Data Structures in C++

Chapter 9

Tim Budd

Oregon State University Corvallis, Oregon USA

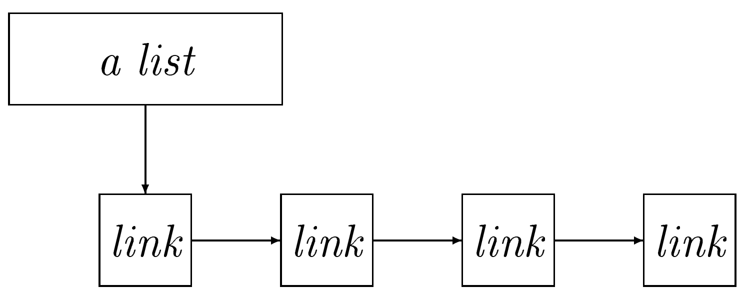
Outline – Chapter 9

Lists

- Concept of linked-list
- Variations
  - Head and Tail Pointers
  - Doubly Linked List
- Summary of List Operations
- Example Programs
  - An Inventory System
  - A Course Registration System
- The implementation of the list data type

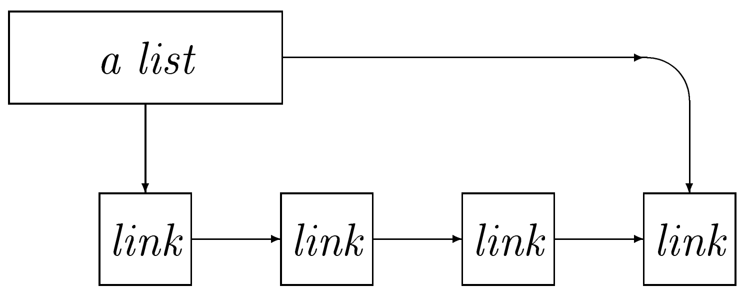
Simple Linked List

- Useful when the number of elements is not known or varies widely during execution
- Allows efficient insertion to head, sequential access.



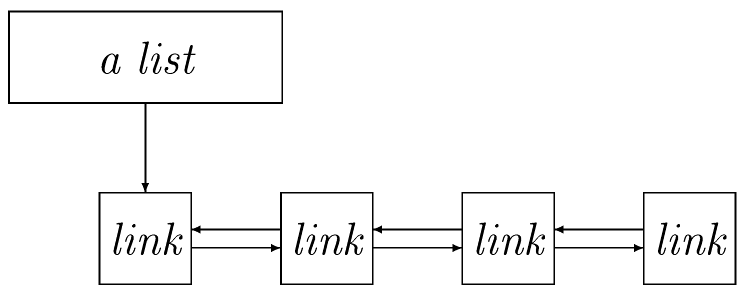
Variation – Head and Tail Pointers

Allows efficient insertion and removal from both head and tail



Variation – Doubly Linked List

Allows movement either forward or backward.



STL list is combination of head and tail pointers and double links.

Summary of List Operations

Constructors and Assignment	
list<T> v;	default constructor
list<T> v (aList);	copy constructor
l = aList	assignment
l.swap (aList)	swap values with another list
Element Access	
l.front ()	first element in list
l.back ()	last element in list
Insertion and Removal	
l.push_front (value)	add value to front of list
l.push_back (value)	add value to end of list
l.insert (iterator, value)	insert value at specified location
l.pop_front ()	remove value from front of list
l.pop_back ()	remove value from end of list
l.erase(iterator)	remove referenced element
l.erase(iterator,iterator)	remove range of elements
l.remove (value)	remove all occurrences of value
l.remove_if (predicate)	removal all values that match condition
Size	
l.empty ()	true if collection is empty
l.size ()	return number of elements in collection
Iterators	
list<T>::iterator itr	declare a new iterator
l.begin ()	starting iterator
l.end ()	ending iterator
Miscellaneous	
l.reverse ()	reverse order of elements
l.sort ()	place elements into ascending order
l.sort (comparison)	order elements using comparison function
l.merge (list)	merge with another ordered list

Declaring a new List

```
list <int> list_one;
    // list of pointers to widgets
list <Widget *> list;
list <Widget> list_three;    // list of widgets

list <int> list_four (list_one);
list <Widget> list_five;
list_five = list_three;

    // exchange values in lists one and four
list_one.swap (list_four);
```

Adding Elements to a List

```
list_one.push_front (12);
list_three.push_back (Widget(6));

        // insert widget 8 at end of list
list_three.insert (list_three.end(), Widget(8));

        // find the location of the
        // first 5 value in list
list<int>::iterator location =
        find (list_one.begin(), list_one.end(), 5);
    // and insert an 11 immediate before it
location = list_one.insert (location, 11);
```

Erasing Elements from a List

```
list_nine.erase (location);

        // erase all values between the first 5
        // and the following 7
list<int>::iterator start =
    find (list_nine.begin(), list_nine.end(), 5);

list<int>::iterator stop =
    find (location, list_nine.end(), 7);

list_nine.erase (start, stop);

list_nine.remove (4); // remove all fours

    //remove elements divisible by 3
list_nine.remove_if (divisableByThree);
```

Number of Elements

```
cout << "Number of elements: "
   << list_nine.size () << endl;

if ( list_nine.empty () )
   cout << "list is empty " << endl;
else
   cout << "list is not empty " << endl;


int num = 0;
count (list_five.begin(), list_five.end(),
      17, num);
if (num > 0)
   cout << "contains a 17" << endl;
else
   cout << "does not contain a 17" <<
endl;
```

Sort, Reverse

```
    // place elements into sequence
list_ten.sort ( );

    // sort using
    // the widget compare function
list_twelve.sort (widgetCompare);

    // elements are now reversed
list_ten.reverse();
```

Insert Iterators

Assignment to an iterator is normally an overwriting operation, replacing the contents of the targeted location.

```
    copy (list_one.begin(), list_one.end(),
list_two.begin());
```

For lists (and sets) often instead want to perform *insertion*. Can be done by creating a list insert iterator.

```
    copy (list_one.begin(), list_one.end(),
back_inserter(list_two));
```

Other insert iterators front_inserter, and inserter.

Adaptors

- An insert iterator is a form of *adaptor*.
- An adaptor changes the interface to an object, but does little or no work itself.
- In this case, change the insert interface into the iterator interface.
- We will later see many different types of adaptors

Example Program–Inventory System

A business, *WorldWideWidgetWorks*, manufactures widgets.

```
class Widget {
public:
      // constructors
   Widget () : id_number (0) { }
   Widget (int a) : id(a) { }

      // operations
   int id () { return id_number; }
   void operator = (Widget & rhs)
      { id_number = rhs.id_number; }
   bool operator == (Widget & rhs)
      { id_number == rhs.id_number; }
   bool operator < (Widget & rhs)
      { id_number < rhs.id_number; }

protected:
   int id_number;   // widget identification number
};
```

Inventory

Inventory is two lists – items on hand, items on back order.

```
//
//    class inventory
//        manage inventory control

class inventory {
public:
        // process order for widget type wid
    void order (int wid);

        // receive widget of type wid
        // in shipment
    void receive (int wid);

protected:
    list <Widget> on_hand;
    list <int> on_order;
};
```

Item Arrives in Shipment

```
void inventory::receive (int wid)
    // process a widget received in shipment
{
    cout <<
        "Received shipment of widget type "
        << wid << endl;

    list<int>::iterator we_need =
            find (on_order.begin(), on_order.end(),
wid);

    if (we_need != on_order.end()) {
        cout << "Ship " << Widget(wid) <<
            " to fill back order" << endl;
        on_order.erase(we_need);
        }
    else
        on_hand.push_front(Widget(wid));
}
```

Item Arrives on Order

```
void inventory::order (int wid)
    // process an order for a widget with given
id number
{
    cout <<
        "Received order for widget type "
            << wid << endl;

    list<Widget>::iterator we_have =
        find_if(on_hand.begin(), on_hand.end(),
WidgetTester(wid));

    if (we_have != on_hand.end()) {
        cout << "Ship " << *wehave << endl;
        on_hand.erase(we_have);
        }
    else {
        cout << "Back order widget of type "
            << wid << endl;
        on_order.push_front(wid);
        }
}
```

Function Objects

Created like an instance of a class, works like a function.

```
class WidgetTester {
public:
    WidgetTester (int id) : test_id(id) { }
    int test_id;

        // define the function call operator
    bool operator () (Widget & wid)
        { return wid.id() == test_id; }
};
```

Course Registration System

Problem: How to assign students to courses, create reports for both instructors and students.

Two input files

```
  ...
ART101 60
HIS213 75
MTH412 35
  ...
```

```
  ...
Smith,Amy ART101
Smith,Amy MTH412
Jones,Randy HIS213
  ...
```

Representation of Classes

```
//
//    class course
//        information about one course

class course {
public:
    course (string n, int s) : name(n), size(s) { }

        // operations
    bool full ()
        { return students.size() >= max; }
    void addStudent (student * s)
        { students.push_back(s); }
    void generateClassList ();


protected:    // data fields
    string name;
    int max;
    list <student *> students;
}
```

Reading the Course File

```
list <course *> all_courses;

void readCourses (istream & infile)
    // read the list of courses from
    // the given input stream
{
    string name;
    int max;

    while (infile >> name >> max) {
        course * newCourse =
            new course (name, max);
        all_courses.push_back (newCourse);
        }
}
```

Representation of Students

```
//
//    class student
//        information about a single student

class student {
    // provide a shorter name for
    // course iterators
typedef list <course *>::iterator citerator;

public:
        // constructor
    student (string n) : nameText(n) { }

        // operations
    void addCourse (course * c) {courses.push_back(c);}
    citerator firstCourse () {return courses.begin();}
    citerator lastCourse () {return courses.end();}
    void removeCourse (citerator & citr)
        {courses.erase(citr);}

protected:
    string nameText;
    list <course *> courses;
};
```

Reading the Student File

```
list <student *> all_students;

void readStudents (istream & infile)
    // read the list of student records
{
    string name;
    string course;

    while (infile >> name >> course) {
        student * theStudent =
            findStudent (name);
        course * theCourse = findCourse (course);
        if (theCourse != 0)
            theStudent.addCourse (theCourse);
        else
            cout << "student " << name <<
                " requested invalid course "
                << course << "\n";
        }
}
```

Finding the Student in the list of Students

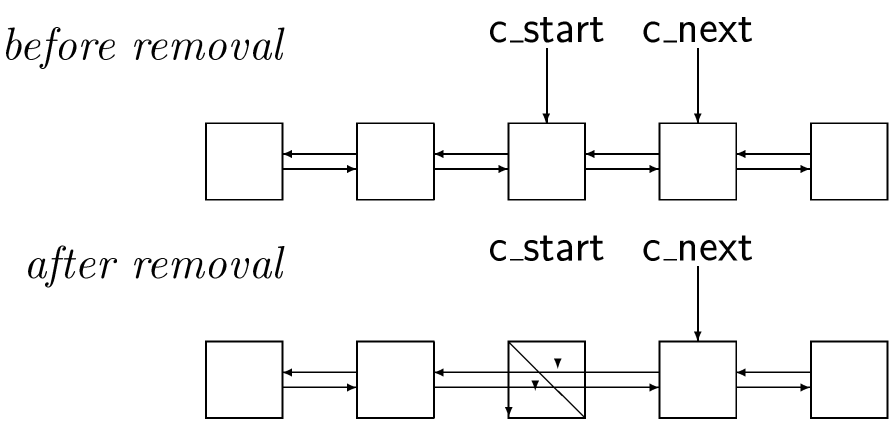
```
student * findStudent (string & searchName)
    // find (or make) a student record
    // for the given name
{
    list <student *>::iterator start, stop;
    start = all_students.begin();
    stop = all_students.end();
    for ( ; start != stop; ++start)
        if ((*start)->name() == searchName)
            return *start;

        // not found, make one now
    student * newStudent =
        new student(searchName);
    all_students.push_back (newStudent);
    return newStudent;
}
```

Merging the Two Lists

```
void fillCourses ()
    // fill students as possible in each course
{
    list<student *>::iterator s_start, s_end;
    s_start = all_students.begin();
    s_end = all_students.end();
    for ( ; s_start != s_end; ++s_start) {
        list<course *>::iterator c_start, c_end;
        list<course *>::iterator c_next;
        c_start = (*s_start)→firstCourse();
        c_end = (*s_start)→lastCourse();
        for ( ; c_start != c_end; c_start = c_next) {
            c_next = c_start; ++c_next;
                // if not full, add student
            if (! (*c_start)→full())
                (*c_start)→addStudent (*s_start);
            else
                (*s_start)→removeCourse(c_start);
            }
    } }
```

Removal

Note carefully how the removal is handled. It is not legal to use a list iterator once it has been removed from the list. So we must get the next element before doing the removal.



Generate Class List

```
void course::generateClassList ()
    // print the class list of all students
{
        // first sort the list
    students.sort (studentCompare);

        // then print it out
    cout << "Class list for "
        << name << "\n";
    list<student *>::iterator start, stop;
    start = students.begin();
    stop = students.end();
    for ( ; start != stop; ++start)
        cout << (*start)->name() << "\n";
}
```

An Example Implementation

```
template <class T>
class list {
public:
        // type definitions
    typedef T value_type;
    typedef listIterator<T> iterator;

        // constructors
    list () : firstLink(0), lastLink(0) { }

        // operations
    bool empty () { return firstLink == 0; }
    int size();
    T & back () { return lastLink→value; }
    T & front () { return firstLink→value; }
    void pop_front ();
    void pop_back ();
    iterator begin () { return iterator (this, firstLink); }
    iterator end () { return iterator (this, 0); }
    void sort ();
    iterator insert (iterator, value);
    void erase (iterator & itr) { erase (itr, itr); }
    void erase (iterator &, iterator &);
```

```
protected:
    link <T> * firstLink;
    link <T> * lastLink;
};
```

Link, a facilitator class working behind the scenes

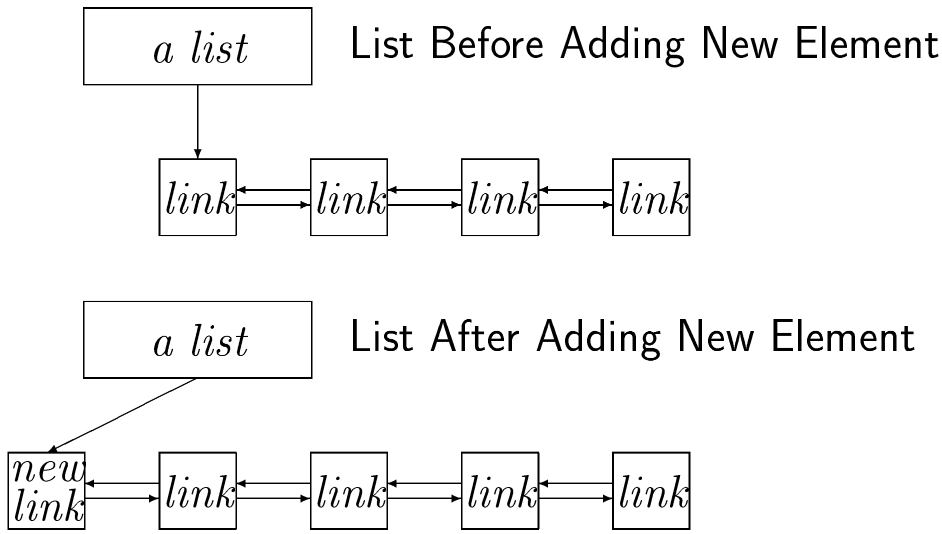
```
template <class T> class link {
public:
    link (T & v)
        : value(v), nextLink(0), prevLink(0) { }
    T value;
    link<T> * nextLink;
    link<T> * prevLink;
};
```

Walking Down the List

Internal operations that access the entire structure simply walk down the list.

```
template <class T> int list<T>::size ()
    // count number of elements in collection
{
    int counter = 0;
    link<T> * ptr = firstLink;
    for ( ; ptr != 0; ptr = ptr→nextLink)
        counter++;
    return counter;
}
```

Adding a New Element to Front of List



Must Check for Empty List

```
template <class T> void list<T>::push_front
        (T & newValue)
    // add a new value to the front of a list
{
    link<T> * newLink =
        new link<T> (newValue);

    if (empty())
        firstLink = lastLink = newLink;
    else {
        firstLink→prevLink = newLink;
        newLink→nextLink = firstLink;
        firstLink = newLink
        }
}
```

Removal From List

```
template <class T> void list<T>::pop_front()
    // remove first element from linked list
{
    link <T> * save = firstLink;
    firstLink = firstLink→nextLink;
    if (firstLink != 0)
        firstLink→prevLink = 0;
    else
        lastLink = 0;
    delete save;
}
```

Note how removing last element from list is handled as special case.

List Iterators

List iterators must look like pointers, but act differently.

Can be created as special class that keeps an internal pointer to the list, and to the current link.

Iterator Class

```
template <class T> class listIterator {
   typedef listIterator<T> iterator;
public:
      // constructor
   listIterator (list<T> * tl, link<T> * cl)
      : theList(tl), currentLink (cl) { }

      // iterator protocol
   T & operator * ()
      { return currentLink→value; }
   void operator = (iterator & rhs)
      { theList = rhs.theList;
      currentLink = rhs.currentLink; }
   bool operator == (iterator & rhs)
      { return currentLink == rhs.currentLink; }
   iterator & operator ++ (int)
      { currentLink = currentLink→nextLink;
      return this; }
   iterator operator ++ ();
   iterator & operator −− (int)
      { currentLink = currentLink→prevLink;
      return this; }
   iterator operator −− ();

protected:
   list <T> * theList; link <T> * currentLink; };
```

Postorder Decrement is More Complex

```
template <class T> listIterator<T>
listIterator<T>::operator ++ ()
    // postfix form of increment
{
        // clone, then increment, return clone
    listIterator<T> clone (theList, currentLink);
    currentLink = currentLink→nextLink;
    return clone;
}
```

Insertion At an Iterator Position

```
template <class T> void list<T>::insert
(listIterator<T> & itr, T & value)
    // insert a new element into the
    // middle of a linked list
{
    link<T> * newLink = new link(value);
    link<T> * current = itr->currentLink;

    newLink->nextLink = current;
    newLink->prevLink = current->prevLink;
    current->prevLink = newLink;
    current = newLink->prevLink;
    if (current != 0)
        current->nextLink = newLink;
}
```

Removal of an Iterator Range

```
template <class T>
void list<T>::erase (listIterator<T> & start,
listIterator<T> & stop)
    // remove range of elements
{
    link<T> * first = start→currentLink;
    link<T> * prev = first→prevLink;
    link<T> * last = stop→currentLink;
    if (prev == 0) {      // removing initial
        firstLink = last;
        if (last == 0)
            lastLink = 0;
        else
            last→prevLink = 0;
        }
    else {
        prev→nextLink = last;
        if (last == 0)
            lastLink = prev;
        else
            last→prevLink = prev;
        }
        // now delete the values
```

```
    while (start != stop) {
        link<T> * next = start;
        ++next;
        delete start;
        start = next;
        }
}
```

Variation through Inheritance

Inheritance is a powerful technique for creating new classes out of existing classes.

Basically, you simply say that the new thing is an extension of the old thing. All data fields, member functions and the like from the old abstraction are available for free in the new class.

```
class newClass : public OldClass {
    ...
}
```

Example, Ordered Lists

```
template <class T>
class orderedList : public list<T> {
public:
    void add (T & newValue);
};
```

The only thing we need add is a new method for adding elements to the list.
All other member functions associated with list are still available, for free, with instances of this new class.

Adding the New Value

Here is how a new value gets added to an ordered list.

```
template <class T>
void orderedList<T>::add (T & newValue)
    // add a new element to an ordered list
{
    list<T>::iterator start, stop;
    start = begin();
    stop = end();
    while ((start != stop) &&
        (*start < newValue))
            ++start;
    insert (start, newValue);
}
```

Application, List Insertion Sort

```
template <class T>
void listInsertionSort (vector<T> & v)
    // place a vector into order,
    // using an ordered list
{
    orderedList<T> sorter;

        // first copy vector to list
    vector<T>::iterator start = v.begin();
    vector<T>::iterator stop = v.end();
    for ( ; start != stop; ++start)
        sorter.add(*start);

        // then copy list back to vector
    list<T>::iterator itr = sorter.begin();
    for (start = v.begin(); start != stop; ++start)
        *start = *itr++;
}
```

Self Organizing Lists

```
template <class T>
class selfOrganizingList<T> : public list<T> {
public:
    bool include (T & value);
};
```

Includes test for SO list

```
template <class T>
bool selfOrganizingList<T>::include (T & value)
    // see if argument value occurs in list
{
        // first find element in list
    list<T>::iterator stop = end();
    list<T>::iterator where =
        find(begin(), stop, value);
        // if not found, return false
    if (where == stop)
        return false;
        // else remove from list,
        // and move to front
    if (where != begin()) {
        erase (where);
        push_front (value);
        }
    return true;
}
```

Private, Protected and Public

With inheritance, there are three levels of protection:

- public, accessible to the world
- protected, accessible to class and subclasses
- private, accessible only to class (not even to subclasses!)