
85     return item_count_;
86 } // end getLength
87
88
89
90 /**
91  @pre list positions follow traditional indexing from 0 to item_count_ -1
92  @param position indicating point of insertion
93  @param new_entry to be inserted in list
94  @post new_entry is added at position in list (the node previously at that position is now at
95  position+1)
96  @return true if valid position (0 <= position <= item_count_) */
97 template<class T>
98 bool LinkedList<T>::insert(int positions, const T& new_entry)
99 {
100     bool able_to_insert = (positions >= 0) && (positions <= item_count_);
101     if (able_to_insert)

```

```

101 {
102     // Create a new node containing the new entry
103     Node<T>* new_node_ptr = new Node<T>(new_entry);
104
105     // Attach new node to chain
106     if (positions == 0)
107     {
108         // Insert new node at beginning of chain
109         new_node_ptr->setNext(head_ptr_);
110         head_ptr_ = new_node_ptr;
111     }
112     else
113     {
114         // Find node that will be before new node
115         Node<T>* prev_ptr = getNodeAt(positions - 1);
116
117         // Insert new node after node to which prev_ptr points
118         new_node_ptr->setNext(prev_ptr->getNext());
119         prev_ptr->setNext(new_node_ptr);
120     } // end if
121
122     item_count++; // Increase count of entries
123 } // end if
124
125 return able_to_insert;
126 } // end insert
127
128
129
130 /**
131  @pre list positions follow traditional indexing from 0 to item_count_ -1
132  @param position indicating point of deletion
133  @post node at position is deleted, if any. List order is retains
134  @return true if there is a node at position to be deleted, false otherwise */
135 template<class T>
136 bool LinkedList<T>::remove(int position)
137 {
138     bool able_to_remove = (position >= 0) && (position < item_count_);
139     if (able_to_remove)
140     {
141         Node<T>* cur_ptr = nullptr;
142         if (position == 0)
143         {
144             // Remove the first node in the chain
145             cur_ptr = head_ptr_; // Save pointer to node
146             head_ptr_ = head_ptr_->getNext();
147         }
148         else
149         {
150             // Find node that is before the one to delete
151             Node<T>* prev_ptr = getNodeAt(position - 1);
152

```

```

153     // Point to node to delete
154     cur_ptr = prev_ptr->getNext();
155
156     // Disconnect indicated node from chain by connecting the
157     // prior node with the one after
158     prev_ptr->setNext(cur_ptr->getNext());
159 } // end if
160
161 // Return node to system
162 cur_ptr->setNext(nullptr);
163 delete cur_ptr;
164 cur_ptr = nullptr;
165
166 item_count--; // Decrease count of entries
167 } // end if
168
169 return able_to_remove;
170 } // end remove
171
172
173
174 /**@post the list is empty and item_count_ == 0*/
175 template<class T>
176 void LinkedList<T>::clear()
177 {
178     while (!isEmpty())
179         remove(0);
180 } // end clear
181
182
183
184 /**
185  @pre list positions follow traditional indexing from 0 to item_count_ -1
186  @param position indicating the position of the data to be retrieved
187  @return data item found at position. If position is not a valid position < item_count_
188  throws PrecondViolatedExcep */
189 template<class T>
190 T LinkedList<T>::getEntry(int position) const
191 {
192     // Enforce precondition
193     bool ableToGet = (position >= 0) && (position < item_count_);
194     if (ableToGet)
195     {
196         Node<T>* nodePtr = getNodeAt(position);
197         return nodePtr->getItem();
198     }
199     else
200     {
201         std::string message = "getEntry() called with an empty list or ";
202         message = message + "invalid position.";
203         throw (PrecondViolatedExcep(message));
204     } // end if

```

```

205 } // end getEntry
206
207
208
209
210
211 /***** PROTECTED METHODS *****/
212
213
214 // Locates a specified node in this linked list.
215 // @pre list positions follow traditional indexing from 0 to item_count_ -1
216 // @param position the index of the desired node
217 //      0 <= position < item_count_
218 // @return A pointer to the node at the given position or nullptr if position is >= item_count_
219 template<class T>
220 Node<T>* LinkedList<T>::getNodeAt(int position) const
221 {
222     // Count from the beginning of the chain
223     Node<T>* cur_ptr = head_ptr_;
224     for (int skip = 0; skip < position; skip++)
225         cur_ptr = cur_ptr->getNext();
226
227     return cur_ptr;
228 } // end getNodeAt
229
230 //position follows classic indexing from 0 to item_count_-1
231 //if position > item_count it returns nullptr
232 template <class T>
233 Node<T> *LinkedList<T>::getPointerTo(size_t position) const
234 {
235
236     Node<T> *find = nullptr;
237     if (position < item_count_)
238     {
239         find = head_ptr_;
240         for (size_t i = 0; i < position; ++i)
241         {
242             find = find->getNext();
243         }
244     }
245
246     return find;
247 } //end getPointerTo
248
249
250
251
252 //returns the head pointer
253 template <class T>
254 Node<T> *LinkedList<T>::getHeadNode() const
255 {
256

```



```
257     return head_ptr_;
258 } //end getHeadNode
259
260
261 // End of implementation file.
262
```


```
1 // Created by Frank M. Carrano and Timothy M. Henry.
2 // Copyright (c) 2017 Pearson Education, Hoboken, New Jersey.
3 //
4 // Modified by Tiziana Ligorio for Hunter College CSCI 235
5 // modified position s.t. 0 <= position < item_count_
6 // some style modification, mainly variable names
7 // added getHeadNode() for grading purposes
8
9 /** ADT list: Singly linked list implementation.
10  Listing 9-2.
11  @file LinkedList.h */
12
13 #ifndef LINKED_LIST_
14 #define LINKED_LIST_
15
16 #include "Node.hpp"
17 #include "PrecondViolatedExcep.hpp"
18
19 template<class T>
20 class LinkedList
21 {
22
23 public:
24     LinkedList(); // constructor
25     LinkedList(const LinkedList<T>& a_list); // copy constructor
26     virtual ~LinkedList(); // destructor
27
28     /**@return true if list is empty - item_count_ == 0 */
29     bool isEmpty() const;
30
31     /**@return the number of items in the list - item_count_ */
32     int getLength() const;
33
34
35     /**
36      @pre list positions follow traditional indexing from 0 to item_count_ -1
37      @param position indicating point of insertion
38      @param new_entry to be inserted in list
39      @post new_entry is added at position in list (the node previously at that position is now at
40      position+1)
41      @return true if valid position (0 <= position <= item_count_) */
42     bool insert(int position, const T& new_entry);
43
44     /**
45      @pre list positions follow traditional indexing from 0 to item_count_ -1
46      @param position indicating point of deletion
47      @post node at position is deleted, if any. List order is retains
48      @return true if there is a node at position to be deleted, false otherwise */
```

```

49     bool remove(int position);
50
51
52
53     /**@post the list is empty and item_count_ == 0*/
54     void clear();
55
56
57     /**
58      * @pre list positions follow traditional indexing from 0 to item_count_ -1
59      * @param position indicating the position of the data to be retrieved
60      * @return data item found at position. If position is not a valid position < item_count_
61      *         throws PrecondViolatedExcep */
62     T getEntry(int position) const;
63
64     //if position > item_count_ returns nullptr
65     Node<T> *getPointerTo(size_t position) const;
66
67     Node<T> *getHeadNode() const;
68
69
70
71
72
73 protected:
74     Node<T>* head_ptr_; // Pointer to first node in the chain;
75     // (contains the first entry in the list)
76     int item_count_;    // Current count of list items
77
78
79
80     // Locates a specified node in this linked list.
81     // @pre list positions follow traditional indexing from 0 to item_count_ -1
82     // @param position the index of the desired node
83     //    0 <= position < item_count_
84     // @return A pointer to the node at the given position or nullptr if position is >= item_count_
85     Node<T>* getNodeAt(int position) const;
86
87
88
89
90
91 }; // end LinkedList
92
93 #include "LinkedList.cpp"
94 #endif
95

```

▼ Makefile

 Download

```
1 CXX = g++
2 CXXFLAGS = -std=c++17 -g -Wall -O2
3
4 PROG ?= main
5 OBJS = PrecondViolatedExcep.o Pantry.o main.o
6
7 all: $(PROG)
8
9 .cpp.o:
10     $(CXX) $(CXXFLAGS) -c -o $@ $<
11
12 $(PROG): $(OBJS)
13     $(CXX) $(CXXFLAGS) -o $@ $(OBJS)
14
15 clean:
16     rm -rf $(EXEC) *.o *.out main
17
18 rebuild: clean all
19
```

```
1 // Created by Frank M. Carrano and Timothy M. Henry.
2 // Copyright (c) 2017 Pearson Education, Hoboken, New Jersey.
3 // Modified by Tiziana Ligorio for Hunter College CSCI 235
4
5 /** @file Node.cpp
6     Node for Singly Linked List*/
7
8
9 #include "Node.hpp"
10 // #include <cstddef>
11
12 // default constructor
13 template<class T>
14 Node<T>::Node() : next_(nullptr)
15 {
16 } // end default constructor
17
18
19 // parameterized constructor
20 template<class T>
21 Node<T>::Node(const T& an_item) : item_(an_item), next_(nullptr)
22 {
23 } // end constructor
24
25 // parameterized constructor
26 template<class T>
27 Node<T>::Node(const T& an_item, Node<T>* next_node_ptr) :
28     item_(an_item), next_(next_node_ptr)
29 {
30 } // end constructor
31
32
33 /** @param an_item contained in the node
34     @post sets item_ to an_item */
35 template<class T>
36 void Node<T>::setItem(const T& an_item)
37 {
38     item_ = an_item;
39 } // end setItem
40
41
42 /** @param next_node_ptr points to the next node in the chain
43     @post sets next_ to next_node_ptr */
44 template<class T>
45 void Node<T>::setNext(Node<T>* next_node_ptr)
46 {
47     next_ = next_node_ptr;
48 } // end setNext
49
```

```
50  /**@return item_*/
51  template<class T>
52  T Node<T>::getItem() const
53  {
54      return item_;
55  } // end getItem
56
57  /**@return next_*/
58  template<class T>
59  Node<T>* Node<T>::getNext() const
60  {
61      return next_;
62  } // end getNext
63
```

```
1 // Created by Frank M. Carrano and Timothy M. Henry.
2 // Copyright (c) 2017 Pearson Education, Hoboken, New Jersey.
3 // Modified by Tiziana Ligorio for Hunter College CSCI 235
4
5 /** @file Node.hpp
6     Node for Singly Linked Chain*/
7
8 #ifndef NODE_
9 #define NODE_
10
11 template<class T>
12 class Node
13 {
14 public:
15     Node(); //default constructor
16     Node(const T& an_item); //parameterized constructor
17     Node(const T& an_item, Node<T>* next_node_ptr); //parameterized constructor
18
19     /** @param an_item contained in the node
20         @post sets item_ to an_item */
21     void setItem(const T& an_item);
22
23     /** @param next_node_ptr points to the next node in the chain
24         @post sets next_ to next_node_ptr */
25     void setNext(Node<T>* next_node_ptr);
26
27     /**@return item_*/
28     T getItem() const ;
29
30     /**@return next_*/
31     Node<T>* getNext() const ;
32
33 private:
34     T item_ ; // A data item_
35     Node<T>* next_ ; // Pointer to next_ node
36 }; // end Node
37
38 #include "Node.cpp"
39 #endif
40
```

```
1  #include "Pantry.hpp"
2  #pragma once
3
4  /*
5   * Default Constructor
6   * @post: Creates a new Ingredient object with default values. String defaults to empty string.
7   * Default quantity is 0, default price is 1.
8   */
9  Ingredient::Ingredient():name_{},description_{}, quantity_{0}, price_{1}{}
10
11 /**
12  Parameterized Constructor
13  @param: A string representing a ingredient name
14  @param: A string representing ingredient description
15  @param: An int representing the ingredient's quantity
16  @param: An int representing the ingredient's price
17  @param: A vector holding Ingredient pointers representing the ingredient's recipe
18  @post: Creates a new Ingredient object with the given parameters
19  */
20 Ingredient::Ingredient(const std::string &NewName_, const std::string &NewDescription_, const int
&NewQuantity_, const int &NewPrice, const std::vector <Ingredient*> &NewRecipes_)
21 :name_{NewName_},description_{NewDescription_}, quantity_{NewQuantity_}, price_{NewPrice},
recipe_{NewRecipes_}{}
22
23 /**
24  Default Constructor
25  */
26 Pantry::Pantry(){} // Default Constructor
27
28 /**
29  @param: the name of an input file
30  @pre: Formatting of the csv file is as follows:
31      Name: A string
32      Description: A string
33      Quantity: A non negative integer
34      Price: A non negative integer
35      Recipe: A list of Ingredient titles of the form [NAME1] [NAME2];
36      For example, to make this ingredient, you need (Ingredient 1 AND Ingredient 2)
37      The value may be NONE.
38  Notes:
39      - The first line of the input file is a header and should be ignored.
40      - The recipe are separated by a semicolon and may be NONE.
41      - The recipe may be in any order.
42      - If any of the recipe are not in the list, they should be created as new ingredients with the
following information:
43          - Title: The name of the ingredient
44          - Description: "UNKNOWN"
45          - Quantity: 0
46          - Price: 0
```



```

47     - Recipe: An empty vector
48     - However, if you eventually encounter a ingredient that matches one of the "UNKNOWN"
ingredients while parsing the file, you should update all the ingredient details.
49
50     For example, given a row in the file:
51     Inferno_Espresso,An energizing elixir brewed with mystical flames providing resistance to
caffeine crashes for a limited time.,1,50,Fiery_Bean Ember_Spice
52
53     The order of the ingredients in the list:
54     Fiery_Bean, Ember_Spice, Inferno_Espresso
55     Hint: update as needed using addIngredient()
56
57     @post: Each line of the input file corresponds to a ingredient to be added to the list. No
duplicates are allowed.
58 */
59 Pantry::Pantry(const std::string &inputfilename){
60     std::ifstream input(inputfilename);
61     std::string ignoredline;
62     std::getline(input, ignoredline); // Ignore first line
63     std::string line;
64
65     while (std::getline(input, line)){
66         std::istringstream stream(line);
67         std::string Name, Description, Recipe, Temp;
68         int Quantity, Price;
69
70         std::vector<Ingredient*> Pramingredient;
71         std::getline(stream, Temp, ',');
72         Name = Temp;
73         std::getline(stream, Temp, ',');
74         Description = Temp;
75         std::getline(stream, Temp, ',');
76         std::istringstream convertquantity(Temp);
77         convertquantity >> Quantity;
78         std::getline(stream, Temp, ',');
79         std::istringstream convertprice(Temp);
80         convertprice >> Price;
81         std::getline(stream, Temp);
82         Recipe = Temp;
83
84         if(Recipe != "NONE"){//If recipe isnt NONE add all elements to the vector
85             std::vector<std::string> stringvector;
86             std::istringstream Rec(Recipe);
87             while(Rec >> Temp){
88                 Temp.erase(std::remove(Temp.begin(), Temp.end(), ','), Temp.end());
89                 stringvector.push_back(Temp);
90             }
91             for (const auto& named : stringvector){
92                 if(contains(named)){
93                     Pramingredient.push_back(getEntry(getPosOf(named)));
94                 } else {
95                     Ingredient *newIngredient = new Ingredient(Name, "UNKNOWN", 0, 1,

```

```

std::vector<Ingredient*>());
96         addIngredient(newIngredient);
97         Pramingredient.push_back(newIngredient);
98     }
99 }
100 }
101 Ingredient *ptr = new Ingredient(Name, Description, Quantity, Price, Pramingredient);
102 Pantry::addIngredient(ptr);
103 }
104 input.close();
105 }
106
107 /**
108     Destructor
109     @post: Explicitly deletes every dynamically allocated Ingredient object
110 */
111 Pantry::~Pantry(){clear();}
112
113 /**
114     @param: A const string reference to a ingredient name
115     @return: The integer position of the given ingredient if it is in the Pantry, -1 if not found.
116 */
117 int Pantry::getPosOf(const std::string&IngredientName){//if empty return -1 else interate through
the list compare each node.
118     if(isEmpty()){
119         return -1;
120     }
121     for(int i = 0; i < getLength(); i++){
122         if(getEntry(i)->name_ == IngredientName){
123             return i;
124         }
125     }
126     return -1;
127 }
128
129 /**
130     @param: A const string reference to a ingredient name
131     @return: True if the ingredient information is already in the Pantry
132 */
133 bool Pantry::contains(const std::string&IngredientName){// Calls getposOf, if it returns a value
greater than -1 then it is in the pantry
134     if(isEmpty()){
135         return false;
136     }
137     if(getPosOf(IngredientName)!= -1){
138         return true;
139     }
140     return false;
141 }
142
143
144 /**

```

```

145     @param: A pointer to an Ingredient object
146     @post: Inserts the given ingredient pointer into the Pantry, unless an ingredient of the same
name is already in the pantry.
147         Each of its Ingredients in its recipe are also added to the Pantry IF not already in the list.
148     @return: True if the ingredient was added successfully, false otherwise.
149 */
150 bool Pantry::addIngredient(Ingredient* const &object){ // Check if subrecipes are in the pantry and
add if not
151     if(object->recipe_.size() > 0){
152         for(int i = 0; i < object->recipe_.size(); i++){
153             if(!contains(object->recipe_[i]->name_)){
154                 insert(getLength(), object->recipe_[i]);
155             }
156         }
157     }
158
159     if(!contains(object->name_)){//Check if the main ingredient is not in the pantry. If not in list add it
and return true. Else return false
160         insert(getLength(), object);
161         return true;
162     }
163     else{
164         return false;
165     }
166 }
167
168 /**
169     @param: A const string reference representing a ingredient name
170     @param: A const string reference representing ingredient description
171     @param: A const int reference representing the ingredient's quantity
172     @param: A const int reference representing the ingredient's price
173     @param: A const reference to a vector holding Ingredient pointers representing the ingredient's
recipe
174     @post: Creates a new Ingredient object and inserts a pointer to it into the Pantry.
175         Each of its Ingredients in its recipe are also added to the Pantry IF not already in the list.
176     @return: True if the ingredient was added successfully
177 */
178 bool Pantry::addIngredient(const std::string& ingredient_name, const std::string&
ingredient_description, const int& ingredient_quantity, const int& ingredient_price, const
std::vector<Ingredient*> ingredient_recipe){
179     Ingredient* ptr = new Ingredient(ingredient_name, ingredient_description, ingredient_quantity,
ingredient_price,ingredient_recipe);
180     return addIngredient(ptr);
181 }//Calls AddIngredient to do the same step and return what it returns.
182
183 /**
184     @param: A Ingredient pointer
185     @return: A boolean indicating if all the given ingredient can be created (all of the ingredients in
its recipe can be created, or if you have enough of each ingredient in its recipe to create it)
186 */
187 bool Pantry::canCreate(Ingredient* const &pointer){
188     if((pointer->recipe_.size() == 0)){//If recipe_ is empty that means nothing can be created /

```

```

UNCRAFTABLE
189     return false;
190 }
191 for(int i=0; i <(pointer->recipe_).size(); i++){ // Check if subrecipe is in the list. If not return false
192     if(!contains((pointer->recipe_[i])->name_)){
193         return false;
194     };
195     if(pointer->recipe_[i]->quantity_ == 0){ // If subrecipe is in the list with 0 instance then we use
recursion and find if it's sub ingredient is in the pantry
196         if(!canCreate(pointer->recipe_[i])){
197             return false;
198         }
199     }
200 }
201 return true;
202 }
203
204 /**
205  @param: A Ingredient pointer
206  @post: Prints the ingredient name, quantity, and description.
207  The output should be of the form:
208  [Ingredient Name]: [Quantity]\n
209  [Ingredient Description]\n
210  Price: [Price]\n
211  Recipe:\n
212  [Ingredient0] [Ingredient1]\n
213
214
215  If the ingredient has no recipe, print "Recipe:\nNONE\n\n" after the price.
216 */
217 void Pantry::printIngredient(Ingredient* pointer)const{// Print the ingredient and it's information
218     if(pointer != nullptr){
219         std::cout
220             << pointer->name_ << ": "
221             << pointer->quantity_ << "\n"
222             << pointer->description_ << "\n"
223             << "Price: " << pointer->price_ << "\n";
224         if((pointer->recipe_).size() > 0){
225             std::cout << "Recipe:\n";
226             for(int i = 0; i < (pointer->recipe_).size(); i++){//Print out sub recipes
227                 std::cout << (pointer->recipe_[i])->name_ << " ";
228             }
229             std::cout << "\n";
230         }
231         if((pointer->recipe_).size() == 0){// No sub recipes
232             std::cout
233                 << "Recipe:\nNONE\n\n";
234         }
235     }
236 }
237
238 /**

```

239 @param: A const string reference to a ingredient name

240 @post: Prints a list of ingredients that must be created before the given ingredient can be
created (missing ingredients for its recipe, where you have 0 of the needed ingredient).

241 If the ingredient is already in the pantry, print "In the pantry([quantity])\n"

242 If there are no instances of the ingredient, if it cannot be crafted because of insufficient
ingredients, print "[Ingredient Name](0)\nMISSING INGREDIENTS"

243 If it can be crafted, recursively print the ingredients that need to be used (if there are
instances of them) or created (if there are no instances of them) in the order that the ingredients
appear in the recipe, joined by "<-".

244 If the ingredient has no recipe, print "UNCRAFTABLE\n" at the end of the function.

245

246 Below are some of the expected forms. "Scenario: [scenario] is not part of the output. It is
just to help you understand the expected output.":

247

248 Scenario: The Ingredient does not exist in the list

249 Query: [Ingredient Name]

250 No such ingredient

251

252 Scenario: The Ingredient exists in the list, and there are >0 instances of it

253 Query: [Ingredient Name]

254 In the pantry ([Quantity])

255

256 Scenario: The Ingredient exists in the list, and there are 0 instances of it, and it is craftable
by using an ingredient that is already in the pantry

257 Query: [Ingredient Name0]

258 [Ingredient Name0](C)

259 [Ingredient Name1](1)

260

261 Scenario: The Ingredient exists in the list, and there are 0 instances of it, and it is craftable
by using an ingredient that has to be crafted

262 Query: [Ingredient Name0]

263 [Ingredient Name0](C)

264 [Ingredient Name1](C) <- [Ingredient Name2](3)

265

266 Scenario: The Ingredient exists in the list, and there are 0 instances of it, and there are
multiple ingredients that have to be crafted (each row represents a different ingredient inã the
original recipe)

267 Query: [Ingredient Name0]

268 [Ingredient Name0](C)

269 [Ingredient Name1](C) <- [Ingredient Name2](3)

270 [Ingredient Name3](C) <- [Ingredient Name4](C) <- [Ingredient Name5] (3)

271

272 Scenario: The Ingredient exists in the list, and there are 0 instances of it, and it is not
craftable (it has no recipe)

273 Query: [Ingredient Name0]

274 UNCRAFTABLE

275

276 Scenario: The Ingredient exists in the list, and there are 0 instances of it, and it has a recipe,
but you do not have enough of the ingredients to craft it

277 Query: [Ingredient Name0]

278 [Ingredient Name0](0)

279 MISSING INGREDIENTS

```

280
281 */
282 void Pantry::ingredientQuery(const std::string &ingredient_name) {
283     std::cout << "Query: " << ingredient_name << "\n";
284     if (!contains(ingredient_name)) { // Not in pantry
285         std::cout << "No such ingredient\n";
286     }
287     else { // In pantry with greater than 0 instance
288         Ingredient* Main = getEntry(getPosOf(ingredient_name));
289         if (Main->quantity_ > 0) {
290             std::cout << "In the pantry (" << Main->quantity_ << ")" << "\n";
291             if (!canCreate(Main)) { // Cannot be crafted
292                 std::cout << "UNCRAFTABLE\n";
293             }
294         }
295         else if (!canCreate(Main)) {
296             std::cout << "UNCRAFTABLE\n";
297         }
298         else { // 0 instance but craftable
299             if (canCreate(Main)) {
300                 std::cout << Main->name_ << "(C)\n";
301                 Helperfunction(Main);
302                 std::cout << "\n";
303             } else { // missing subrecipes n pantry
304                 std::cout << Main->name_ << "(0)" << "\n" << "MISSING INGREDIENTS" << "\n";
305             }
306         }
307     }
308 }
309
310 /**
311  @return: An integer sum of the price of all the ingredients currently in the list.
312  Note: This should only include price values from ingredients that you have 1 or more of. Do not
313  consider ingredients that you have 0 of, even if you have the ingredients to make them.
314 */
315 int Pantry::calculatePantryValue() const {
316     int num = 0;
317     for(int i = 0; i < getLength(); i++) { // Iterate through the pantry and adding it to sum with price *
318         // quantity
319         num = num + (getEntry(i)->price_ * getEntry(i)->quantity_);
320     }
321     return num;
322 }
323
324 /**
325  @param: A const string reference to a filter with a default value of "NONE".
326  @post: With default filter "NONE": Print out every ingredient in the list.
327  With filter "CONTAINS": Only print out the ingredients with >0 instances in the list.
328  With filter "MISSING": Only print out the ingredients with 0 instances in the list.
329  With filter "CRAFTABLE": Only print out the ingredients where you have all the ingredients to
330  craft them.
331  If an invalid filter is passed, print "INVALID FILTER\n"

```

```

329     Printing ingredients should be of the form:
330
331     [Ingredient name]: [Quantity]
332     [Ingredient description]\n
333     Price: [price]\n
334     Recipe:\n
335     [Ingredient0] [Ingredient1]\n\n
336
337     If the ingredient has no recipe, print "Recipe:\nNONE\n\n" after the price.
338 */
339 void Pantry::pantryList(const std::string &list){// Calls Print Ingredient if ingredeient matches the
requirement
340     if(list == "NONE"){//Filters for NONE
341         for(int i = 0; i < item_count_; i++){
342             Ingredient* temp = getEntry(i);
343             printIngredient(temp);
344         }
345     }
346     else if(list == "CONTAINS"){//Filters for CONTAIN
347         for(int i = 0; i < item_count_; i++){
348             Ingredient* temp = getEntry(i);
349             if(temp->quantity_ > 0){
350                 printIngredient(temp);
351             }
352         }
353     }
354     else if(list == "MISSING"){//Filter for MISSING
355         for(int i = 0; i < item_count_; i++){
356             Ingredient* temp = getEntry(i);
357             if(temp->quantity_ == 0){
358                 printIngredient(temp);
359             }
360         }
361     }
362     else if(list == "CRAFTABLE"){//Filter for Craftable
363         for(int i = 0; i < item_count_; i++){
364             Ingredient* temp = getEntry(i);
365             if(canCreate(temp)){
366                 printIngredient(temp);
367             }
368         }
369     }
370     else{//Invaild Filter
371         std::cout << "INVALID FILTER\n";
372     }
373 }
374
375 /**
376  * @param: A Ingredient pointer
377  * @post: Helper function for ingredientQuery()
378  * Prints the name of sub ingredient for Ingredient pointer.
379  * If the quantity of sub ingredient is 0 then print all neccesary subingredient require to craft it.

```

Using recursion.

```
380 */
381 void Pantry::Helperfunction(Ingredient* Main){
382     for (int i = 0; i < Main->recipe_.size(); i ++){
383         Ingredient* Side = Main->recipe_[i];
384         std::cout << Side->name_;
385         if (canCreate(Side) && Side->quantity_ == 0) { //Check if Sub Ingredient Quantity is 0. If zero
            declare it as craftable and print out all sub subingredient required for the sub ingredeint
386             std::cout << "(C) <- ";
387             Helperfunction(Side);
388         }
389         else if (Side->quantity_ > 0) {
390             std::cout << "(" << Side->quantity_ << ")" << "\n";
391         }
392     }
393 }
```



```
1  #include <string>
2  #include <vector>
3  #include <fstream>
4  #include <iostream>
5  #include "LinkedList.hpp"
6  #include <sstream>
7  #include <algorithm>
8  #pragma once
9  struct Ingredient{
10     std::string name_;
11     std::string description_;
12     int quantity_;
13     int price_;
14     std::vector <Ingredient*> recipe_;
15     /**
16      Default Constructor
17      @post: Creates a new Ingredient object with default values. String defaults to empty string.
18           Default quantity is 0, default price is 1.
19     */
20     Ingredient();// Default
21
22     /**
23      Parameterized Constructor
24      @param: A string representing a ingredient name
25      @param: A string representing ingredient description
26      @param: An int representing the ingredient's quantity
27      @param: An int representing the ingredient's price
28      @param: A vector holding Ingredient pointers representing the ingredient's recipe
29      @post: Creates a new Ingredient object with the given parameters
30     */
31     Ingredient(const std::string &NewName_, const std::string &NewDescription_, const int
&NewQuantity_,
32               const int &NewPrice, const std::vector <Ingredient*> &NewRecipes_); //Parameterized
33 };
34
35 class Pantry: public LinkedList<Ingredient*>{
36     public:
37     /**
38      Default Constructor
39     */
40     Pantry(); // Default Constructor
41
42     /**
43      @param: the name of an input file
44      @pre: Formatting of the csv file is as follows:
45           Name: A string
46           Description: A string
47           Quantity: A non negative integer
48           Price: A non negative integer
```

```

49     Recipe: A list of Ingredient titles of the form [NAME1] [NAME2];
50     For example, to make this ingredient, you need (Ingredient 1 AND Ingredient 2)
51     The value may be NONE.
52     Notes:
53     - The first line of the input file is a header and should be ignored.
54     - The recipe are separated by a semicolon and may be NONE.
55     - The recipe may be in any order.
56     - If any of the recipe are not in the list, they should be created as new ingredients with the
following information:
57     - Title: The name of the ingredient
58     - Description: "UNKNOWN"
59     - Quantity: 0
60     - Price: 0
61     - Recipe: An empty vector
62     - However, if you eventually encounter a ingredient that matches one of the "UNKNOWN"
ingredients while parsing the file, you should update all the ingredient details.
63
64     For example, given a row in the file:
65     Inferno_Espresso,An energizing elixir brewed with mystical flames providing resistance to
caffeine crashes for a limited time.,1,50,Fiery_Bean Ember_Spice
66
67     The order of the ingredients in the list:
68     Fiery_Bean, Ember_Spice, Inferno_Espresso
69     Hint: update as needed using addIngredient()
70
71     @post: Each line of the input file corresponds to a ingredient to be added to the list. No
duplicates are allowed.
72 */
73     Pantry(const std::string &inputfilename); // Parameterize constructor
74
75 /**
76     Destructor
77     @post: Explicitly deletes every dynamically allocated Ingredient object
78 */
79     ~Pantry();
80
81 /**
82     @param: A const string reference to a ingredient name
83     @return: The integer position of the given ingredient if it is in the Pantry, -1 if not found.
REMEMBER, indexing starts at 0.
84 */
85     int getPosOf(const std::string&IngredientName);
86
87 /**
88     @param: A const string reference to a ingredient name
89     @return: True if the ingredient information is already in the Pantry
90 */
91     bool contains(const std::string&IngredientName);
92
93 /**
94     @param: A pointer to an Ingredient object
95     @post: Inserts the given ingredient pointer into the Pantry, unless an ingredient of the same

```

name is already in the pantry.

Each of its Ingredients in its recipe are also added to the Pantry IF not already in the list.

@return: True if the ingredient was added successfully, false otherwise.

*/

```
bool addIngredient(Ingredient* const &object);
```

/**

@param: A const string reference representing a ingredient name

@param: A const string reference representing ingredient description

@param: A const int reference representing the ingredient's quantity

@param: A const int reference representing the ingredient's price

@param: A const reference to a vector holding Ingredient pointers representing the ingredient's recipe

@post: Creates a new Ingredient object and inserts a pointer to it into the Pantry.

Each of its Ingredients in its recipe are also added to the Pantry IF not already in the list.

@return: True if the ingredient was added successfully

*/

```
bool addIngredient(const std::string& ingredient_name, const std::string&
ingredient_description, const int& ingredient_quantity
, const int& ingredient_price, const std::vector<Ingredient*> ingredient_recipe);
```

/**

@param: A Ingredient pointer

@return: A boolean indicating if all the given ingredient can be created (all of the ingredients in its recipe can be created, or if you have enough of each ingredient in its recipe to create it)

*/

```
bool canCreate(Ingredient* const &pointer);
```

/**

@param: A Ingredient pointer

@post: Prints the ingredient name, quantity, and description.

The output should be of the form:

[Ingredient Name]: [Quantity]\n

[Ingredient Description]\n

Price: [Price]\n

Recipe:\n

[Ingredient0] [Ingredient1]\n

If the ingredient has no recipe, print "Recipe:\nNONE\n\n" after the price.

*/

```
void printIngredient(Ingredient* pointer)const;
```

/**

@param: A const string reference to a ingredient name

@post: Prints a list of ingredients that must be created before the given ingredient can be created (missing ingredients for its recipe, where you have 0 of the needed ingredient).

If the ingredient is already in the pantry, print "In the pantry([quantity])\n"

If there are no instances of the ingredient, if it cannot be crafted because of insufficient ingredients, print "[Ingredient Name](0)\nMISSING INGREDIENTS"

If it can be crafted, recursively print the ingredients that need to be used (if there are instances of them) or created (if there are no instances of them) in the order that the ingredients

```

appear in the recipe, joined by "<-.
141     If the ingredient has no recipe, print "UNCRAFTABLE\n" at the end of the function.
142
143     Below are some of the expected forms. "Scenario: [scenario] is not part of the output. It is
just to help you understand the expected output.":
144
145     Scenario: The Ingredient does not exist in the list
146     Query: [Ingredient Name]
147     No such ingredient
148
149     Scenario: The Ingredient exists in the list, and there are >0 instances of it
150     Query: [Ingredient Name]
151     In the pantry ([Quantity])
152
153     Scenario: The Ingredient exists in the list, and there are 0 instances of it, and it is craftable
by using an ingredient that is already in the pantry
154     Query: [Ingredient Name0]
155     [Ingredient Name0](C)
156     [Ingredient Name1](1)
157
158     Scenario: The Ingredient exists in the list, and there are 0 instances of it, and it is craftable
by using an ingredient that has to be crafted
159     Query: [Ingredient Name0]
160     [Ingredient Name0](C)
161     [Ingredient Name1](C) <- [Ingredient Name2](3)
162
163     Scenario: The Ingredient exists in the list, and there are 0 instances of it, and there are
multiple ingredients that have to be crafted (each row represents a different ingredient inå the
original recipe)
164     Query: [Ingredient Name0]
165     [Ingredient Name0](C)
166     [Ingredient Name1](C) <- [Ingredient Name2](3)
167     [Ingredient Name3](C) <- [Ingredient Name4](C) <- [Ingredient Name5] (3)
168
169     Scenario: The Ingredient exists in the list, and there are 0 instances of it, and it is not
craftable (it has no recipe)
170     Query: [Ingredient Name0]
171     UNCRAFTABLE
172
173     Scenario: The Ingredient exists in the list, and there are 0 instances of it, and it has a recipe,
but you do not have enough of the ingredients to craft it
174     Query: [Ingredient Name0]
175     [Ingredient Name0](0)
176     MISSING INGREDIENTS
177
178 */
179 void ingredientQuery(const std::string &ingredient_name);//Undoable GG
180
181 /**
182     @return: An integer sum of the price of all the ingredients currently in the list.
183     Note: This should only include price values from ingredients that you have 1 or more of. Do not
consider ingredients that you have 0 of, even if you have the ingredients to make them.

```

```

184 */
185     int calculatePantryValue()const;
186
187 /**
188     @param: A const string reference to a filter with a default value of "NONE".
189     @post: With default filter "NONE": Print out every ingredient in the list.
190           With filter "CONTAINS": Only print out the ingredients with >0 instances in the list.
191           With filter "MISSING": Only print out the ingredients with 0 instances in the list.
192           With filter "CRAFTABLE": Only print out the ingredients where you have all the ingredients to
craft them.
193           If an invalid filter is passed, print "INVALID FILTER\n"
194           Printing ingredients should be of the form:
195
196           [Ingredient name]: [Quantity]
197           [Ingredient description]\n
198           Price: [price]\n
199           Recipe:\n
200           [Ingredient0] [Ingredient1]\n\n
201
202           If the ingredient has no recipe, print "Recipe:\nNONE\n\n" after the price.
203 */
204     void pantryList(const std::string &list = "NONE");
205
206 /**
207     * @param: A Ingredient pointer
208     * @post: Helper function for ingredientQuery()
209     * Prints the name of sub ingredient for Ingredient pointer.
210     * If the quantity of sub ingredient is 0 then print all necessary subingredient require to craft it.
Using recursion.
211 */
212     void Helperfunction(Ingredient* Main);
213 };
214

```

▼ PrecondViolatedExcep.cpp

 Download

```

1 // Created by Frank M. Carrano and Timothy M. Henry.
2 // Copyright (c) 2017 Pearson Education, Hoboken, New Jersey.
3
4 /** Listing 7-6.
5     @file PrecondViolatedExcep.cpp */
6 #include "PrecondViolatedExcep.hpp"
7
8 PrecondViolatedExcep::PrecondViolatedExcep(const std::string& message)
9     : std::logic_error("Precondition Violated Exception: " + message)
10 {
11 } // end constructor
12
13 // End of implementation file.
14
15

```

▼ PrecondViolatedExcep.hpp

Download

1

// Created by Frank M. Carrano and Timothy M. Henry.

2

// Copyright (c) 2017 Pearson Education, Hoboken, New Jersey.

3

4

/** Listing 7-5.

5

@file PrecondViolatedExcep.hpp */

6

7

#ifndef PRECOND_VIOLATED_EXCEP_

8

#define PRECOND_VIOLATED_EXCEP_

9

10

#include <stdexcept>

11

#include <string>

12

13

class PrecondViolatedExcep : public std::logic_error

14

{

15

public:

16

PrecondViolatedExcep(const std::string& message = "");

17

}; // end PrecondViolatedExcep

18

#endif

19

▼ PrecondViolatedExcep.o

Download

1

Large file hidden. You can download it using the button above.

▼ README.md

Download

1

[![Review Assignment Due Date](https://classroom.github.com/assets/deadline-readme-button-24ddc0f5d75046c5622901739e7c5dd533143b0c8e959d652212380cedb1ea36.svg)]

2

(https://classroom.github.com/a/8RoCzZwm)

3

Project5

4

The project specification can be found on Blackboard

5

▼ debug.csv

Download

1

Name,Description,Quantity,Price,Recipe

2

in1,des1,3,72,NONE

3

in2,des2,0,138,in1;

4

in3,des3,0,103,in2;

5

in4,des4,0,185,in2 in3;

6

in5,des5,0,185,in4;

7

in6,des6,0,97,in1 in5;

8

in7,des7,0,56,NONE

9

in8,des8,0,87,in2 in7;

```

1 Name,Description,Quantity,Price,Recipe
2 Inferno_Espresso,An energizing elixir brewed with mystical flames providing resistance to caffeine
  crashes for a limited time.,1,50,Fiery_Bean Ember_Spice;
3 Spectral_Handcuffs,Enchanted cuffs imbued with mischievous spirits allowing the user to
  temporarily restrain prankster creatures in combat.,2,40,Whispering_Essence Enchanted_Chains;
4 Psychic_Kaleidoscope,A mystical kaleidoscope that enhances psychic abilities revealing the user's
  inner rainbow of emotions.,1,60,Crystalized_Mindstone Psychedelic_Essence;
5 Mystical_Smoothie,A magical smoothie that enhances vitality and focus blended with rare herbs
  and a dash of pixie dust.,3,30,Mystical_Mushroom Arcane_Smoothie Shadowroot;
6 Inferno_Disco_Suit,Highly resistant disco attire set ablaze with groovy flames offering exceptional
  protection against dance-induced fire hazards.,1,80,Fiery_Suit Ember_Sequins
  Enchanted_Iron_Ingot;
7 Soul_Whisk,A mysterious whisk crafted to whisk away otherworldly calories dealing extra damage to
  calories and unwanted muffin tops.,1,45,Ectoplasmic_Essence Whisk_of_Whimsy Silver_Twine;
8 Psionic_Party_Hat,A festive hat infused with psychic energy granting a bonus to intelligence and a
  flair for telepathic dance coordination.,1,55,Crystalized_Mindstone Psionic_Silk Astral_Confetti;
9 Mystical_Snack_Trap,A special trap designed to attract and capture elusive creatures for crafting
  purposes using enchanted snacks as bait.,5,20,Glowing_Nibbles Enchanted_Gummies
  Enigmatic_Bait;
10 Inferno_Hot_Sauce,A fiery potion that temporarily grants the user the ability to unleash spicy breath
  attacks perfect for adding heat to any dish.,1,70,Fiery_Pepper Ember_Essence Essence_of_Inferno;
11 Ethereal_Bubble_Wrap,A shield designed to repel spectral pranks offering additional protection
  against ethereal whoopee cushions and ectoplasmic goo.,1,50,Ethereal_Essence
  Enchanted_Bubble_Wrap Blessed_Laughter;
12 Astral_Kazoo,A magical kazoo that enhances psychic abilities allowing the user to unleash
  devastating ethereal tunes causing enemies to dance uncontrollably.,1,65,Crystalized_Mindstone
  Arcane_Essence Eldritch_Reed;
13 Shadow_Slippery_Salve,An aromatic salve infused with mystical shadows providing a temporary
  boost to stealth and agility plus a dash of slippery humor.,2,25,Mystical_Mushroom Shadowroot
  Echoblossom;
14 Fiery_Bean,A magical bean grown in the heart of an enchanted volcano imbued with the essence of
  mystical flames.,1,10,NONE
15 Ember_Spice,A rare spice harvested from Ember Orchards known for adding a touch of magical
  heat to any dish.,1,15,NONE
16 Whispering_Essence,A mischievous essence captured from playful spirits often found hiding in
  ancient ruins or haunted forests.,1,25,NONE
17 Mystical_Mushroom,A mystical mushroom found in abundance in dark caverns.,1,15,NONE
18 MycoMorsel,A mystical mushroom dish known for its unique properties. When consumed it boosts
  health and provides a temporary burst of energy...allegedly.,0,20,Mystical_Mushroom
  Enchanted_Sugar Glowing_Moss;
19 Enchanted_Sugar,A sweet and magical sugar infused with mystical energy perfect for enhancing
  the flavor and enchantment of various recipes.,5,5,NONE
20 Glowing_Moss,Mystical moss that emits a soft glow harvested from enchanted forests. Used to add
  a magical radiance to dishes and concoctions.,3,8,NONE
21

```