List, Stack & Queue

LI Hao 李颢, Assoc. Prof. SPEIT & Dept. Automation of SEIEE

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

List

- **Generic list**
  - a **finite**, **ordered** sequence of data items known as *elements*
    - *ordered* means each element has a *position* in the list, namely a list is a set of elements with *ordinal numbers*
  - *length* (empty if 0)
  - *position & current position*
  - *get* (current element)
  - *head & tail*
  - *next & prev*(ious)
  - *move* (to start, end, or any position)
  - *insert & append*
  - *remove*
  - *clear*

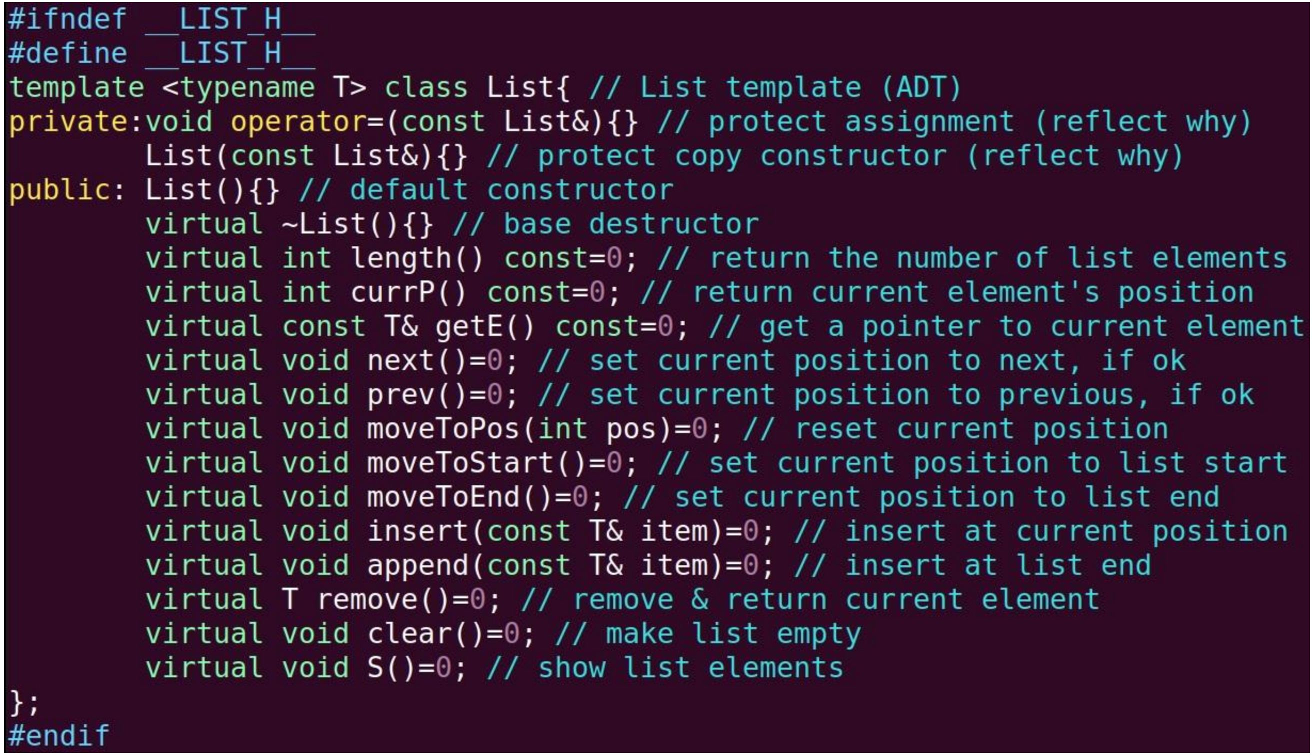
languages
philosophy
mathematics
arts
physics
informatics
literature
engineering
sports
music

structure abstraction

List

- **Generic list**

structure abstraction



```
#ifndef __LIST_H__
#define __LIST_H__
template <typename T> class List{ // List template (ADT)
private:void operator=(const List&){} // protect assignment (reflect why)
        List(const List&){} // protect copy constructor (reflect why)
public: List(){} // default constructor
        virtual ~List(){} // base destructor
        virtual int length() const=0; // return the number of list elements
        virtual int currP() const=0; // return current element's position
        virtual const T& getE() const=0; // get a pointer to current element
        virtual void next()=0; // set current position to next, if ok
        virtual void prev()=0; // set current position to previous, if ok
        virtual void moveToPos(int pos)=0; // reset current position
        virtual void moveToStart()=0; // set current position to list start
        virtual void moveToEnd()=0; // set current position to list end
        virtual void insert(const T& item)=0; // insert at current position
        virtual void append(const T& item)=0; // insert at list end
        virtual T remove()=0; // remove & return current element
        virtual void clear()=0; // make list empty
        virtual void S()=0; // show list elements
};
#endif
```

List

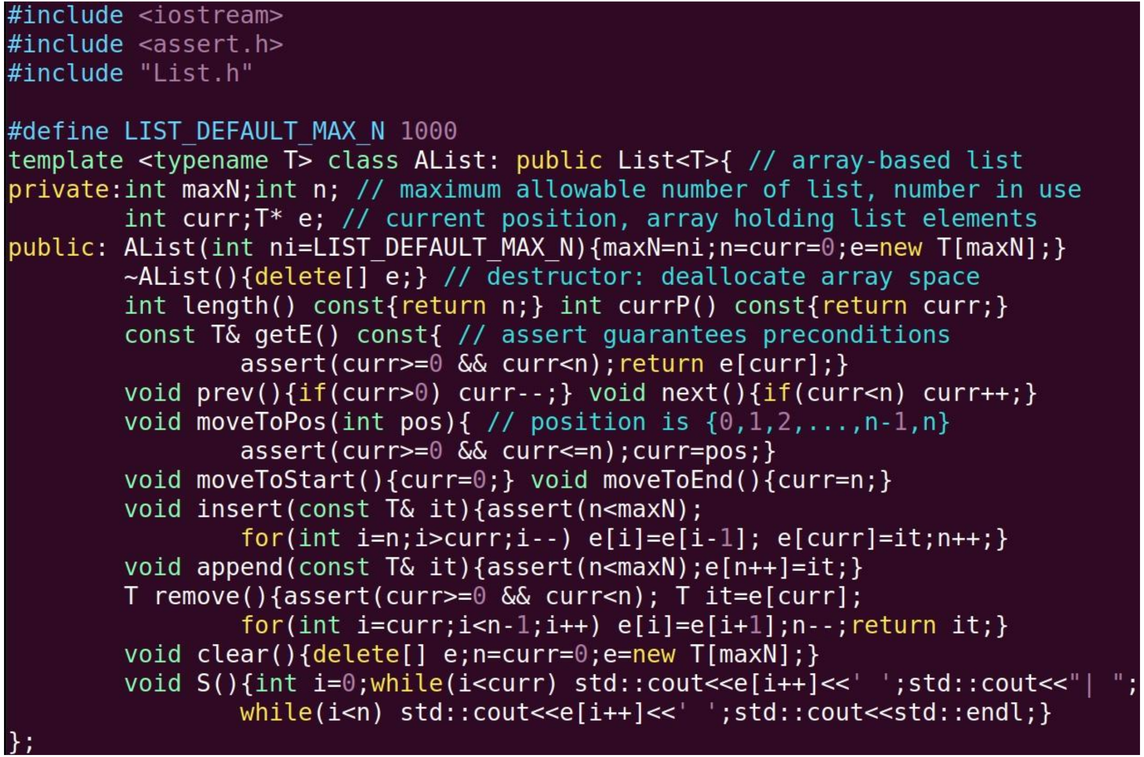
structure abstraction

- **Array-based list implementation**
  - a **finite**, **ordered** sequence of data items known as *elements*
  - *length* (empty if 0)
  - *position & current position*
  - *get* (current element)
  - *head & tail*
  - *next & prev*(ious)
  - *move* (to start, end, or any position)
  - *insert & append*
  - *remove*
  - *clear*

List

structure abstraction

- **Array-based list implementation**

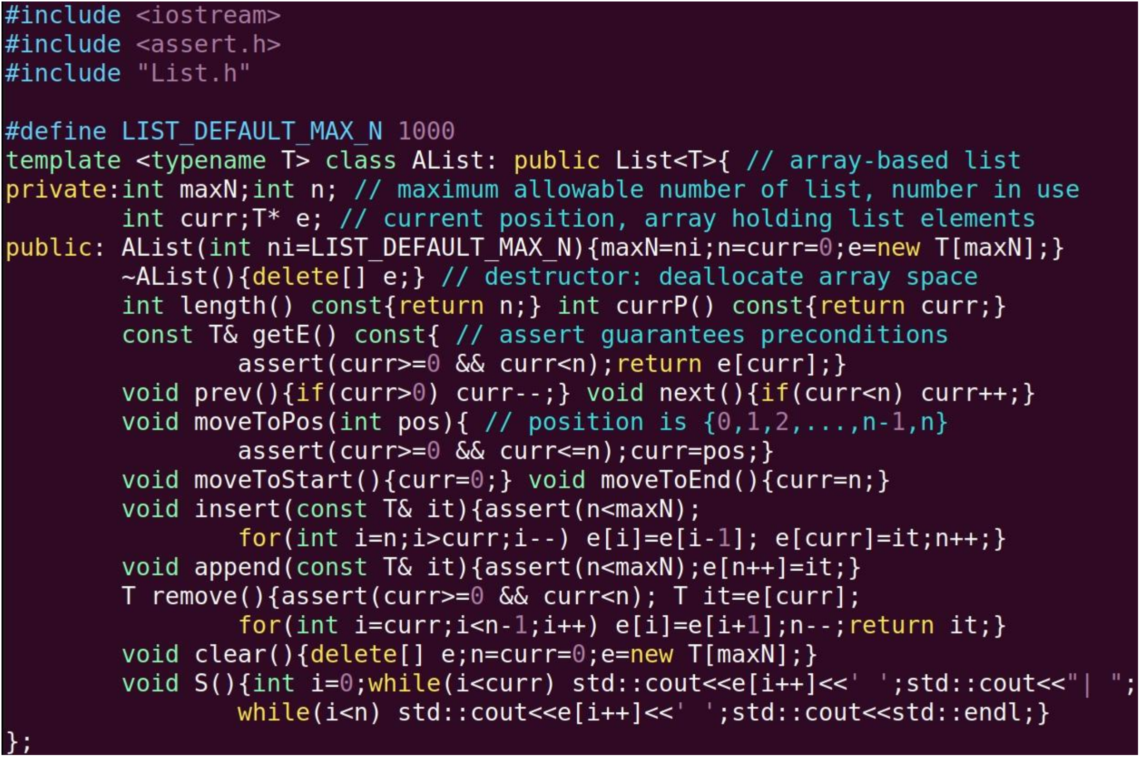


```
#include <iostream>
#include <assert.h>
#include "List.h"

#define LIST_DEFAULT_MAX_N 1000
template <typename T> class AList: public List<T>{ // array-based list
private:int maxN;int n; // maximum allowable number of list, number in use
        int curr;T* e; // current position, array holding list elements
public: AList(int ni=LIST_DEFAULT_MAX_N){maxN=ni;n=curr=0;e=new T[maxN];}
        ~AList(){delete[] e;} // destructor: deallocate array space
        int length() const{return n;} int currP() const{return curr;}
        const T& getE() const{ // assert guarantees preconditions
                assert(curr>=0 && curr<n);return e[curr];}
        void prev(){if(curr>0) curr--;} void next(){if(curr<n) curr++;}
        void moveToPos(int pos){ // position is {0,1,2,...,n-1,n}
                assert(curr>=0 && curr<=n);curr=pos;}
        void moveToStart(){curr=0;} void moveToEnd(){curr=n;}
        void insert(const T& it){assert(n<maxN);
                for(int i=n;i>curr;i--) e[i]=e[i-1]; e[curr]=it;n++;}
        void append(const T& it){assert(n<maxN);e[n++]=it;}
        T remove(){assert(curr>=0 && curr<n); T it=e[curr];
                for(int i=curr;i<n-1;i++) e[i]=e[i+1];n--;return it;}
        void clear(){delete[] e;n=curr=0;e=new T[maxN];}
        void S(){int i=0;while(i<curr) std::cout<<e[i++]<<' ';std::cout<<"| ";
                while(i<n) std::cout<<e[i++]<<' ';std::cout<<std::endl;}
};
```

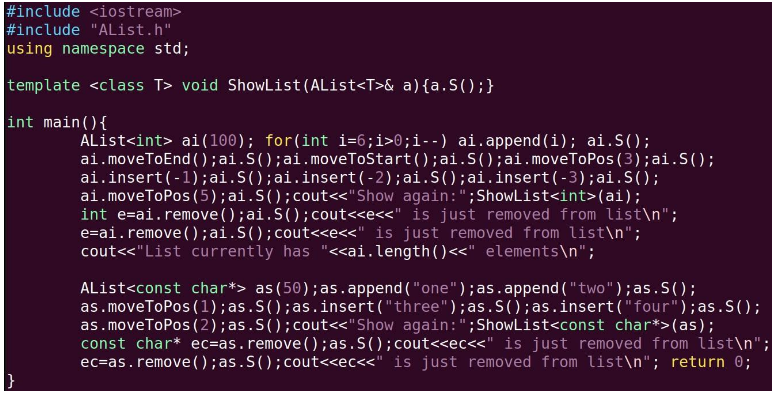
List

Array-based list implementation

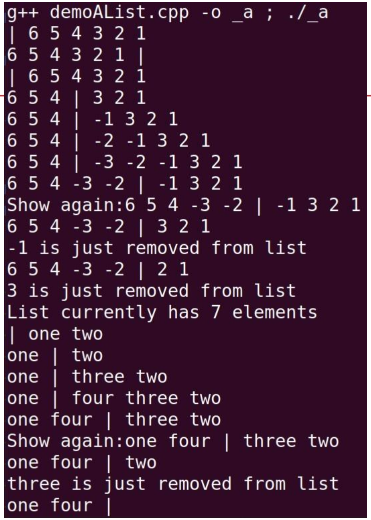
structure abstraction



```
#include <iostream>
#include <assert.h>
#include "List.h"

#define LIST_DEFAULT_MAX_N 1000
template <typename T> class AList: public List<T>{ // array-based list
private:int maxN;int n; // maximum allowable number of list, number in use
        int curr;T* e; // current position, array holding list elements
public: AList(int ni=LIST_DEFAULT_MAX_N){maxN=ni;n=curr=0;e=new T[maxN];}
        ~AList(){delete[] e;} // destructor: deallocate array space
        int length() const{return n;} int currP() const{return curr;}
        const T& getE() const{ // assert guarantees preconditions
                assert(curr>=0 && curr<n);return e[curr];}
        void prev(){if(curr>0) curr--;} void next(){if(curr<n) curr++;}
        void moveToPos(int pos){ // position is {0,1,2,...,n-1,n}
                assert(curr>=0 && curr<=n);curr=pos;}
        void moveToStart(){curr=0;} void moveToEnd(){curr=n;}
        void insert(const T& it){assert(n<maxN);
                for(int i=n;i>curr;i--) e[i]=e[i-1]; e[curr]=it;n++;}
        void append(const T& it){assert(n<maxN);e[n++]=it;}
        T remove(){assert(curr>=0 && curr<n); T it=e[curr];
                for(int i=curr;i<n-1;i++) e[i]=e[i+1];n--;return it;}
        void clear(){delete[] e;n=curr=0;e=new T[maxN];}
        void S(){int i=0;while(i<curr) std::cout<<e[i++]<<' ';std::cout<<"| ";
                while(i<n) std::cout<<e[i++]<<' ';std::cout<<std::endl;}
};
```



```
#include <iostream>
#include "AList.h"
using namespace std;

template <class T> void ShowList(AList<T>& a){a.S();}

int main(){
        AList<int> ai(100); for(int i=6;i>0;i--) ai.append(i); ai.S();
        ai.moveToEnd();ai.S();ai.moveToStart();ai.S();ai.moveToPos(3);ai.S();
        ai.insert(-1);ai.S();ai.insert(-2);ai.S();ai.insert(-3);ai.S();
        ai.moveToPos(5);ai.S();cout<<"Show again:";ShowList<int>(ai);
        int e=ai.remove();ai.S();cout<<e<<" is just removed from list\n";
        e=ai.remove();ai.S();cout<<e<<" is just removed from list\n";
        cout<<"List currently has "<<ai.length()<<" elements\n";

        AList<const char*> as(50);as.append("one");as.append("two");as.S();
        as.moveToPos(1);as.S();as.insert("three");as.S();as.insert("four");as.S();
        as.moveToPos(2);as.S();cout<<"Show again:";ShowList<const char*>(as);
        const char* ec=as.remove();as.S();cout<<ec<<" is just removed from list\n";
        ec=as.remove();as.S();cout<<ec<<" is just removed from list\n"; return 0;
}
```



```
g++ demoAList.cpp -o _a ; ./_a
| 6 5 4 3 2 1
6 5 4 3 2 1 |
| 6 5 4 3 2 1
6 5 4 | 3 2 1
6 5 4 | -1 3 2 1
6 5 4 | -2 -1 3 2 1
6 5 4 | -3 -2 -1 3 2 1
6 5 4 -3 -2 | -1 3 2 1
Show again:6 5 4 -3 -2 | -1 3 2 1
6 5 4 -3 -2 | 3 2 1
-1 is just removed from list
6 5 4 -3 -2 | 2 1
3 is just removed from list
List currently has 7 elements
| one two
one | two
one | three two
one | four three two
one four | three two
Show again:one four | three two
one four | two
three is just removed from list
one four |
```

List

- **Array-based list implementation**
  - a **finite**, **ordered** sequence of data items known as *elements*
  - *length* (empty if 0)
  - *position* & *current position*
  - *get* (current element)
  - *head* & *tail*
  - *next* & *prev*(ious) // each costs O(1)
  - *move* (to start, end, or any position) // each costs O(1)
  - *insert* & *append* // *insert* costs O(n), *append* costs O(1)
  - *remove* // costs O(n)
  - *clear*

List

- **Array-based list implementation**
  - a **finite**, **ordered** sequence of data items known as *elements*
  - *length* (empty if 0)
  - *position* & *current position*
  - *get* (current element)
  - *head* & *tail*
  - *next* & *prev*(ious) // do not take ++ & -- as choice for granted
    - e.g. for *next/prev*, curr=(curr+/-offset)%maxN is a valid choice if (offset,maxN)=1 namely offset & maxN are mutually prime
  - *move* (to start, end, or any position) // do not take direct index as choice for granted
  - *insert* & *append*
  - *remove*
  - *clear*

List

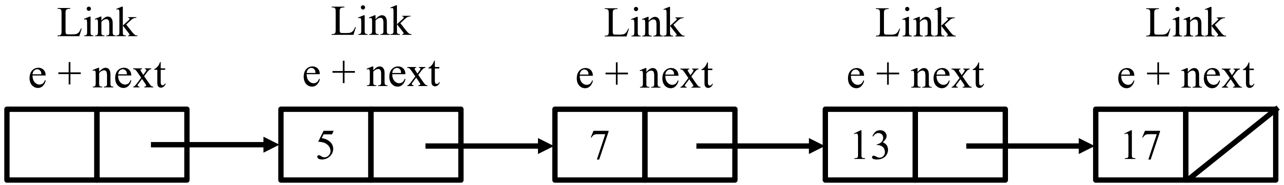
structure abstraction

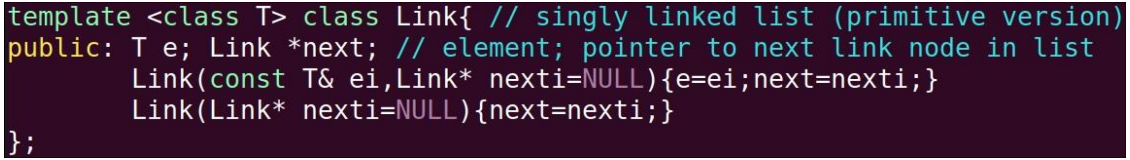
- **Linked list implementation**
  - a **finite**, **ordered** sequence of data items known as *elements*
  - **dynamic memory allocation**: for new list elements as needed
  - *singly linked list*
  - *doubly linked list*

List

structure abstraction

- **Linked list implementation**
  - a **finite**, **ordered** sequence of data items known as *elements*
  - **dynamic memory allocation**: for new list elements as needed
  - *singly linked list*
    - singly *linked node* (class *Link*)
  - *doubly linked list*

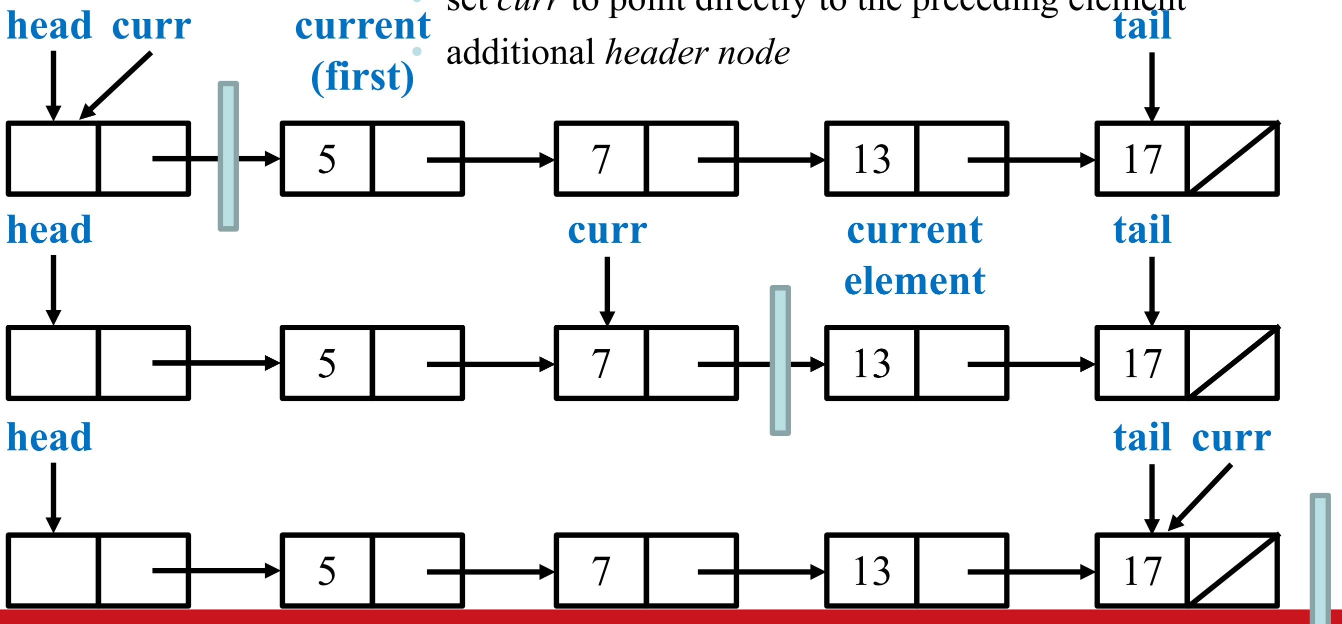




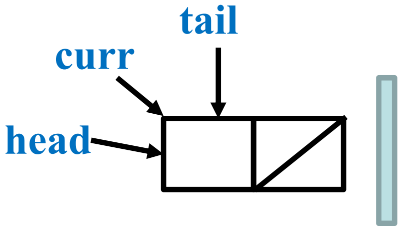
```
template <class T> class Link{ // singly linked list (primitive version)
public: T e; Link *next; // element; pointer to next link node in list
        Link(const T& ei,Link* nexti=NULL){e=ei;next=nexti;}
        Link(Link* nexti=NULL){next=nexti;}
};
```

List

- **Linked list implementation**
  - *singly linked list*
    - singly *linked node*
    - set *curr* to point directly to the preceding element
    - additional *header node*



structure abstraction

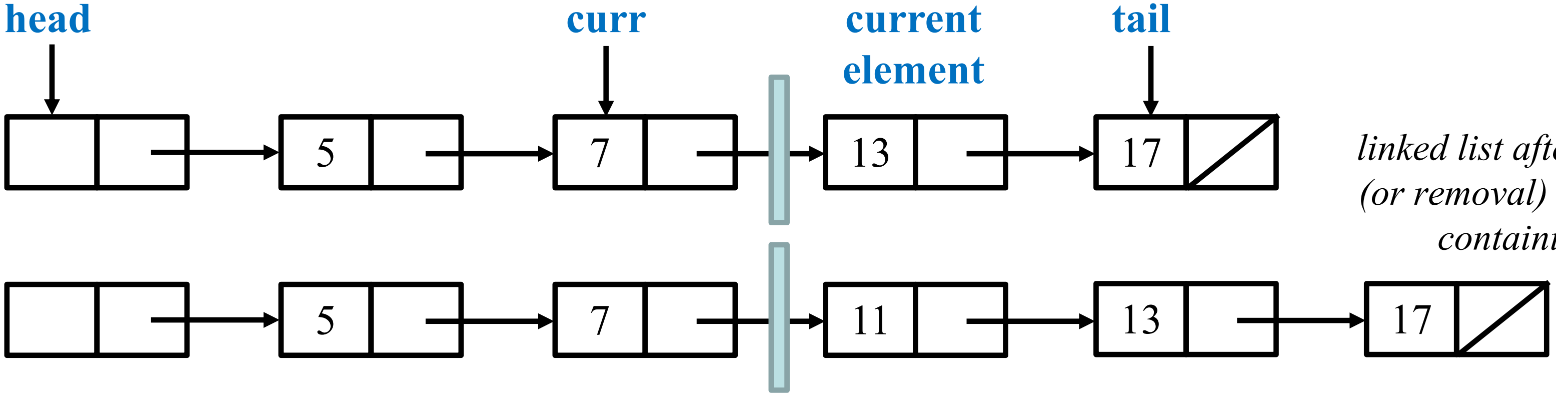


initial state of a linked list when using a header node

List

structure abstraction

- **Linked list implementation**
  - *singly linked list*
    - singly *linked node*
    - set *curr* to point directly to the preceding element
    - additional *header node*
  - *efficient insert & remove*
    - pointer adjustment only, no tedious shifting of elements

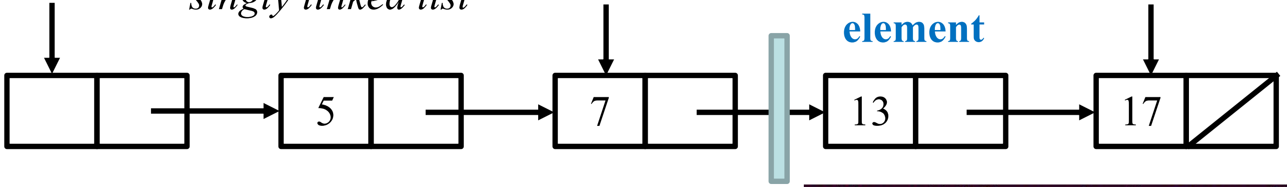


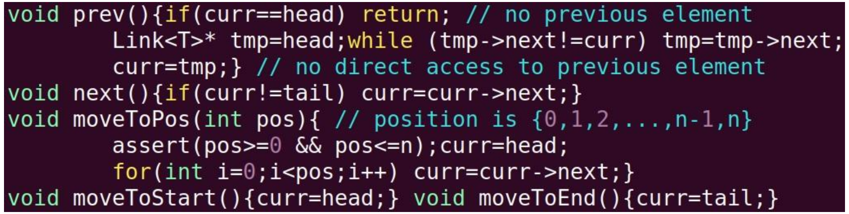
linked list after insertion (or removal) of the node containing 11

List

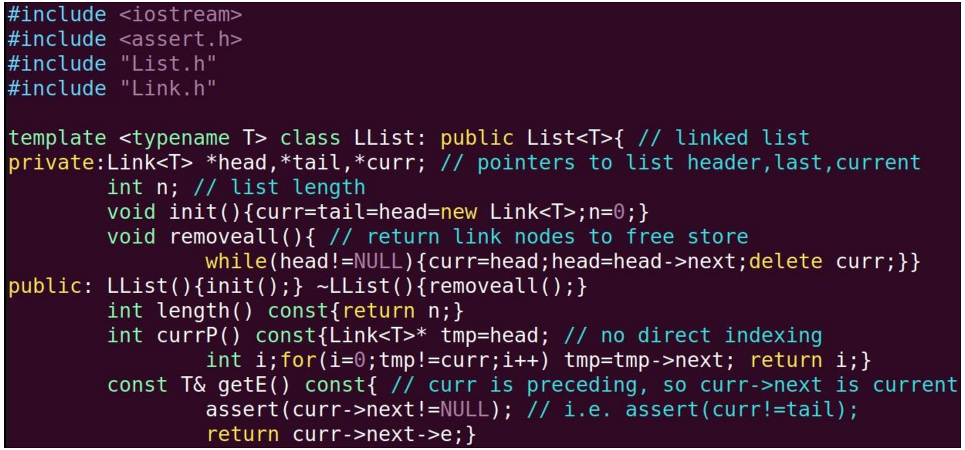
- **Linked list implementation**
  - *singly linked list*

structure abstraction

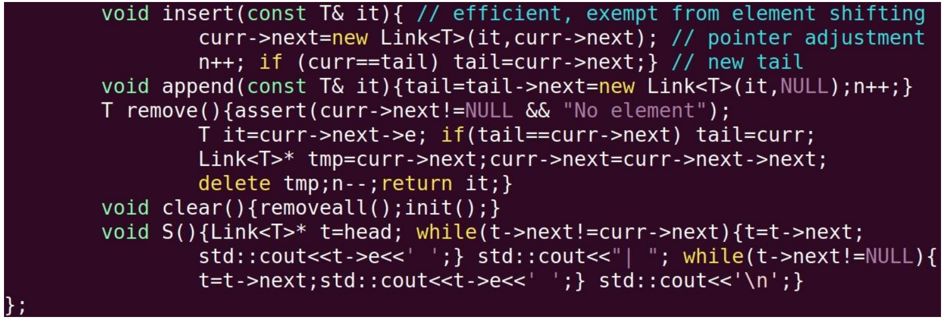




```
void prev(){if(curr==head) return; // no previous element
        Link<T>* tmp=head;while (tmp->next!=curr) tmp=tmp->next;
        curr=tmp;} // no direct access to previous element
void next(){if(curr!=tail) curr=curr->next;}
void moveToPos(int pos){ // position is {0,1,2,...,n-1,n}
        assert(pos>=0 && pos<=n);curr=head;
        for(int i=0;i<pos;i++) curr=curr->next;}
void moveToStart(){curr=head;} void moveToEnd(){curr=tail;}
```



```
#include <iostream>
#include <assert.h>
#include "List.h"
#include "Link.h"

template <typename T> class LList: public List<T>{ // linked list
private:Link<T> *head,*tail,*curr; // pointers to list header,last,current
        int n; // list length
        void init(){curr=tail=head=new Link<T>;n=0;}
        void removeall(){ // return link nodes to free store
                while(head!=NULL){curr=head;head=head->next;delete curr;}}
public: LList(){init();} ~LList(){removeall();}
        int length() const{return n;}
        int currP() const{Link<T>* tmp=head; // no direct indexing
                int i;for(i=0;tmp!=curr;i++) tmp=tmp->next; return i;}
        const T& getE() const{ // curr is preceding, so curr->next is current
                assert(curr->next!=NULL); // i.e. assert(curr!=tail);
                return curr->next->e;}
```

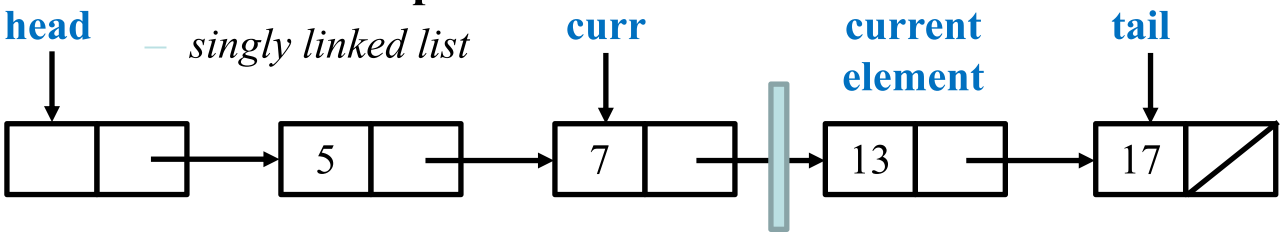


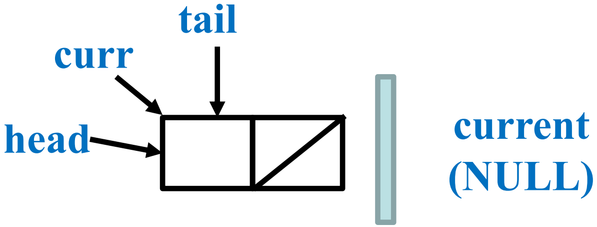
```
        void insert(const T& it){ // efficient, exempt from element shifting
                curr->next=new Link<T>(it,curr->next); // pointer adjustment
                n++; if (curr==tail) tail=curr->next;} // new tail
        void append(const T& it){tail=tail->next=new Link<T>(it,NULL);n++;}
        T remove(){assert(curr->next!=NULL && "No element");
                T it=curr->next->e; if(tail==curr->next) tail=curr;
                Link<T>* tmp=curr->next;curr->next=curr->next->next;
                delete tmp;n--;return it;}
        void clear(){removeall();init();}
        void S(){Link<T>* t=head; while(t->next!=curr->next){t=t->next;
                std::cout<<t->e<<' ';} std::cout<<"| "; while(t->next!=NULL){
                t=t->next;std::cout<<t->e<<' ';} std::cout<<'\n';}
};
```

List

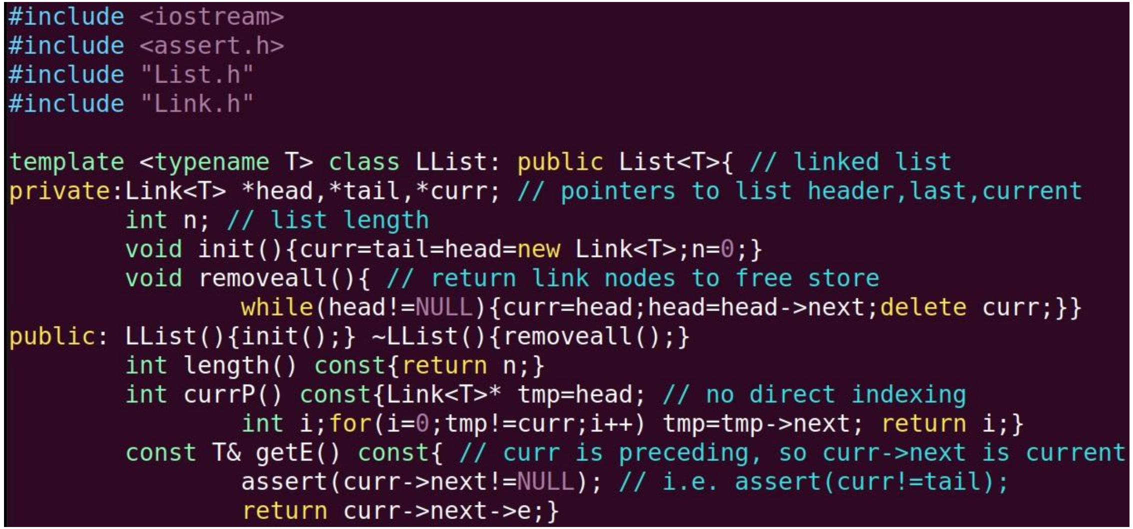
- **Linked list implementation**
  - *singly linked list*

structure abstraction





initial state of a linked list when using a header node



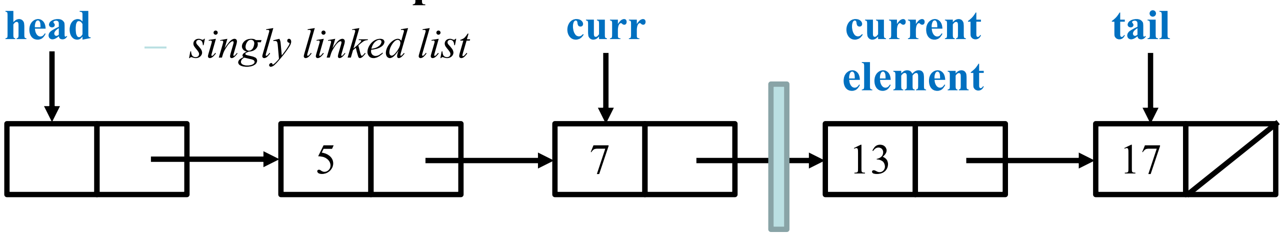
```
#include <iostream>
#include <assert.h>
#include "List.h"
#include "Link.h"

template <typename T> class LList: public List<T>{ // linked list
private:Link<T> *head,*tail,*curr; // pointers to list header,last,current
        int n; // list length
        void init(){curr=tail=head=new Link<T>;n=0;}
        void removeall(){ // return link nodes to free store
                while(head!=NULL){curr=head;head=head->next;delete curr;}}
public: LList(){init();} ~LList(){removeall();}
        int length() const{return n;}
        int currP() const{Link<T>* tmp=head; // no direct indexing
                int i;for(i=0;tmp!=curr;i++) tmp=tmp->next; return i;}
        const T& getE() const{ // curr is preceding, so curr->next is current
                assert(curr->next!=NULL); // i.e. assert(curr!=tail);
                return curr->next->e;}
```

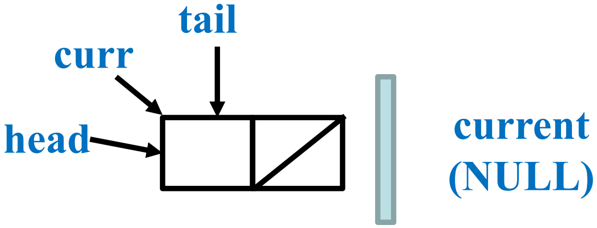
reflect why *getE*() returns *const T&*: 1) why *&*? 2) why *const*?

List

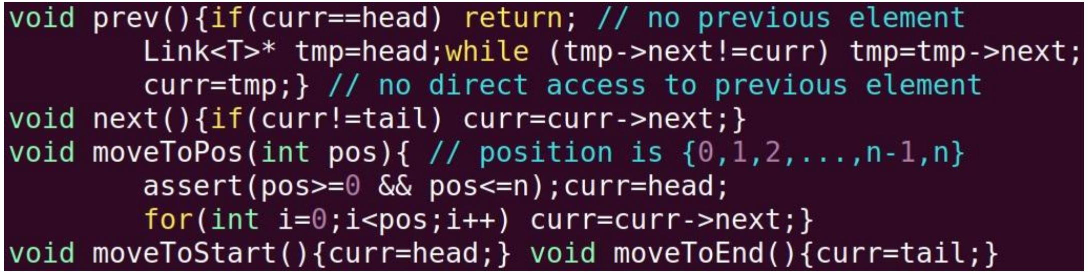
- **Linked list implementation**
  - *singly linked list*

structure abstraction





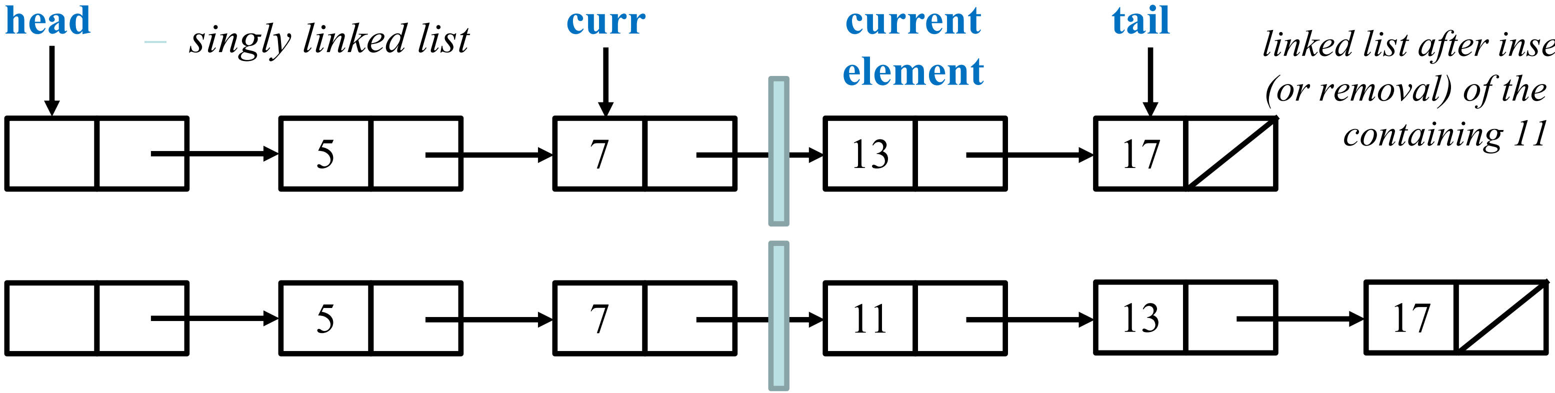
initial state of a linked list when using a header node



```
void prev(){if(curr==head) return; // no previous element
        Link<T>* tmp=head;while (tmp->next!=curr) tmp=tmp->next;
        curr=tmp;} // no direct access to previous element
void next(){if(curr!=tail) curr=curr->next;}
void moveToPos(int pos){ // position is {0,1,2,...,n-1,n}
        assert(pos>=0 && pos<=n);curr=head;
        for(int i=0;i<pos;i++) curr=curr->next;}
void moveToStart(){curr=head;} void moveToEnd(){curr=tail;}
```

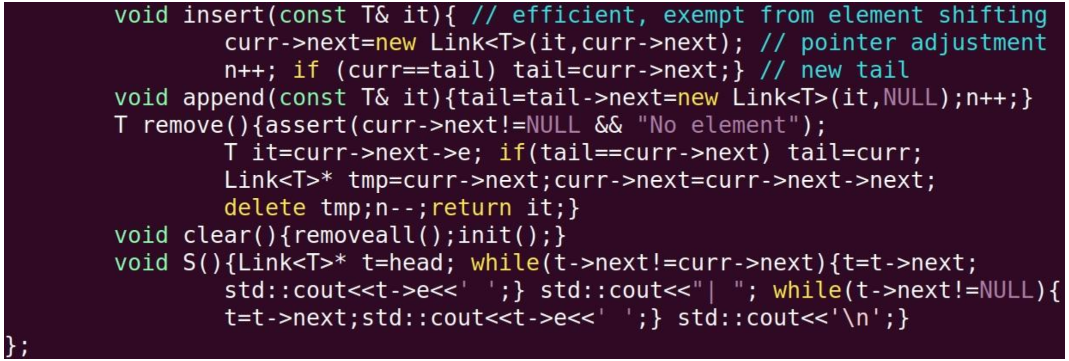
List

- **Linked list implementation**
  - *singly linked list*



structure abstraction

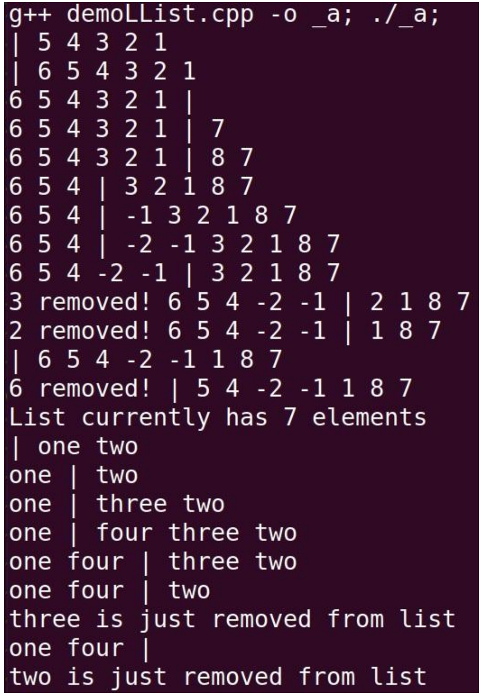
linked list after insertion (or removal) of the node containing 11



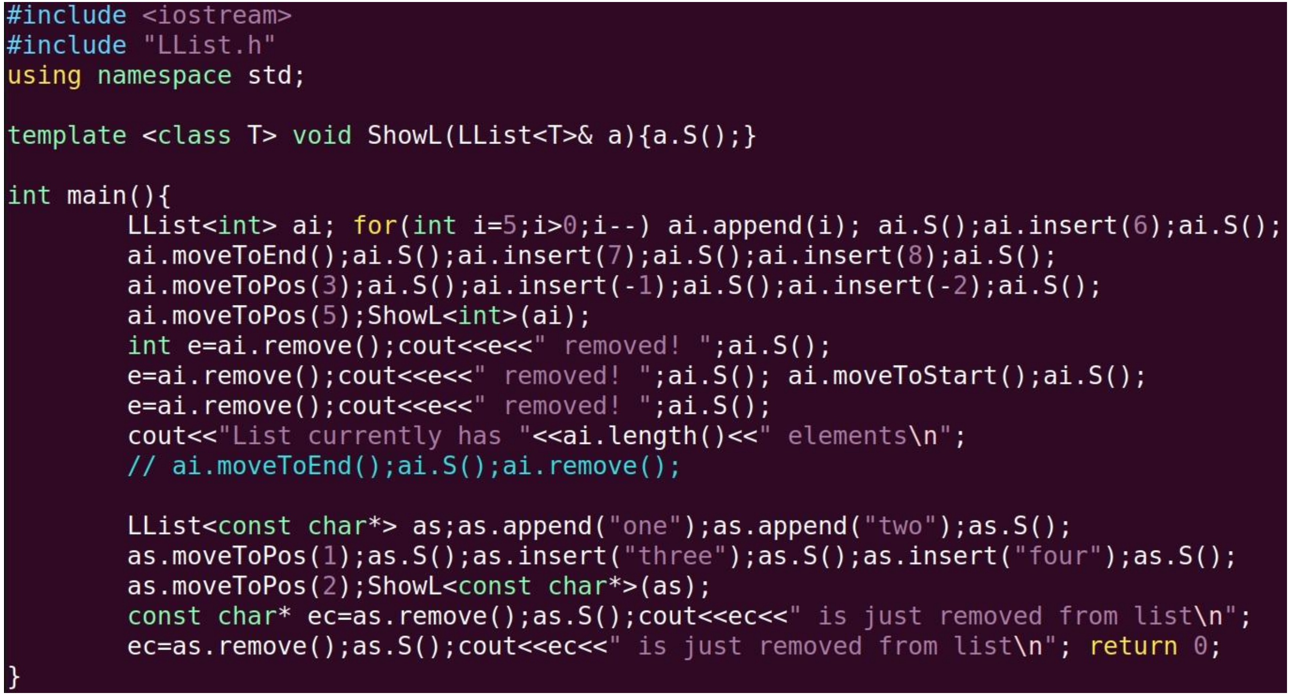
```
        void insert(const T& it){ // efficient, exempt from element shifting
                curr->next=new Link<T>(it,curr->next); // pointer adjustment
                n++; if (curr==tail) tail=curr->next;} // new tail
        void append(const T& it){tail=tail->next=new Link<T>(it,NULL);n++;}
        T remove(){assert(curr->next!=NULL && "No element");
                T it=curr->next->e; if(tail==curr->next) tail=curr;
                Link<T>* tmp=curr->next;curr->next=curr->next->next;
                delete tmp;n--;return it;}
        void clear(){removeall();init();}
        void S(){Link<T>* t=head; while(t->next!=curr->next){t=t->next;
                std::cout<<t->e<<' ';} std::cout<<"| "; while(t->next!=NULL){
                t=t->next;std::cout<<t->e<<' ';} std::cout<<'\n';}
};
```

List

- **Linked list implementation**
  - *singly linked list*



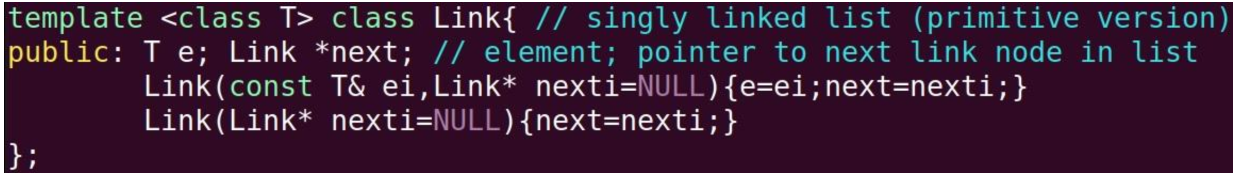
```
g++ demoLList.cpp -o _a; ./_a;
| 5 4 3 2 1
| 6 5 4 3 2 1
6 5 4 3 2 1 |
6 5 4 3 2 1 | 7
6 5 4 3 2 1 | 8 7
6 5 4 | 3 2 1 8 7
6 5 4 | -1 3 2 1 8 7
6 5 4 | -2 -1 3 2 1 8 7
6 5 4 -2 -1 | 3 2 1 8 7
3 removed! 6 5 4 -2 -1 | 2 1 8 7
2 removed! 6 5 4 -2 -1 | 1 8 7
| 6 5 4 -2 -1 1 8 7
6 removed! | 5 4 -2 -1 1 8 7
List currently has 7 elements
| one two
one | two
one | three two
one | four three two
one four | three two
one four | two
three is just removed from list
one four |
two is just removed from list
```



```
#include <iostream>
#include "LList.h"
using namespace std;

template <class T> void ShowL(LList<T>& a){a.S();}

int main(){
        LList<int> ai; for(int i=5;i>0;i--) ai.append(i); ai.S();ai.insert(6);ai.S();
        ai.moveToEnd();ai.S();ai.insert(7);ai.S();ai.insert(8);ai.S();
        ai.moveToPos(3);ai.S();ai.insert(-1);ai.S();ai.insert(-2);ai.S();
        ai.moveToPos(5);ShowL<int>(ai);
        int e=ai.remove();cout<<e<<" removed! ";ai.S();
        e=ai.remove();cout<<e<<" removed! ";ai.S(); ai.moveToStart();ai.S();
        e=ai.remove();cout<<e<<" removed! ";ai.S();
        cout<<"List currently has "<<ai.length()<<" elements\n";
        // ai.moveToEnd();ai.S();ai.remove();

        LList<const char*> as;as.append("one");as.append("two");as.S();
        as.moveToPos(1);as.S();as.insert("three");as.S();as.insert("four");as.S();
        as.moveToPos(2);ShowL<const char*>(as);
        const char* ec=as.remove();as.S();cout<<ec<<" is just removed from list\n";
        ec=as.remove();as.S();cout<<ec<<" is just removed from list\n"; return 0;
}
```

List

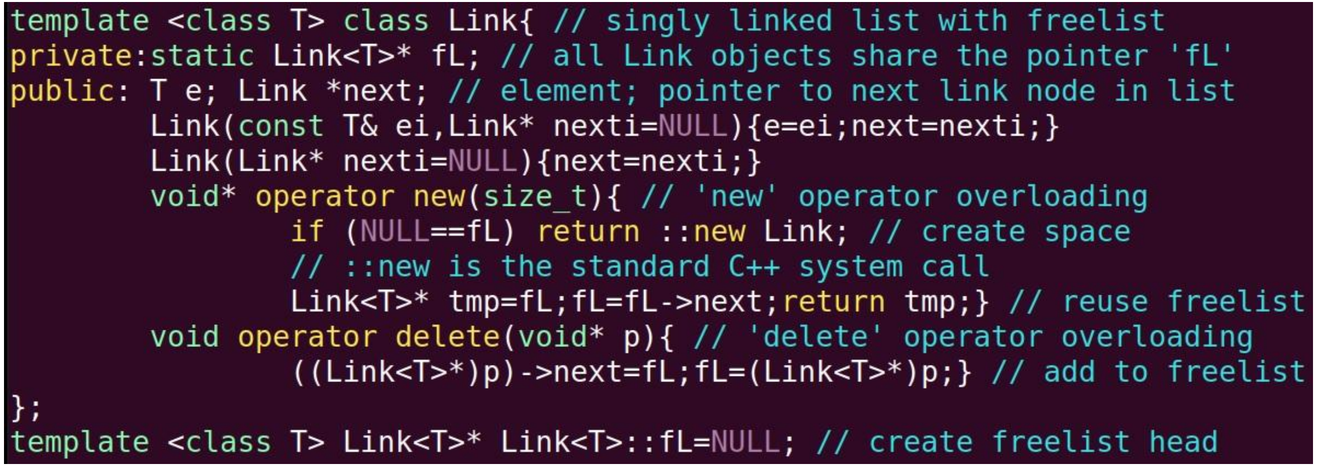
- **free list**
  - take advantage of already allocated space



```
template <class T> class Link{ // singly linked list (primitive version)
public: T e; Link *next; // element; pointer to next link node in list
        Link(const T& ei,Link* nexti=NULL){e=ei;next=nexti;}
        Link(Link* nexti=NULL){next=nexti;}
};
```

singly linked list with freelist

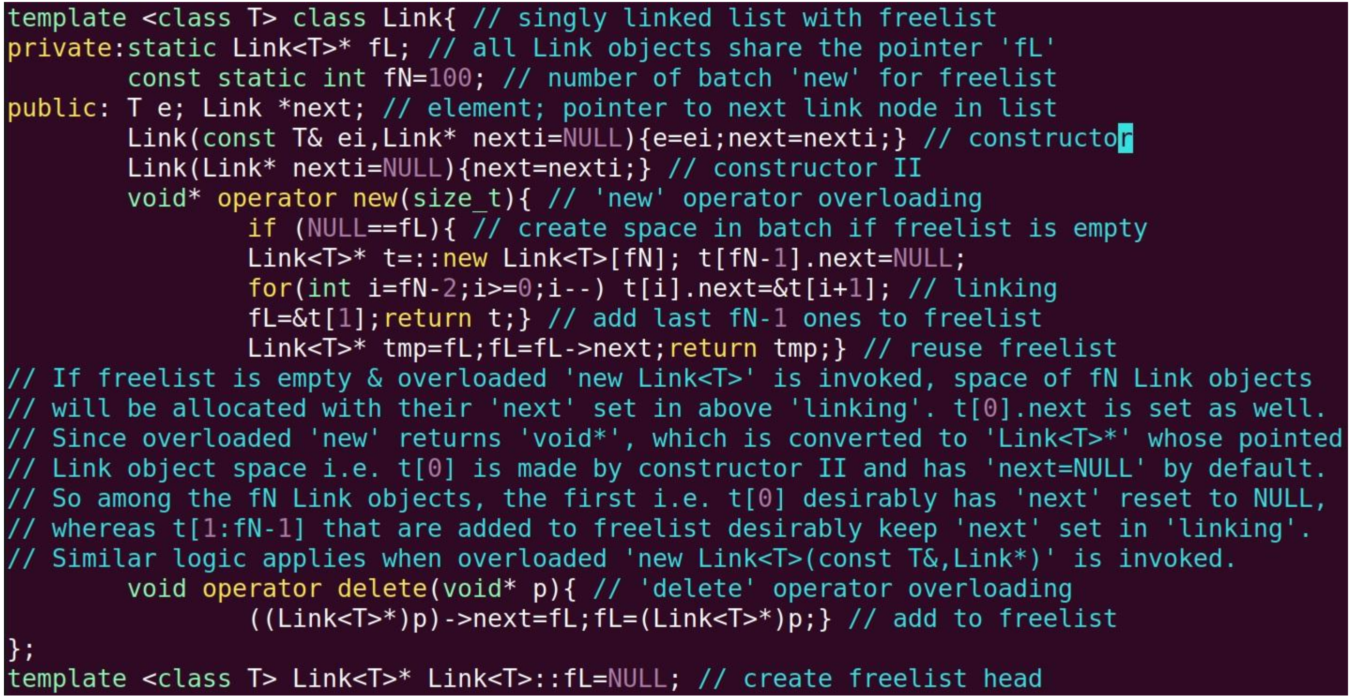
overloaded new via individual ::new



```
template <class T> class Link{ // singly linked list with freelist
private:static Link<T>* fL; // all Link objects share the pointer 'fL'
public: T e; Link *next; // element; pointer to next link node in list
        Link(const T& ei,Link* nexti=NULL){e=ei;next=nexti;}
        Link(Link* nexti=NULL){next=nexti;}
        void* operator new(size_t){ // 'new' operator overloading
                if (NULL==fL) return ::new Link; // create space
                // ::new is the standard C++ system call
                Link<T>* tmp=fL;fL=fL->next;return tmp;} // reuse freelist
        void operator delete(void* p){ // 'delete' operator overloading
                ((Link<T>*)p)->next=fL;fL=(Link<T>*)p;} // add to freelist
};
template <class T> Link<T>* Link<T>::fL=NULL; // create freelist head
```

List

- **free list**
  - take advantage of already allocated space



```
template <class T> class Link{ // singly linked list with freelist
private:static Link<T>* fL; // all Link objects share the pointer 'fL'
        const static int fN=100; // number of batch 'new' for freelist
public: T e; Link *next; // element; pointer to next link node in list
        Link(const T& ei,Link* nexti=NULL){e=ei;next=nexti;} // constructor
        Link(Link* nexti=NULL){next=nexti;} // constructor II
        void* operator new(size_t){ // 'new' operator overloading
                if (NULL==fL){ // create space in batch if freelist is empty
                Link<T>* t=::new Link<T>[fN]; t[fN-1].next=NULL;
                for(int i=fN-2;i>=0;i--) t[i].next=&t[i+1]; // linking
                fL=&t[1];return t;} // add last fN-1 ones to freelist
                Link<T>* tmp=fL;fL=fL->next;return tmp;} // reuse freelist
// If freelist is empty & overloaded 'new Link<T>' is invoked, space of fN Link objects
// will be allocated with their 'next' set in above 'linking'. t[0].next is set as well.
// Since overloaded 'new' returns 'void*', which is converted to 'Link<T>*' whose pointed
// Link object space i.e. t[0] is made by constructor II and has 'next=NULL' by default.
// So among the fN Link objects, the first i.e. t[0] desirably has 'next' reset to NULL,
// whereas t[1:fN-1] that are added to freelist desirably keep 'next' set in 'linking'.
// Similar logic applies when overloaded 'new Link<T>(const T&,Link*)' is invoked.
        void operator delete(void* p){ // 'delete' operator overloading
                ((Link<T>*)p)->next=fL;fL=(Link<T>*)p;} // add to freelist
};
template <class T> Link<T>* Link<T>::fL=NULL; // create freelist head
```

singly linked list with freelist

overloaded new via batch ::new

THANK YOU

上海交通大學

SHANGHAI JIAO TONG UNIVERSITY