Lab 9

Name: Haichuan Wei

Lab: Lab 9

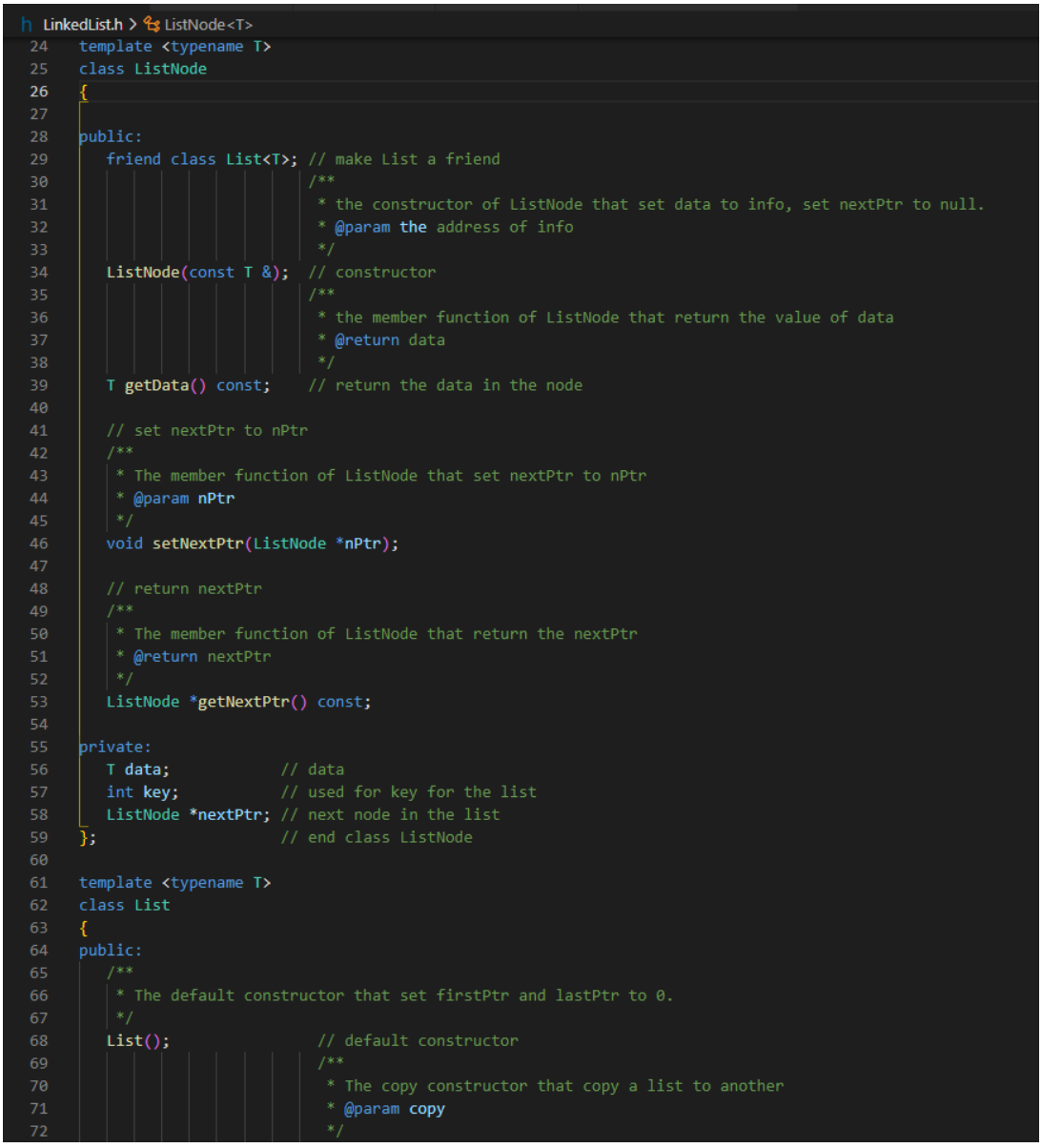
Date: 11/21/2021

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



Lab 9

LinkedList.h > ListNode<T>

```
                                * @param copy
                                */
    List(const List<T> &); // copy constructor
                               /**
                                * The destructor of List that delete data and nodes
                                */
    ~List();                   // destructor
                               /**
                                * The member function that insert a node at the front of the list
                                * @param value, key
                                */
    void insertAtFront(const T &, int);
    /**
     * The member function of List that insert at the back of the list
     * @param value, key
     */
    void insertAtBack(const T &, int);
    /**
     * The member function that remove one from front
     * @param value
     */
    bool removeFromFront(T &);
    /**
     * The member function that remove one from back
     * @param value
     */
    bool removeFromBack(T &);
    /**
     * The member function that check whether the first ptr is null
     * @return true if the firstPtr is null, false if it isn't
     */
    bool isEmpty() const;
    /**
     * The member function of List that print the contents of the List
     */
    void print() const;
    void printPtrFunc();
    /**
     * The member function of List that display the contents of the List
     * @param myKey
     * @return NULL until can't find anything else
     */
    T *getInfo(int myKey);
    // return nextPtr
    /**
     * The member function of List that return the firstPtr
     * @return firstPtr
     */
    ListNode<T> *getFirstPtr() const; // end function getNextPtr
```

Lab 9
LinkedList_Test.cpp

LinkedList.h > ...

```h
template <typename T>
class ListNode
{

public:
   friend class List<T>; // make List a friend
                        /**
                         * the constructor of ListNode that set data to info, set nextPtr to null.
                         * @param the address of info
                         */
   ListNode(const T &);  // constructor
                        /**
                         * the member function of ListNode that return the value of data
                         * @return data
                         */
   T getData() const;    // return the data in the node

   // set nextPtr to nPtr
   /**
    * The member function of ListNode that set nextPtr to nPtr
    * @param nPtr
    */
   void setNextPtr(ListNode *nPtr);

   // return nextPtr
   /**
    * The member function of ListNode that return the nextPtr
    * @return nextPtr
    */
   ListNode *getNextPtr() const;

private:
   T data;           // data
   int key;          // used for key for the list
   ListNode *nextPtr; // next node in the list
};                   // end class ListNode

template <typename T>
class List
{
public:
   /**
    * The default constructor that set firstPtr and lastPtr to 0.
    */
   List();               // default constructor
                        /**
                         * The copy constructor that copy a list to another
                         * @param copy
                         */
   List(const List<T> &); // copy constructor
                        /**
                         * The destructor of List that delete data and nodes
                         */
   ~List();              // destructor
                        /**
                         * The member function that insert a node at the front of the list
                         * @param value, key
                         */
   void insertAtFront(const T &, int);
   /**
    * The member function of List that insert at the back of the list
    * @param value, key
    */
   void insertAtBack(const T &, int);
   /**
    * The member function that remove one from front
    * @param value
    */
   bool removeFromFront(T &);
   /**
    * The member function that remove one from back
    * @param value
    */
   bool removeFromBack(T &);
   /**
    * The member function that check whether the first ptr is null
    * @return true if the firstPtr is null, false if it isn't
    */
   bool isEmpty() const;
   /**
    * The member function of List that print the contents of the List
    */
   void print() const;
   void printPtrFunc();
   /**
    * The member function of List that display the contents of the List
    * @param myKey
    * @return NULL until can't find anything else
    */
   T *getInfo(int myKey);
   // return nextPtr
   /**
    * The member function of List that return the firstPtr
    * @return firstPtr
    */
   ListNode<T> *getFirstPtr() const; // end function getNextPtr
```

Lab 9

```
protected:
   ListNode<T> *firstPtr; // pointer to first node
   ListNode<T> *lastPtr;  // pointer to last node

   // Utility function to allocate a new node
   /**
    * The member function that return the new node
    * @param value and an integer
    * @return ptr
    */
   ListNode<T> *getNewNode(const T &, int);
}; // end typename Template List

// constructor
/**
 * the constructor of ListNode that set data to info, set nextPtr to null.
 * @param the address of info
 */
template <typename T>
ListNode<T>::ListNode(const T &info)
{
   data = info;
   nextPtr = 0;
} // end constructor

// return a copy of the data in the node
/**
 * the member function of ListNode that return the value of data
 * @return data
 */
template <typename T>
T ListNode<T>::getData() const
{
   return data;
} // end function getData

// set nextPtr to nPtr
/**
 * The member function of ListNode that set nextPtr to nPtr
 * @param nPtr
 */
template <typename T>
void ListNode<T>::setNextPtr(ListNode *nPtr)
{
   nextPtr = nPtr;
} // end function setNextPtr
```

```
// return nextPtr
/**
 * The member function of ListNode that return the nextPtr
 * @return nextPtr
 */
template <typename T>
ListNode<T> *ListNode<T>::getNextPtr() const
{
   return nextPtr;
} // end function getNextPtr

// default constructor
/**
 * The default constructor that set firstPtr and lastPtr to 0.
 */
template <typename T>
List<T>::List()
{
   firstPtr = lastPtr = 0;
} // end constructor

// copy constructor
/**
 * The copy constructor of List that copy a list to another
 * @param copy
 */
template <typename T>
List<T>::List(const List<T> &copy)
{
   firstPtr = lastPtr = 0; // initialize pointers

   ListNode<T> *currentPtr = copy.firstPtr;

   // insert into the list
   while (currentPtr != 0)
   {
      insertAtBack(currentPtr->data);
      currentPtr = currentPtr->nextPtr;
   } // end while
} // end List copy constructor

// destructor
/**
 * The destructor of List that delete data and nodes
 */
template <typename T>
List<T>::~List()
```

Lab 9

```cpp
List<T>::~List()
{
   if (!isEmpty()) // List is not empty
   {
      // cout << "Destroying nodes ...\n";

      ListNode<T> *currentPtr = firstPtr;
      ListNode<T> *tempPtr;

      while (currentPtr != 0) // delete remaining nodes
      {
         tempPtr = currentPtr;
         //         cout << tempPtr->data << ' ';
         currentPtr = currentPtr->nextPtr;
         delete tempPtr;
      } // end while
   }    // end if

   //  cout << "\nAll nodes destroyed\n\n";
} // end destructor

// Insert a node at the front of the list
/**
 * The member function that insert a node at the front of the list
 * @param value, key
 */
template <typename T>
void List<T>::insertAtFront(const T &value,
                            int key)
{
   ListNode<T> *newPtr = getNewNode(value, key);

   if (isEmpty()) // List is empty
      firstPtr = lastPtr = newPtr;
   else // List is not empty
   {
      /*
       * write code to implement insert at front
       * 1. the new node needs to point to the first node
       * 2. first node needs to point to the new node
       * */
      newPtr->nextPtr = firstPtr;
      firstPtr = newPtr;

   } // end else
} // end function insertAtFront

// Insert a node at the back of the list
/**
 * The member function of List that insert at the back of the list
 * @param value, key
 */
template <typename T>
void List<T>::insertAtBack(const T &value, int key)
{
   ListNode<T> *newPtr = getNewNode(value, key);

   if (isEmpty()) // List is empty
      firstPtr = lastPtr = newPtr;
   else // List is not empty
   {
      /*
       * write code to implement insert at back
       * 1. next pointer of the last node points to the new node
       * 2. last node needs to point to the new node
       */

      lastPtr->nextPtr = newPtr;
      lastPtr = newPtr;

   } // end else
} // end function insertAtBack

// Delete a node from the front of the list
/**
 * The member function that remove one from front
 * @param value
 */
template <typename T>
bool List<T>::removeFromFront(T &value)
{
   if (isEmpty())   // List is empty
      return false; // delete unsuccessful
   else
   {
      ListNode<T> *tempPtr = firstPtr;

      /*
       * 1. check to see if first pointer is the same as last pointer
       * if it is the same, that means there is only one node, so
       * set both pointer to NULL
       * 2.  if it is not the same, that means there is more than one node
       * in the linked list.  So, make the the first node pointer points
       * to the the next node of the first node
       *
       * don't forget delete the pointer since it has been removed/deleted
```

Lab 9

LinkedList.h > ...

```
        *
        * don't forget delete the pointer since it has been removed/deleted
        *
        * Write your code to implement the remove the 1st node
        *
        * */
       if (firstPtr == lastPtr)
       {
          firstPtr = lastPtr = NULL;
       }
       else
       {
          firstPtr = firstPtr->nextPtr;
       }
       value = tempPtr->data;
       delete tempPtr;
       return true; // delete successful
    }              // end else
 } // end function removeFromFront

 // delete a node from the back of the list
 template <typename T>
 /**
  * The member function that remove one from back
  * @param value
  */
 bool List<T>::removeFromBack(T &value)
 {
    if (isEmpty())
       return false; // delete unsuccessful
    else
    {
       ListNode<T> *tempPtr = lastPtr;

       if (firstPtr == lastPtr)
          firstPtr = lastPtr = 0;
       else
       {
          ListNode<T> *currentPtr = firstPtr;

          while (currentPtr->nextPtr != lastPtr)
             currentPtr = currentPtr->nextPtr;

          lastPtr = currentPtr;
          currentPtr->nextPtr = 0;
       } // end else

       value = tempPtr->data;
       delete tempPtr;
       return true; // delete successful
    }              // end else
 } // end function removeFromBack

 // Is the List empty?
 /**
  * The member function that check whether the first ptr is null
  * @return true if the firstPtr is null, false if it isn't
  */
 template <typename T>
 bool List<T>::isEmpty() const
 {
    return firstPtr == 0;
 } // end function isEmpty

 // Return a pointer to a newly allocated node
 /**
  * The member function that return the new node
  * @param value and an integer
  * @return ptr
  */
 template <typename T>
 ListNode<T> *List<T>::getNewNode(
    const T &value, int)
 {
    ListNode<T> *ptr = new ListNode<T>(value);
    return ptr;
 } // end function getNewNode

 // Display the contents of the List
 /**
  * The member function of List that print the contents of the List
  */
 template <typename T>
 void List<T>::print() const
 {
    if (isEmpty()) // empty list
    {
       cout << "The list is empty\n\n";
       return;
    } // end if

    ListNode<T> *currentPtr = firstPtr;

    // cout << "The list is: ";

    while (currentPtr != 0) // display elements in list
    {
```

Lab 9

```
   while (currentPtr != 0) // display elements in list
   {
      int i;
      string s;
      double d;
      char c;
      if (typeid(currentPtr->data).name() == typeid(i).name() ||
          typeid(currentPtr->data).name() == typeid(d).name() ||
          typeid(currentPtr->data).name() == typeid(s).name() ||
          typeid(currentPtr->data).name() == typeid(c).name())
      {
         // data value is a simple data type and can be printed
         cout << currentPtr->data << ' ';
      }
      else
      {
         cout << "Can't print - Not a simple data type (int, string, char, double)\n";
      }
      currentPtr = currentPtr->nextPtr;
   } // end while

   cout << "\n\n";
} // end function print

/**
 * The member function of List that return the firstPtr
 * @return firstPtr
 */
template <typename T>
ListNode<T> *List<T>::getFirstPtr() const
{
   return firstPtr;
} // end function getNextPtr

// Display the contents of the List
/**
 * The member function of List that display the contents of the List
 * @param myKey
 * @return NULL until can't find anything else
 */
template <typename T>
T *List<T>::getInfo(int myKey)
{
   if (isEmpty()) // empty list
   {
      cout << "The list is empty\n\n";
      return NULL;
   } // end if
```

```
      cout << "The list is empty\n\n";
      return NULL;
   } // end if

   ListNode<T> *currentPtr = firstPtr;

   while (currentPtr != 0) // display elements in list
   {
      if (currentPtr->key == myKey) // found
         return (&currentPtr->data);

      currentPtr = currentPtr->nextPtr;
   } // end while

   return NULL; // can't find
} // end function print

/**
 * The function that can hold any type of node list and print the information it has
 * @param nodeList
 */
template <typename T>
void printNoteInfo(List<T> &nodeList)
{
   T *wp;
   wp = (T *)nodeList.getInfo(0); // get node based on key

   ListNode<T> *currentPtr;

   currentPtr = nodeList.getFirstPtr();

   cout << "The node list is: \n";
   // print out all the info in linked list
   while (currentPtr != 0) // display elements in list
   {
      wp = (T *)currentPtr; // convert to correct data type
      wp->printInfo();      // calling the function that is part of the
                            // data type
      currentPtr = currentPtr->getNextPtr();
   } // end while
}
#endif
```

Lab 9

Person.cpp

Person.cpp > printInfo()

```cpp
Person::Person()
{
    name = " ";
    age = 0;
}

/**
 * the constructor of Person that set name to pname and age to page
 * @param pname, page
 */
Person::Person(string pname, int page)
{
    name = pname;
    age = page;
}

/**
 * the member function of Person that set name to n
 * @param n
 */
void Person::set_name(string n)
{
    name = n;
}

/**
 * the member function of Person that set age to a
 * @param a
 */
void Person::set_age(int a)
{
    age = a;
}

/**
 * the member function of Person that set name to n, age to a
 * @param n, a
 */
void Person::set_info(string n, int a)
{
    name = n;
    age = a;
}

/**
 * the memeber function of Person that return name
 * @return name
 */
string Person::get_name() const
{
    return name;
}

/**
 * the member function of Person that return age
 * @return age
 */
int Person::get_age() const
{
    return age;
}

/**
 * the member function that cout name and age in correct form
 */
void Person::printInfo()
{
    cout << "Name: " << name;
    cout << "\tAge: " << age << endl;
}
```

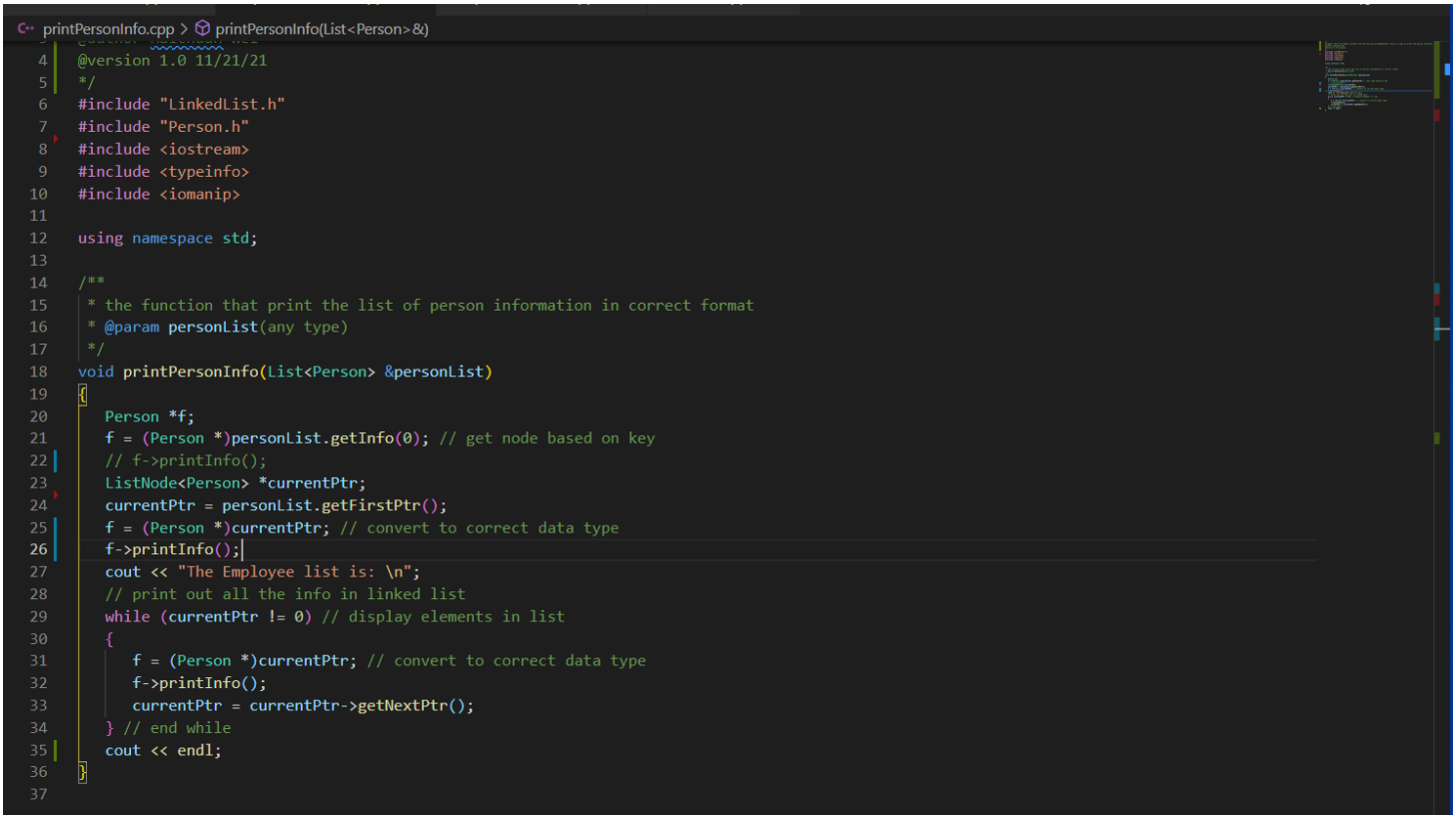
Lab 9
Person.h

```h
class Person
{
public:
   /**
    * the defualt constructor of Person
    */
   Person();
   /**
    * the constructor of Person that set name to pname and age to page
    * @param pname, page
    */
   Person(string pname, int page);
   /**
    * the member function of Person that set name to n
    * @param n
    */
   void set_name(string n);
   /**
    * the member function of Person that set age to a
    * @param a
    */
   void set_age(int a);
   /**
    * the member function of Person that set name to n, age to a
    * @param n, a
    */
   void set_info(string n, int a);
   /**
    * the memeber function of Person that return name
    * @return name
    */
   string get_name() const;
   /**
    * the member function of Person that return age
    * @return age
    */
   int get_age() const;
   /**
    * the member function that cout name and age in correct form
    */
   void printInfo();

private:
   string name;
   int age; /* 0 if unknown */
};

#endif
```

printPersonInfo.cpp

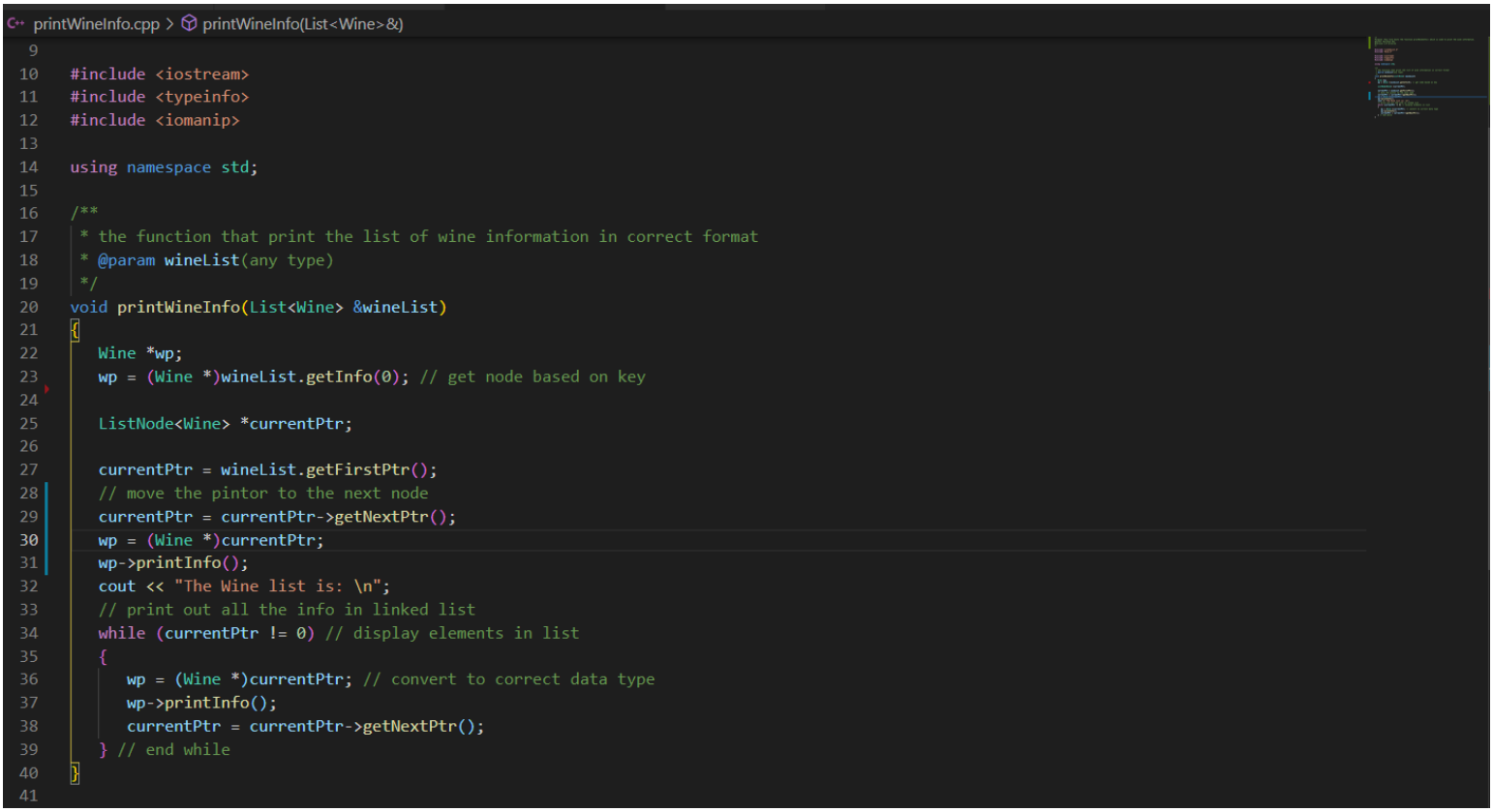


```cpp
@version 1.0 11/21/21
*/
#include "LinkedList.h"
#include "Person.h"
#include <iostream>
#include <typeinfo>
#include <iomanip>

using namespace std;

/**
 * the function that print the list of person information in correct format
 * @param personList(any type)
 */
void printPersonInfo(List<Person> &personList)
{
   Person *f;
   f = (Person *)personList.getInfo(0); // get node based on key
   // f->printInfo();
   ListNode<Person> *currentPtr;
   currentPtr = personList.getFirstPtr();
   f = (Person *)currentPtr; // convert to correct data type
   f->printInfo();
   cout << "The Employee list is: \n";
   // print out all the info in linked list
   while (currentPtr != 0) // display elements in list
   {
      f = (Person *)currentPtr; // convert to correct data type
      f->printInfo();
      currentPtr = currentPtr->getNextPtr();
   } // end while
   cout << endl;
}
```

Lab 9
printWineInfo.cpp



Lab 9

Wine.cpp

Wine.cpp > printInfo()

```cpp
Wine::Wine()
{
    price = 0;
}

/**
 * the consturctor that set the name of the class to parameter name,
 * set the vintage of the class to parameter vintage,
 * set the score of the class to parameter score,
 * set the price of the class to parameter price,
 * set the type of the class to parameter type.
 * @param name, vintage, score, price, type
 */
Wine::Wine(string name, int vintage, int score, double price, string type)
{
    this->name = name;
    this->vintage = vintage;
    this->score = score;
    this->price = price;
    this->type = type;
}

/**
 * the member function that set the name of the class to parameter name,
 * set the vintage of the class to parameter vintage,
 * set the score of the class to parameter score,
 * set the price of the class to parameter price,
 * set the type of the class to parameter type.
 * @param name, vintage, score, price, type
 */
void Wine::setInfo(string name, int vintage, int score, double price, string type)
{

    this->name = name;
    this->vintage = vintage;
    this->score = score;
    this->price = price;
    this->type = type;
}

/**
 * the member function that set the price of this class to parameter price
 * @param price
 */
void Wine::setPrice(double price)
{
    this->price = price;
}
```

Wine.cpp > printInfo()

```cpp

/**
 * the member function that set the price of this class to parameter price
 * @param price
 */
void Wine::setPrice(double price)
{
    this->price = price;
}

/**
 * the member function that return name
 * @return name
 */
string Wine::getName() const
{
    return name;
}

/**
 * the member function that return price
 * @return price
 */
int Wine::getPrice() const
{
    return price;
}

/**
 * the member function that cout name, price, score, year and vintage in correct format
 */
void Wine::printInfo()
{
    /*
     * use setw() function to make the output print out
     * nicely formated.
    cout <<  name <<   type;
    cout <<   price << " ; Rating: " << score << "  Year: "
         << vintage << endl;
     */
    cout << setw(20) << left << name << setw(10) << left << type << setw(10) << left << price << " ; Rating: " << score << "  Year: "
         << vintage << endl;
}

```

Lab 9
Wine.h

```
17  class Wine
18  {
19  public:
20      /**
21       * the defualt constructor that set price to 0
22       */
23      Wine();
24      /**
25       * the consturctor 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$
```