```
#ifndef H_StackType
#define H_StackType

#include <iostream>

using namespace std;

//Definition of the node
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type>* link;
};

template<class Type>
class linkedStackType
{
public:
    const linkedStackType<Type>& operator=
        (const linkedStackType<Type>&);
    //overload the assignment operator
    void initializeStack();
    //Initialize the stack to an empty state.
    //Post condition: Stack elements are removed; head = NULL
    bool isEmptyStack();
    //Function returns true if the stack is empty;
    //otherwise, it returns false
    bool isFullStack();
    //Function returns true if the stack is full;
    //otherwise, it returns false

    Type top();

    void push(const Type& newItem);
    //Add the newItem to the stack.
    //Pre condition: stack exists and is not full
    //Post condition: stack is changed and the newItem
    //     is added to the head of stack. head points to
    //     the updated stack
    void pop(Type& poppedElement);
    void pop();
    //Remove the head element of the stack.
    //Pre condition: Stack exists and is not empty
    //Post condition: stack is changed and the head
    //    element is removed from the stack. The head
    //    element of the stack is saved in poppedElement
    void destroyStack();
    //Remove all elements of the stack, leaving the
    //stack in an empty state.
    //Post condition: head = NULL
    linkedStackType();
```

```
53      //default constructor
54      //Post condition: head = NULL
55      linkedStackType(const linkedStackType<Type>& otherStack);
56      //copy constructor
57      ~linkedStackType();
58      //destructor
59      //All elements of the stack are removed from the stack
60
61      void printStack();
62
63  private:
64      nodeType<Type>* head; // pointer to the stack
65  };
66
67
68  template<class Type> //default constructor
69  linkedStackType<Type>::linkedStackType()
70  {
71      head = NULL;
72  }
73
74  template<class Type>
75  void linkedStackType<Type>::destroyStack()
76  {
77      nodeType<Type>* temp; //pointer to delete the node
78
79      while (head != NULL)  //while there are elements in the stack
80      {
81          temp = head;      //set temp to point to the current node
82          head = head->link; //advance head to the next node
83          delete temp;     //deallocate memory occupied by temp
84      }
85  }// end destroyStack
86
87
88
89  template<class Type>
90  void linkedStackType<Type>::initializeStack()
91  {
92      destroyStack();
93  }
94
95  template<class Type>
96  bool linkedStackType<Type>::isEmptyStack()
97  {
98      return(head == NULL);
99  }
100
101 template<class Type>
102 bool linkedStackType<Type>::isFullStack()
103 {
104     return 0;
```

```
105 }
106
107 template<class Type>
108 Type linkedStackType<Type>::top()
109 {
110     return head->info;
111 }
112
113 template<class Type>
114 void linkedStackType<Type>::push(const Type& newElement)
115 {
116     nodeType<Type>* newNode; //pointer to create the new node
117
118     newNode = new nodeType<Type>; //create the node
119     newNode->info = newElement;   //store newElement in the node
120     newNode->link = head;          //insert newNode before head
121     head = newNode;            //set head to point to the head node
122 } //end push
123
124
125 template<class Type>
126 void linkedStackType<Type>::pop(Type& poppedElement)
127 {
128     nodeType<Type>* temp;       //pointer to deallocate memory
129
130     poppedElement = head->info; //copy the head element into
131                                 //poppedElement
132     cout << "Popped item is " << poppedElement << endl;
133     temp = head;                    //set temp to point to the head node
134     head = head->link;              //advance head to the next node
135     delete temp;                    //delete the head node
136 }//end pop
137
138 template<class Type>
139 void linkedStackType<Type>::pop()
140 {
141     nodeType<Type>* temp;       //pointer to deallocate memory
142     temp = head;                    //set temp to point to the head node
143     head = head->link;              //advance head to the next node
144     delete temp;                    //delete the head node
145 }//end pop
146
147
148 template<class Type>   //copy constructor
149 linkedStackType<Type>::linkedStackType(const linkedStackType<Type>& otherStack)
150 {
151     nodeType<Type>* newNode, * current, * last;
152
153     if (otherStack.head == NULL)
154         head = NULL;
155     else
156     {
```

```
157         current = otherStack.head;  //set current to point to the
158                                     //stack to be copied
159
160             //copy the head element of the stack
161         head = new nodeType<Type>;   //create the node
162         head->info = current->info;  //copy the info
163         head->link = NULL;          //set the link field of the
164                                     //node to null
165         last = head;                 //set last to point to the node
166         current = current->link;    //set current to point to the
167                                     //next node
168
169             //copy the remaining stack
170         while (current != NULL)
171         {
172             newNode = new nodeType<Type>;
173             newNode->info = current->info;
174             newNode->link = NULL;
175             last->link = newNode;
176             last = newNode;
177             current = current->link;
178         }//end while
179     }//end else
180 }//end copy constructor
181
182
183 template<class Type> //destructor
184 linkedStackType<Type>::~linkedStackType()
185 {
186     nodeType<Type>* temp;
187
188     while (head != NULL)     //while there are elements in the stack
189     {
190         temp = head;        //set temp to point to the current node
191         head = head->link;  //advance first to the next node
192         delete temp;       //deallocate the memory occupied by temp
193     }//end while
194 }
195 //end destructor
196
197 template<class Type>
198 inline void linkedStackType<Type>::printStack()
199 {
200     cout << "Printing stack:" << endl;
201     nodeType<Type>* tempPtr = head;
202     for (nodeType<Type>* tempPtr = head; tempPtr != NULL; tempPtr = tempPtr-
        >link)
203     {
204         cout << tempPtr->info << endl;
205     }
206     cout << endl;
207 }
```

```
208
209
210 template<class Type>   //overloading the assignment operator
211 const linkedStackType<Type>& linkedStackType<Type>::operator=
212 (const linkedStackType<Type>& otherStack)
213 {
214     nodeType<Type>* newNode, * current, * last;
215
216     if (this != &otherStack) //avoid self-copy
217     {
218         if (head != NULL)  //if the stack is not empty, destroy it
219             destroyStack();
220
221         if (otherStack.head == NULL)
222             head = NULL;
223         else
224         {
225             current = otherStack.head;  //set current to point to
226                                         //the stack to be copied
227
228                 //copy the head element of otherStack
229             head = new nodeType<Type>;  //create the node
230             head->info = current->info; //copy the info
231             head->link = NULL;          //set the link field of the
232                                         //node to null
233             last = head;                //make last point to the node
234             current = current->link;    //make current point to
235                                         //the next node
236
237                 //copy the remaining elements of the stack
238             while (current != NULL)
239             {
240                 newNode = new nodeType<Type>;
241                 newNode->info = current->info;
242                 newNode->link = NULL;
243                 last->link = newNode;
244                 last = newNode;
245                 current = current->link;
246             }//end while
247         }//end else
248     }//end if
249
250     return *this;
251 }//end operator=
252 #endif
```