TRƯỜNG ĐẠI HỌC BÁCH KHOA TP.HCM

Khoa Khoa học & Kỹ thuật Máy tính

DATA STRUCTURES & ALGORITHMS

Lab session 2

ADVANCED PROCESSING in LINKED LIST

1. OBJECTIVE

The objectives of Lab 2 are (1) to introduce on the concept of template in C++ and (2) to have students practice with some advanced techniques to process a linked list

2. USING TEMPLATES

Class interface and implementation

For the sake of convenience, C++ allows (and suggests) developers to separate interface and implementation parts when developing a class. Listing 1 illustrates the separation. In this listing, the interface for class List is declared first. Note that, the parameters of its methods are declared by only the data type. For example, the method void addFirst(int) is about to receive an input of type int and returns nothing.

The implementation of all methods in the class List can be declared after that. Note that, the method should be prefixed by the class name and a double colon (::) and the parameter names should be declared. For example, the method addFirst is implemented as void List::addFirst(int newdata).

```
//just an entry in the list, a "struct++" in fact
class Node {
    public:
        int data;
        Node* next;
};

//interface part
class List {
   private:
       int count;
       Node* pHead;
   public:
       List() ;
       void addFirst(int) ;
       void display();
       ~List();
};

//implementation part
List::List() {pHead = NULL; count = 0;}

void List::addFirst(int newdata) {
       Node* pTemp = new Node;
       pTemp->data = newdata;
       pTemp->next = pHead;
       pHead = pTemp;
       count++;
}

void List::display() {
       Node* pTemp = pHead;
       while (pTemp != NULL) {
               cout << pTemp->data << "\t";
               pTemp = pTemp->next;
       }
}

List::~List()  {
       Node* pTemp = pHead;
       while (pTemp != NULL) {
               pTemp = pTemp->next;
               delete pHead;
               pHead = pTemp;
       }
}
```

Listing 1

Templates

Template is a useful technique in C++ which is used to reduce tedious and boring repetition work when developing classes. In Listing 2, we intend to develop two linked lists whose element data are *int* and *float* respectively. When programming this way, a lot of copy/paste works must be done, which would drive programmers crazy easily.

```
//just an entry in the list, a "struct++" in fact
class IntNode {
      public:
            int data;
             Node* next;
};
class FloatNode {
     public:
            float data;
            Node* next;
};

//interface part
class IntList {
     private:
            int count;
            IntNode* pHead;
     public:
            IntList() ;
            void addFirst(int) ;
            void display();
            ~IntList();
};

class FloatList {
     private:
            int count;
            FloatNode* pHead;
     public:
            FloatList() ;
            void addFirst(float) ;
```

```
        void display();
        ~FloatList();
};

//implementation part
IntList::IntList() {pHead = NULL; count = 0;}

void IntList::addFirst(int newdata) {
        IntNode* pTemp = new IntNode;
        pTemp->data = newdata;
        pTemp->next = pHead;
        pHead = pTemp;
        count++;
}

void IntList::display() {
        IntNode* pTemp = pHead;
        while (pTemp != NULL) {
                cout << pTemp->data << "\t";
                pTemp = pTemp->next;
        }
}

IntList::~IntList() {
        IntNode* pTemp = pHead;
        while (pTemp != NULL) {
                pTemp = pTemp->next;
                delete pHead;
                pHead = pTemp;
        }
}

FloatList::FloatList() {pHead = NULL; count = 0;}

void FloatList::addFirst(float newdata) {
        FloatNode* pTemp = new FloatNode;
        pTemp->data = newdata;
        pTemp->next = pHead;
        pHead = pTemp;
        count++;
}

void FloatList::display() {
        FloatNode* pTemp = pHead;
        while (pTemp != NULL) {
                cout << pTemp->data << "\t";
                pTemp = pTemp->next;
        }
}
FloatList::~FloatList() {
        FloatNode* pTemp = pHead;
        while (pTemp != NULL) {
                pTemp = pTemp->next;
                delete pHead;
                pHead = pTemp;
        }
}


void main() {
        IntList intList;
        intList.addFirst(5);
        intList.addFirst(3);
        intList.addFirst(2);
        intList.display();

        FloatList floatList;
        floatList.addFirst(5.2);
        floatList.addFirst(3.14);
        floatList.addFirst(2.13);
        floatList.display();
}
```

Listing 2

To deal with this problem, the best way is to use *template* to implement list. First of all, we create a template class of Node.

```
template<class NodeType>
class Node {
    public:
        NodeType data;
        Node<NodeType> * next;
};
```

Listing 3

In Listing 3, we do not clearly declare the type of *data* as whether it is *int* or *float*. Instead we use a general type *NodeType*. The example in Listing 4 makes 2 nodes, one of type int and one of type float.

```
void main() {
        Node<int> intNode;
        intNode.data = 3;
        intNode.next = NULL;

        Node<float> floatNode;
        floatNode.data = 5.2;
        floatNode.next = NULL;
}
```

Listing 4

Having declared the template Node in Listing 3, in the following Listing 5, we declare a template class List of a general type *ListType* using that template Node.

```
//interface part
template<class ListType>
class List {
    private:
        int count;
        Node<ListType>* pHead;
    public:
        List() ;
        void addFirst(ListType) ;
        void display();
        ~List();
};

//implementation part
template<class ListType>
List<ListType>::List() {pHead = NULL; count = 0;}

template<class ListType>
void List<ListType>::addFirst(ListType newdata) {
        Node<ListType>* pTemp = new Node<ListType>;
        pTemp->data = newdata;
        pTemp->next = pHead;
        pHead = pTemp;
        count++;
}

template<class ListType>
void List<ListType>::display() {
        Node<ListType>* pTemp = pHead;
        while (pTemp != NULL) {
                cout << pTemp->data << "\t";
                pTemp = pTemp->next;
        }
}

template<class ListType>
List<ListType>::~List()  {
        Node<ListType>* pTemp = pHead;
        while (pTemp != NULL) {
                pTemp = pTemp->next;
                delete pHead;
                pHead = pTemp;
        }
}
```

Listing 5

After getting this hard work done, in Listing 6 we declare a list of *int*. To do that, just simply replace the generic type *ListType* by the concrete type *int* in the declaration.

```
void main() {
        List<int> intList;
        intList.addFirst(5);
        intList.addFirst(3);
        intList.addFirst(2);
        intList.display();
}
```

Listing 6

Listing 7 illustrates how to handle two lists of different types.

```
void main() {
        List<int> intList;
        intList.addFirst(5);
        intList.addFirst(3);
        intList.addFirst(2);
        intList.display();

        List<float> floatList;
        floatList.addFirst(5.2);
        floatList.addFirst(3.14);
        floatList.addFirst(2.13);
        floatList.display();
}
```

Listing 7

3. EXERCISES

3.1. Define Stack structure and create method pushElement and popElement. Add list of intergers as follows

{12, 5, 79, 82, 21}.

Delete 2 elements and add list of intergers as follows

{43, 31, 35, 57}

Display the final Stack.

3.2. Define Queue structure and create method encodeElement and decodeElement. Add list of intergers as follows

{12, 5, 79, 82, 21}.

Delete 2 elements and add list of intergers as follows

{43, 31, 35, 57}

Display the final Queue

Consider two files *List.h* and *List.cpp* attached. Use this initial code to accomplish the following tasks.

3.3. Use the defined *List*, build a linked list of integers as follows {12, 5, 79, 82, 21, 43, 31, 35, 57}.

3.4. Uncomment the method *printAll* in file *List.cpp*, implement it and use it to display the list built in Exercise 3.1.

3.5. Uncomment the rest of commented methods and implement them. Write some pieces of code in the *main* function to test your implemented methods.

3.6. Write an additional method to remove the element, which is equal to an input data, in a linked list. If there is more than one element satisfied, do not remove any element.

<u>Example</u> aList = {1,2,3,4,5,**6**,7,3,8,3,0,2}

Remove 6 in aList => {1,2,3,4,5,7,3,8,3,0,2} bList = {1,2,**3**,4,5,6,7,**3**,8,**3**,0,2}

Remove 3 in bList => {1,2,**3**,4,5,6,7,**3**,8,**3**,0,2} (bList remains unchanged)

3.7. Write an additional method to remove the last element, which is equal to an input data, in a linked list.

Example aList = {1,2,3,4,5,6,7,3,8,9,**3**,0,0,2}

Remove the last 3 in aList => {1,2,3,4,5,6,7,3,8,9,0,0,2}

3.8. Write an additional method to remove all elements whose position is a prime number in a linked list.

Example aList = {1,2,**8**,**9**,4,**0**,3,**2**}

=> {1,2,4,3}

3.9. Write an additional method to remove all occurrences of an input in a linked list.

Example aList = {1,2,**3**,4,5,6,7,**3**,8,9,**3**,0,0,2}

Remove all 3 in aList => {1,2,4,5,6,7,8,9,0,0,2}

-- End --