

Lab 9

Name: Haichuan Wei

Lab: Lab 9

Date: 11/21/2021

Program Description:

This program was made to practice templates, generic class object, and linked lists. The program takes in data and uses templates to create and lists of data. The program will then print the data in the list.

Source Code:

LinkedList.h

```
h LinkedList.h > ListNode<T>
24 template <typename T>
25 class ListNode
26 {
27
28 public:
29     friend class List<T>; // make List a friend
30     /**
31      * the constructor of ListNode that set data to info, set nextPtr to null.
32      * @param the address of info
33      */
34     ListNode(const T &); // constructor
35     /**
36      * the member function of ListNode that return the value of data
37      * @return data
38      */
39     T getData() const; // return the data in the node
40
41     // set nextPtr to nPtr
42     /**
43      * The member function of ListNode that set nextPtr to nPtr
44      * @param nPtr
45      */
46     void setNextPtr(ListNode *nPtr);
47
48     // return nextPtr
49     /**
50      * The member function of ListNode that return the nextPtr
51      * @return nextPtr
52      */
53     ListNode *getNextPtr() const;
54
55 private:
56     T data; // data
57     int key; // used for key for the list
58     ListNode *nextPtr; // next node in the list
59 }; // end class ListNode
60
61 template <typename T>
62 class List
63 {
64 public:
65     /**
66      * The default constructor that set firstPtr and lastPtr to 0.
67      */
68     List(); // default constructor
69     /**
70      * The copy constructor that copy a list to another
71      * @param copy
72      */
```

Lab 9

```

h LinkedList.h > ListNode<T>
71      @param copy
72      */
73      List(const List<T> &); // copy constructor
74  ✓      /**
75      * The destructor of List that delete data and nodes
76      */
77      ~List(); // destructor
78  ✓      /**
79      * The member function that insert a node at the front of the list
80      * @param value, key
81      */
82      void insertAtFront(const T &, int);
83  ✓      /**
84      * The member function of List that insert at the back of the list
85      * @param value, key
86      */
87      void insertAtBack(const T &, int);
88  ✓      /**
89      * The member function that remove one from front
90      * @param value
91      */
92      bool removeFromFront(T &);
93  ✓      /**
94      * The member function that remove one from back
95      * @param value
96      */
97      bool removeFromBack(T &);
98  ✓      /**
99      * The member function that check whether the first ptr is null
100     * @return true if the firstPtr is null, false if it isn't
101     */
102     bool isEmpty() const;
103  ✓     /**
104     * The member function of List that print the contents of the List
105     */
106     void print() const;
107     void printPtrFunc();
108  ✓     /**
109     * The member function of List that display the contents of the List
110     * @param myKey
111     * @return NULL until can't find anything else
112     */
113     T *getInfo(int myKey);
114     // return nextPtr
115  ✓     /**
116     * The member function of List that return the firstPtr
117     * @return firstPtr
118     */
119     ListNode<T> *getFirstPtr() const; // end function getNextPtr
120

```

Lab 9

LinkedList_Test.cpp

```
h LinkedList> ...
24 template <typename T>
25 class ListNode
26 {
27
28 public:
29     friend class List<T>; // make List a friend
30
31     /**
32      * the constructor of ListNode that set data to info, set nextPtr to null.
33      * @param the address of info
34      */
35     ListNode(const T &); // constructor
36
37     /**
38      * the member function of ListNode that return the value of data
39      * @return data
40      */
41     T getData() const; // return the data in the node
42
43     // set nextPtr to nPtr
44     /**
45      * The member function of ListNode that set nextPtr to nPtr
46      * @param nPtr
47      */
48     void setNextPtr(ListNode *nPtr);
49
50     // return nextPtr
51     /**
52      * The member function of ListNode that return the nextPtr
53      * @return nextPtr
54      */
55     ListNode *getNextPtr() const;
56
57 private:
58     T data; // data
59     int key; // used for key for the list
60     ListNode *nextPtr; // next node in the list
61 }; // end class ListNode
62
63 template <typename T>
64 class List
65 {
66 public:
67     /**
68      * The default constructor that set firstPtr and lastPtr to 0.
69      */
70     List(); // default constructor
71
72     /**
73      * The copy constructor that copy a list to another
74      * @param copy
75      */
76     List(const List<T> &); // copy constructor
77
78     /**
79      * The destructor of List that delete data and nodes
80      */
81     ~List(); // destructor
82
83     /**
84      * The member function that insert a node at the front of the list
85      * @param value, key
86      */
87     void insertAtFront(const T &, int);
88
89     /**
90      * The member function of List that insert at the back of the list
91      * @param value, key
92      */
93     void insertAtBack(const T &, int);
94
95     /**
96      * The member function that remove one from front
97      * @param value
98      */
99     bool removeFromFront(T &);
100
101     /**
102      * The member function that remove one from back
103      * @param value
104      */
105     bool removeFromBack(T &);
106
107     /**
108      * The member function that check whether the first ptr is null
109      * @return true if the firstPtr is null, false if it isn't
110      */
111     bool isEmpty() const;
112
113     /**
114      * The member function of List that print the contents of the List
115      */
116     void print() const;
117     void printPtrFunc();
118
119     /**
120      * The member function of List that display the contents of the List
121      * @param myKey
122      * @return NULL until can't find anything else
123      */
124     T *getInfo(int myKey);
125
126     // return nextPtr
127
128     /**
129      * The member function of List that return the firstPtr
130      * @return firstPtr
131      */
132     ListNode<T> *getFirstPtr() const; // end function getNextPtr
```

Lab 9

```
119     ListNode<T> *getFirstPtr() const; // end function getFirstPtr
120
121 protected:
122     ListNode<T> *firstPtr; // pointer to first node
123     ListNode<T> *lastPtr; // pointer to last node
124
125     // Utility function to allocate a new node
126     /**
127      * The member function that return the new node
128      * @param value and an integer
129      * @return ptr
130      */
131     ListNode<T> *getNode(const T &, int);
132 }; // end typename Template List
133
134 // constructor
135 /**
136  * the constructor of ListNode that set data to info, set nextPtr to null.
137  * @param the address of info
138  */
139 template <typename T>
140 ListNode<T>::ListNode(const T &info)
141 {
142     data = info;
143     nextPtr = 0;
144 } // end constructor
145
146 // return a copy of the data in the node
147 /**
148  * the member function of ListNode that return the value of data
149  * @return data
150  */
151 template <typename T>
152 T ListNode<T>::getData() const
153 {
154     return data;
155 } // end function getData
156
157 // set nextPtr to nPtr
158 /**
159  * The member function of ListNode that set nextPtr to nPtr
160  * @param nPtr
161  */
162 template <typename T>
163 void ListNode<T>::setNextPtr(ListNode *nPtr)
164 {
165     nextPtr = nPtr;
166 } // end function setNextPtr
167
```

```
167
168 // return nextPtr
169 /**
170  * The member function of ListNode that return the nextPtr
171  * @return nextPtr
172  */
173 template <typename T>
174 ListNode<T> *ListNode<T>::getNextPtr() const
175 {
176     return nextPtr;
177 } // end function getNextPtr
178
179 // default constructor
180 /**
181  * The default constructor that set firstPtr and lastPtr to 0.
182  */
183 template <typename T>
184 List<T>::List()
185 {
186     firstPtr = lastPtr = 0;
187 } // end constructor
188
189 // copy constructor
190 /**
191  * The copy constructor of List that copy a list to another
192  * @param copy
193  */
194 template <typename T>
195 List<T>::List(const List<T> &copy)
196 {
197     firstPtr = lastPtr = 0; // initialize pointers
198
199     ListNode<T> *currentPtr = copy.firstPtr;
200
201     // insert into the list
202     while (currentPtr != 0)
203     {
204         insertAtBack(currentPtr->data);
205         currentPtr = currentPtr->nextPtr;
206     } // end while
207 } // end List copy constructor
208
209 // destructor
210 /**
211  * The destructor of List that delete data and nodes
212  */
213 template <typename T>
214 List<T>::~List()
```

Lab 9

```
214 ~List<T>::~List()
215 {
216     if (!isEmpty()) // List is not empty
217     {
218         // cout << "Destroying nodes ...\n";
219
220         ListNode<T> *currentPtr = firstPtr;
221         ListNode<T> *tempPtr;
222
223         while (currentPtr != 0) // delete remaining nodes
224         {
225             tempPtr = currentPtr;
226             // cout << tempPtr->data << ' ';
227             currentPtr = currentPtr->nextPtr;
228             delete tempPtr;
229         } // end while
230     } // end if
231
232     // cout << "\nAll nodes destroyed\n\n";
233 } // end destructor
234
235 // Insert a node at the front of the list
236 /**
237  * The member function that insert a node at the front of the list
238  * @param value, key
239  */
240 template <typename T>
241 void List<T>::insertAtFront(const T &value,
242                             int key)
243 {
244     ListNode<T> *newPtr = getNewNode(value, key);
245
246     if (isEmpty()) // List is empty
247         firstPtr = lastPtr = newPtr;
248     else // List is not empty
249     {
250         /*
251          * write code to implement insert at front
252          * 1. the new node needs to point to the first node
253          * 2. first node needs to point to the new node
254          */
255         newPtr->nextPtr = firstPtr;
256         firstPtr = newPtr;
257     } // end else
258 } // end function insertAtFront
259
260 // Insert a node at the back of the list
261 /**
262  * The member function of List that insert at the back of the list
263  * @param value, key
264  */
265 template <typename T>
266 void List<T>::insertAtBack(const T &value, int key)
267 {
268     ListNode<T> *newPtr = getNewNode(value, key);
269
270     if (isEmpty()) // List is empty
271         firstPtr = lastPtr = newPtr;
272     else // List is not empty
273     {
274         /*
275          * write code to implement insert at back
276          * 1. next pointer of the last node points to the new node
277          * 2. last node needs to point to the new node
278          */
279         lastPtr->nextPtr = newPtr;
280         lastPtr = newPtr;
281     } // end else
282 } // end function insertAtBack
283
284 // Delete a node from the front of the list
285 /**
286  * The member function that remove one from front
287  * @param value
288  */
289 template <typename T>
290 bool List<T>::removeFromFront(T &value)
291 {
292     if (isEmpty()) // List is empty
293         return false; // delete unsuccessful
294     else
295     {
296         ListNode<T> *tempPtr = firstPtr;
297
298         /*
299          * 1. check to see if first pointer is the same as last pointer
300          * 2. if it is the same, that means there is only one node, so
301          *    set both pointer to NULL
302          * 3. if it is not the same, that means there is more than one node
303          *    in the linked list. So, make the the first node pointer points
304          *    to the the next node of the first node
305          * 4. don't forget delete the pointer since it has been removed/deleted
306          */
307     }
308 }
```

Lab 9

```

309 * don't forget delete the pointer since it has been removed/deleted
310 *
311 * Write your code to implement the remove the 1st node
312 *
313 */
314 if (firstPtr == lastPtr)
315 {
316     firstPtr = lastPtr = NULL;
317 }
318 else
319 {
320     firstPtr = firstPtr->nextPtr;
321 }
322 value = tempPtr->data;
323 delete tempPtr;
324 return true; // delete successful
325 } // end else
326 } // end function removeFromFront
327
328 // delete a node from the back of the list
329 template <typename T>
330 /**
331  * The member function that remove one from back
332  * @param value
333  */
334 bool List<T>::removeFromBack(T &value)
335 {
336     if (isEmpty())
337         return false; // delete unsuccessful
338     else
339     {
340         ListNode<T> *tempPtr = lastPtr;
341
342         if (firstPtr == lastPtr)
343             firstPtr = lastPtr = 0;
344         else
345         {
346             ListNode<T> *currentPtr = firstPtr;
347
348             while (currentPtr->nextPtr != lastPtr)
349                 currentPtr = currentPtr->nextPtr;
350
351             lastPtr = currentPtr;
352             currentPtr->nextPtr = 0;
353         } // end else
354
355         value = tempPtr->data;
356         delete tempPtr;
357     }
358     // end else
359 } // end function removeFromBack
360
361 // Is the List empty?
362 /**
363  * The member function that check whether the first ptr is null
364  * @return true if the firstPtr is null, false if it isn't
365  */
366 template <typename T>
367 bool List<T>::isEmpty() const
368 {
369     return firstPtr == 0;
370 } // end function isEmpty
371
372 // Return a pointer to a newly allocated node
373 /**
374  * The member function that return the new node
375  * @param value and an integer
376  * @return ptr
377  */
378 template <typename T>
379 ListNode<T> *List<T>::getNewNode(
380     const T &value, int)
381 {
382     ListNode<T> *ptr = new ListNode<T>(value);
383     return ptr;
384 } // end function getNewNode
385
386 // Display the contents of the List
387 /**
388  * The member function of List that print the contents of the List
389  */
390 template <typename T>
391 void List<T>::print() const
392 {
393     if (isEmpty()) // empty list
394     {
395         cout << "The list is empty\n\n";
396         return;
397     } // end if
398
399     ListNode<T> *currentPtr = firstPtr;
400
401     // cout << "The list is: ";
402
403     while (currentPtr != 0) // display elements in list
404     {

```

```

357     return true; // delete successful
358     }
359 } // end function removeFromBack
360
361 // Is the List empty?
362 /**
363  * The member function that check whether the first ptr is null
364  * @return true if the firstPtr is null, false if it isn't
365  */
366 template <typename T>
367 bool List<T>::isEmpty() const
368 {
369     return firstPtr == 0;
370 } // end function isEmpty
371
372 // Return a pointer to a newly allocated node
373 /**
374  * The member function that return the new node
375  * @param value and an integer
376  * @return ptr
377  */
378 template <typename T>
379 ListNode<T> *List<T>::getNewNode(
380     const T &value, int)
381 {
382     ListNode<T> *ptr = new ListNode<T>(value);
383     return ptr;
384 } // end function getNewNode
385
386 // Display the contents of the List
387 /**
388  * The member function of List that print the contents of the List
389  */
390 template <typename T>
391 void List<T>::print() const
392 {
393     if (isEmpty()) // empty list
394     {
395         cout << "The list is empty\n\n";
396         return;
397     } // end if
398
399     ListNode<T> *currentPtr = firstPtr;
400
401     // cout << "The list is: ";
402
403     while (currentPtr != 0) // display elements in list
404     {

```

Lab 9

```

403 while (currentPtr != 0) // display elements in list
404 {
405     int i;
406     string s;
407     double d;
408     char c;
409     if (typeid(currentPtr->data).name() == typeid(i).name() ||
410         typeid(currentPtr->data).name() == typeid(d).name() ||
411         typeid(currentPtr->data).name() == typeid(s).name() ||
412         typeid(currentPtr->data).name() == typeid(c).name())
413     {
414         // data value is a simple data type and can be printed
415         cout << currentPtr->data << ' ';
416     }
417     else
418     {
419         cout << "Can't print - Not a simple data type (int, string, char, double)\n";
420     }
421     currentPtr = currentPtr->nextPtr;
422 } // end while
423
424 cout << "\n\n";
425 } // end function print
426
427 /**
428  * The member function of List that return the firstPtr
429  * @return firstPtr
430  */
431 template <typename T>
432 ListNode<T> *List<T>::getFirstPtr() const
433 {
434     return firstPtr;
435 } // end function getNextPtr
436
437 // Display the contents of the List
438 /**
439  * The member function of List that display the contents of the List
440  * @param myKey
441  * @return NULL until can't find anything else
442  */
443 template <typename T>
444 T *List<T>::getInfo(int myKey)
445 {
446     if (isEmpty()) // empty list
447     {
448         cout << "The list is empty\n\n";
449         return NULL;
450     } // end if
451

```

```

448     cout << "The list is empty\n\n";
449     return NULL;
450 } // end if
451
452 ListNode<T> *currentPtr = firstPtr;
453
454 while (currentPtr != 0) // display elements in list
455 {
456     if (currentPtr->key == myKey) // found
457         return (&currentPtr->data);
458     currentPtr = currentPtr->nextPtr;
459 } // end while
460
461 return NULL; // can't find
462 } // end function print
463
464 /**
465  * The function that can hold any type of node list and print the information it has
466  * @param nodeList
467  */
468 template <typename T>
469 void printNoteInfo(List<T> &nodeList)
470 {
471     T *wp;
472     wp = (T *)nodeList.getInfo(0); // get node based on key
473
474     ListNode<T> *currentPtr;
475     currentPtr = nodeList.getFirstPtr();
476
477     cout << "The node list is: \n";
478     // print out all the info in linked list
479     while (currentPtr != 0) // display elements in list
480     {
481         wp = (T *)currentPtr; // convert to correct data type
482         wp->printInfo(); // calling the function that is part of the
483         // data type
484         currentPtr = currentPtr->getNextPtr();
485     } // end while
486 }
487 #endif
488
489
490

```

Lab 9

Person.cpp

```
C++ Person.cpp > printInfo0
14 Person::Person()
15 {
16     name = " ";
17     age = 0;
18 }
19
20 /**
21  * the constructor of Person that set name to pname and age to page
22  * @param pname, page
23  */
24 Person::Person(string pname, int page)
25 {
26     name = pname;
27     age = page;
28 }
29
30 /**
31  * the member function of Person that set name to n
32  * @param n
33  */
34 void Person::set_name(string n)
35 {
36     name = n;
37 }
38
39 /**
40  * the member function of Person that set age to a
41  * @param a
42  */
43 void Person::set_age(int a)
44 {
45     age = a;
46 }
47
48 /**
49  * the member function of Person that set name to n, age to a
50  * @param n, a
51  */
52 void Person::set_info(string n, int a)
53 {
54     name = n;
55     age = a;
56 }
57
58 /**
59  * the member function of Person that return name
60  * @return name
61  */
62 string Person::get_name() const
63 {
64     return name;
65 }
66
67 /**
68  * the member function of Person that return age
69  * @return age
70  */
71 int Person::get_age() const
72 {
73     return age;
74 }
75
76 /**
77  * the member function that cout name and age in correct form
78  */
79 void Person::printInfo()
80 {
81     cout << "Name: " << name;
82     cout << "\tAge: " << age << endl;
83 }
84
```


Lab 9

Person.h

```
Person.h > Person
17 class Person
18 {
19 public:
20 /**
21  * the default constructor of Person
22  */
23 Person();
24 /**
25  * the constructor of Person that set name to pname and age to page
26  * @param pname, page
27  */
28 Person(string pname, int page);
29 /**
30  * the member function of Person that set name to n
31  * @param n
32  */
33 void set_name(string n);
34 /**
35  * the member function of Person that set age to a
36  * @param a
37  */
38 void set_age(int a);
39 /**
40  * the member function of Person that set name to n, age to a
41  * @param n, a
42  */
43 void set_info(string n, int a);
44 /**
45  * the member function of Person that return name
46  * @return name
47  */
48 string get_name() const;
49 /**
50  * the member function of Person that return age
51  * @return age
52  */
53 int get_age() const;
54 /**
55  * the member function that cout name and age in correct form
56  */
57 void printInfo();
58 private:
59 string name;
60 int age; /* 0 if unknown */
61 };
62
63 #endif
64
65
```

printPersonInfo.cpp

```
C++ printPersonInfo.cpp > printPersonInfo(List<Person>&)
4  @version 1.0 11/21/21
5  */
6  #include "LinkedList.h"
7  #include "Person.h"
8  #include <iostream>
9  #include <typeinfo>
10 #include <iomanip>
11
12 using namespace std;
13
14 /**
15  * the function that print the list of person information in correct format
16  * @param personList(any type)
17  */
18 void printPersonInfo(List<Person> &personList)
19 {
20     Person *f;
21     f = (Person *)personList.getInfo(0); // get node based on key
22     // f->printInfo();
23     ListNode<Person> *currentPtr;
24     currentPtr = personList.getFirstPtr();
25     f = (Person *)currentPtr; // convert to correct data type
26     f->printInfo();
27     cout << "The Employee list is: \n";
28     // print out all the info in linked list
29     while (currentPtr != 0) // display elements in list
30     {
31         f = (Person *)currentPtr; // convert to correct data type
32         f->printInfo();
33         currentPtr = currentPtr->getNextPtr();
34     } // end while
35     cout << endl;
36 }
37
```

Lab 9

printWineInfo.cpp

```
C++ printWineInfo.cpp > printWineInfo(List<Wine>&)  
9  
10 #include <iostream>  
11 #include <typeinfo>  
12 #include <iomanip>  
13  
14 using namespace std;  
15  
16 /**  
17  * the function that print the list of wine information in correct format  
18  * @param wineList(any type)  
19  */  
20 void printWineInfo(List<Wine> &wineList)  
21 {  
22     Wine *wp;  
23     wp = (Wine *)wineList.getInfo(0); // get node based on key  
24  
25     ListNode<Wine> *currentPtr;  
26  
27     currentPtr = wineList.getFirstPtr();  
28     // move the pointer to the next node  
29     currentPtr = currentPtr->getNextPtr();  
30     wp = (Wine *)currentPtr;  
31     wp->printInfo();  
32     cout << "The Wine list is: \n";  
33     // print out all the info in linked list  
34     while (currentPtr != 0) // display elements in list  
35     {  
36         wp = (Wine *)currentPtr; // convert to correct data type  
37         wp->printInfo();  
38         currentPtr = currentPtr->getNextPtr();  
39     } // end while  
40 }  
41
```

Lab 9

Wine.cpp

C++ Wine.cpp > printInfo()

```
12 Wine::Wine()
13 {
14     price = 0;
15 }
16
17 /**
18  * the constructor that set the name of the class to parameter name,
19  * set the vintage of the class to parameter vintage,
20  * set the score of the class to parameter score,
21  * set the price of the class to parameter price,
22  * set the type of the class to parameter type.
23  * @param name, vintage, score, price, type
24  */
25 Wine::Wine(string name, int vintage, int score, double price, string type)
26 {
27     this->name = name;
28     this->vintage = vintage;
29     this->score = score;
30     this->price = price;
31     this->type = type;
32 }
33
34 /**
35  * the member function that set the name of the class to parameter name,
36  * set the vintage of the class to parameter vintage,
37  * set the score of the class to parameter score,
38  * set the price of the class to parameter price,
39  * set the type of the class to parameter type.
40  * @param name, vintage, score, price, type
41  */
42 void Wine::setInfo(string name, int vintage, int score, double price, string type)
43 {
44
45     this->name = name;
46     this->vintage = vintage;
47     this->score = score;
48     this->price = price;
49     this->type = type;
50 }
51
52 /**
53  * the member function that set the price of this class to parameter price
54  * @param price
55  */
56 void Wine::setPrice(double price)
57 {
58     this->price = price;
59 }
60
```

C++ Wine.cpp > printInfo()

```
51
52 /**
53  * the member function that set the price of this class to parameter price
54  * @param price
55  */
56 void Wine::setPrice(double price)
57 {
58     this->price = price;
59 }
60
61 /**
62  * the member function that return name
63  * @return name
64  */
65 string Wine::getName() const
66 {
67     return name;
68 }
69
70 /**
71  * the member function that return price
72  * @return price
73  */
74 int Wine::getPrice() const
75 {
76     return price;
77 }
78
79 /**
80  * the member function that cout name, price, score, year and vintage in correct format
81  */
82 void Wine::printInfo()
83 {
84     /*
85     * use setw() function to make the output print out
86     * nicely formatted.
87     cout << name << type;
88     cout << price << " ; Rating: " << score << " Year: "
89     | << vintage << endl;
90     */
91     cout << setw(20) << left << name << setw(10) << left << type << setw(10) << left << price << " ; Rating: " << score << " Year: "
92     | << vintage << endl;
93 }
94
```

Lab 9
Wine.h

```
17 class Wine
18 {
19 public:
20     /**
21      * the default constructor that set price to 0
22      */
23     Wine();
24     /**
25      * the constructor that set the name of the class to parameter name,
26      * set the vintage of the class to parameter vintage,
27      * set the score of the class to parameter score,
28      * set the price of the class to parameter price,
29      * set the type of the class to parameter type.
30      * @param name, vintage, score, price, type
31      */
32     Wine(string name, int vintage, int score, double price, string type);
33     /**
34      * the member function that set the name of the class to parameter name,
35      * set the vintage of the class to parameter vintage,
36      * set the score of the class to parameter score,
37      * set the price of the class to parameter price,
38      * set the type of the class to parameter type.
39      * @param name, vintage, score, price, type
40      */
41     void setInfo(string name, int vintage, int score,
42                 double price, string type);
43     /**
44      * the member function that set the price of this class to parameter price
45      * @param price
46      */
47     void setPrice(double price);
48     /**
49      * the member function that return name
50      * @return name
51      */
52     string getName() const;
53     /**
54      * the member function that return price
55      * @return price
56      */
57     int getPrice() const;
58     /**
59      * the member function that cout name, price, score, year and vintage in correct format
60      */
61     void printInfo();
62
63 private:
64     string name;
65     int vintage;
66     int score;
67     double price;
68     string type;
69 };
70
71 #endif
72
```

Lab 9

Program Output:

My output matched the example

```
g++ LinkedList_Test.o printMeFirst.o printPersonInfo.o printWineInfo.o Wine.o Person.o -o LinkedList_Test
arthur@DESKTOP-UP5LF24:/mnt/c/Users/Arthur/Documents/Github/Cpp_Projects/Intermediate C++/Lab 9$ make run
./LinkedList_Test
Program written by: Haichuan Wei
Course Info: CS-116 Linked List Lab
Date: Sun Nov 21 18:32:12 2021

***Print using printPersonInfo***
Name: Ron      Age: 22
The Employee list is:
Name: Ron      Age: 22
Name: Sha      Age: 30

***Print using printNoteInfo***
The node list is:
Name: Ron      Age: 22
Name: Sha      Age: 30

***Print using printWineInfo***
Vermentino      White 27 ; Rating: 85  Year: 2014
The Wine list is:
Vermentino      White 27 ; Rating: 85  Year: 2014
Prisoner        Red 44.99 ; Rating: 92  Year: 2014
Stags Chardonnay Carneros  White 45 ; Rating: 89  Year: 2013
Castello Barone Reserve Cabernet  Red 92 ; Rating: 92  Year: 2011
Futo Bordeaux Red  Red 324.99 ; Rating: 97  Year: 2009

***Print using printNoteInfo***
The node list is:
Harlan Estate Bordeaux      Red 850 ; Rating: 97  Year: 2011
Vermentino      White 27 ; Rating: 85  Year: 2014
Prisoner        Red 44.99 ; Rating: 92  Year: 2014
Stags Chardonnay Carneros  White 45 ; Rating: 89  Year: 2013
Castello Barone Reserve Cabernet  Red 92 ; Rating: 92  Year: 2011
Futo Bordeaux Red  Red 324.99 ; Rating: 97  Year: 2009

***AFTER REMOVING front node, Print using printNoteInfo***

***Print using printNoteInfo***
The node list is:
Vermentino      White 27 ; Rating: 85  Year: 2014
Prisoner        Red 44.99 ; Rating: 92  Year: 2014
Stags Chardonnay Carneros  White 45 ; Rating: 89  Year: 2013
Castello Barone Reserve Cabernet  Red 92 ; Rating: 92  Year: 2011
Futo Bordeaux Red  Red 324.99 ; Rating: 97  Year: 2009

***AFTER REMOVING last node, Print using printNoteInfo***

***Print using printNoteInfo***
The node list is:
Vermentino      White 27 ; Rating: 85  Year: 2014
Prisoner        Red 44.99 ; Rating: 92  Year: 2014
Stags Chardonnay Carneros  White 45 ; Rating: 89  Year: 2013
Castello Barone Reserve Cabernet  Red 92 ; Rating: 92  Year: 2011

Linked List using int
0 1 2 3 4
9 8 7 6 5

arthur@DESKTOP-UP5LF24:/mnt/c/Users/Arthur/Documents/Github/Cpp_Projects/Intermediate C++/Lab 9$ _
```