

```
// array1.cpp
```

```
// shows days from start of year to date specified
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int month, day, totalDays;
```

```
    int daysPerMonth[12] = { 31, 28, 31, 30, 31, 30,  
                             31, 31, 30, 31, 30, 31 };
```

```
    cout << "\nEnter month (1 to 12): "; // get date
```

```
    cin >> month;
```

```
    cout << "Enter day (1 to 31): ";
```

```
    cin >> day;
```

```
    totalDays = day; // separate days
```

```
    for(int index=0; index<month-1; index++) // add days each month
```

```
        totalDays += daysPerMonth[index];
```

```
    cout << "Total days from start of year is: " << totalDays;
```

```
}
```

```
output
```

```
Enter month (1 to 12): 12
```

```
Enter day (1 to 31): 24
```

```
Total days from start of year is: 358
```

```

// array2.cpp
// illustrates array operations
#include <iostream>
using namespace std;
#define maximumCells 5
    void printArray(int array[], int numberOfCells) ;
    int main()
    {
        int firstArray[maximumCells];
        int secondArray[maximumCells];
        int index;

//to input array elements

//      cout << firstArray ; invalid (no aggregate operations)

        cout << "Please enter 5 numbers" << endl;
        for (index=0;index<maximumCells;index++)
            cin >> firstArray[index];
        printArray(firstArray,maximumCells);

// to copy arrays
//      firstArray=secondArray;  invalid (no aggregate operations)

        for (index=0;index<maximumCells;index++)
            secondArray[index]=firstArray[index];
        printArray(secondArray,maximumCells);

// to add array elements
//      firstArray=firstArray+secondArray;  invalid (no aggregate operations)

        for (index=0;index<maximumCells;index++)
            firstArray[index]=firstArray[index]+ secondArray[index];
        printArray(firstArray,maximumCells);
    }

    void printArray(int array[],int numberOfCells)
    {
        int index;
        cout << "The current array:" << endl;
        for (index=0;index < numberOfCells ;index++)
            cout << array[index] << endl;
    }

```

output

Please enter 5 numbers

2 4 6 8 10

The current array:

2

4

6

8

10

The current array:

2

4

6

8

10

The current array:

4

8

12

16

20

// array3.cpp

// Aggregate C String I/O in C++

#include <iostream>

using namespace std;

int main()

{

    char message [ 80 ] ;

    cin >> message ;

    cout << message << endl; // only valid with strings

    int index=0;

    while (message[index] != '\0')

        cout << message[index++];

}

output

this is a string

this

this

// array4.cpp

// Aggregate C String I/O in C++

```
#include <iostream>
using namespace std;
int main()
{
    char message [ 80 ];
    cin >> message ;
    cout << message << endl; // only valid with strings
    cout << "Aggregate C String I/O in C++" << endl;
    int index=0;
    while (message[index] != '\0')
        cout << message[index++];
}
```

output

this is a string

this

Aggregate C String I/O in C++

this

// array5.cpp

// displays sales chart, initializes 2-d array

```
#include <iostream>
using namespace std;
```

```
const int DISTRICTS = 4;    // array dimensions
```

```
const int MONTHS = 3;
```

```
int main()
```

```
{
```

```
    int district, month;
```

```
                                // initialize array elements
```

```
    float sales[DISTRICTS][MONTHS]
```

```
        = { { 1432.07, 234.50, 654.01 },
              { 322.00, 13838.32, 17589.88 },
              { 9328.34, 934.00, 4492.30 },
              { 12838.29, 2332.63, 32.93 } };
```

```
    for(district=0; district<DISTRICTS; district++)
```

```
    {
```

```
        cout << "\nDistrict " << district+1 << endl;
```

```
        for(month=0; month<MONTHS; month++)
```

```
            cout << sales[district][month] << endl; // access array element
```

```
    }
```

```
    return 0;
}
```

output

District 1  
1432.07  
234.5  
654.01

District 2  
322  
13838.3  
17589.9

District 3  
9328.34  
934  
4492.3

District 4  
12838.3  
2332.63  
32.93

```
// array6.cpp
```

```
#include <iostream>
using namespace std;
```

```
void printArray( int rows, int columns, int array [][][5] );
```

```
int main()
```

```
{
```

```
    int twoDimArray[4][5],
        row, column;
```

```
    for ( row = 0; row < 4; row++ )
```

```
        for ( column = 0; column < 5; column++ )
            twoDimArray[row][column] = row * column;
```

```
    printArray( 4, 5, twoDimArray );
```

```
}
```

```
/** ***** printArray() ***** */
```

```
// An output routine. Displays the contents of an array of
// type int. The array is passed as a parameter along with
```

// the number of rows and columns to be displayed.

```
void printArray( int rows, int columns, int array[][5] )
{
    int i = 0, j = 0;

    while ( i < rows )
    {
        cout << array[i][j];

        ( j == columns-1 ) ? cout << '\n' : cout << '\t' ;

        ( j == columns - 1 ) ? i++,j=0 : j++;
    }
}
```

output

0	0	0	0	0
0	1	2	3	4
0	2	4	6	8
0	3	6	9	12

```
// array7.cpp
```

```
// illustrates passing two dimensional arrays
```

```
//
```

```
// Objective - Demonstrates multidimensional arrays and
```

```
// the?: construct.
```

```
//
```

```
// Illustrates using three-dimensional arrays
```

```
#include <iostream>
```

```
#include <iomanip>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    double triple[2][3][4] = {
```

```
        { { 0.0, 0.1, 0.2, 0.3 },
```

```
          { 1.0, 1.1, 1.2, 1.3 },
```

```
          { 2.0, 2.1, 2.2, 2.3 } 
```

```
        },
```

```
        { { 10.0, 10.1, 10.2, 10.3 },
```

```
          { 11.0, 11.1, 11.2, 11.3 },
```

```
          { 12.0, 12.1, 12.2, 12.3 } 
```

```
        } 
```

```
    };
```

```
    int index;
```

```
    int jindex;
```

```
    int kindex;
```

```
    cout << fixed << showpoint << setprecision(1) ;
```

```
    for ( index = 0; index < 2; index++ )
```

```
    {
```

```
        for ( jindex = 0; jindex < 3; jindex++ )
```

```
        {
```

```
            for ( kindex = 0; kindex < 4; kindex++ )
```

```
            {
```

```
                cout << setw(5) << triple[index][jindex][kindex];
```

```
            }
```

```
            cout << endl;
```

```
        }
```

```
        cout << endl << endl;
```

```
    }
```

```
}
```

### output

```
0.0 0.1 0.2 0.3
1.0 1.1 1.2 1.3
2.0 2.1 2.2 2.3
```

```
10.0 10.1 10.2 10.3
11.0 11.1 11.2 11.3
12.0 12.1 12.2 12.3
```

### // VECTOR 1.cpp

```
// comparing size, capacity and max_size
#include <iostream>
#include <vector>

int main ()
{
    std::vector<int> myvector;

    std::cout << "capacity: " << myvector.capacity() << "\n";
    // set some content in the vector:
    for (int i=0; i<100; i++) myvector.push_back(i);

    std::cout << "size: " << myvector.size() << "\n";
    std::cout << "capacity: " << myvector.capacity() << "\n";
    std::cout << "max_size: " << myvector.max_size() << "\n";
    return 0;
}
```

### Output

```
capacity: 0
size: 100
capacity: 128
max_size: 1073741823
```

### // VECTOR 2.cpp

```
// inserting into a vector
// inserting into a vector
//The vector is extended by inserting new elements before the element at the
//specified position, effectively increasing the container size by the number of
//elements inserted.
```

```
#include <iostream>
#include <vector>
```

```
int main ()
```



```

{
    std::vector<int> myvector (3,100);
    std::vector<int>::iterator it;

    //print out the vector
    std::cout << "myvector contains:";
    for (it=myvector.begin(); it<myvector.end(); it++)
        std::cout << ' ' << *it;
    std::cout << '\n';

    it = myvector.begin();
    it = myvector.insert ( it , 200 );

    //print out the vector
    std::cout << "myvector contains:";
    for (it=myvector.begin(); it<myvector.end(); it++)
        std::cout << ' ' << *it;
    std::cout << '\n';

    myvector.insert (it,2,300);

    //print out the vector
    std::cout << "myvector contains:";
    for (it=myvector.begin(); it<myvector.end(); it++)
        std::cout << ' ' << *it;
    std::cout << '\n';

    it = myvector.begin();

    std::vector<int> anothervector (2,400);
    myvector.insert (it+2,anothervector.begin(),anothervector.end());

    //print out the vector
    std::cout << "myvector contains:";
    for (it=myvector.begin(); it<myvector.end(); it++)
        std::cout << ' ' << *it;
    std::cout << '\n';

    int myarray [] = { 501,502,503 };
    myvector.insert (myvector.begin(), myarray, myarray+3);

    std::cout << "myvector contains:";
    for (it=myvector.begin(); it<myvector.end(); it++)
        std::cout << ' ' << *it;
    std::cout << '\n';

```

```
    return 0;  
}
```

#### Output

myvector contains: 100 100 100

myvector contains: 200 100 100 100

myvector contains: 200 100 100 100 300 300

myvector contains: 200 100 400 400 100 100 300 300

myvector contains: 501 502 503 200 100 400 400 100 100 300 300

**// linkedList1**

```
#ifndef LINKEDLIST_H_  
#define LINKEDLIST_H_
```

```
// to illustrate using a linked list with classes (header file)  
//
```

```
    #include <iostream>  
using namespace std;
```

```
// SPECIFICATION FILE DYNAMIC-LINKED SORTED LIST
```

```
typedef int ItemType ; // Type of each component  
                        // is simple type or string type
```

```
struct NodeType  
{  
    ItemType    item ;      // data  
    NodeType*   link ;      // link to next node in list  
};
```

```
typedef NodeType* NodePtr;
```

```
// SPECIFICATION FILE DYNAMIC-LINKED SORTED LIST
```

```
class SortedList  
{  
public :
```

```
    bool    IsEmpty ( ) const ;
```

```
    void    Print ( ) const ;
```

```
    void    InsertTop ( /* in */ ItemType item ) ;
```

```
    void    Insert ( /* in */ ItemType item ) ;
```

```
    void    DeleteTop ( /* out */ ItemType& item ) ;
```

```
    void    Delete ( /* in */ ItemType item ) ;
```

```
    SortedList ( ) ;                // Constructor
```

```

        ~SortedList ( ) ;                               // Destructor
        SortedList ( const SortedList& otherList ) ; // Copy-constructor

private :

        NodeType* head;
};

#endif /* LINKEDLIST_H_ */
.....

#include "linkedList.h"

// IMPLEMENTATION DYNAMIC-LINKED SORTED LIST
SortedList ::SortedList ( ) // Constructor
// Post: head == NULL
{
        head = NULL ;
}

SortedList :: ~SortedList ( ) // Destructor
// Post: All linked nodes deallocated
{
        ItemType temp ;
                                // keep deleting top node
        while ( !IsEmpty ( ) )
                DeleteTop ( temp );
}

bool SortedList ::IsEmpty ( ) const
// Postcondition
// function value == true, if head == NULL
// == false, otherwise
{
        return (head==NULL);
}

void SortedList ::Print ( ) const
// print out link list
{
        NodePtr currPtr ;
        currPtr = head ; // point to the beginning of the list

        while (currPtr != NULL)
        {

```

```

        cout << currPtr->item << endl;
        currPtr = currPtr->link ; // point to the next component
    }
}

```

```

void SortedList :: Insert( /* in */ ItemType item )
// Pre:      item is assigned && list components in ascending order
// Post:      new node containing item is in its proper place
//            && list components in ascending order
{
    NodePtr currPtr ;
    NodePtr prevPtr ;
    NodePtr newNodePtr ;
    newNodePtr = new NodeType ;
    newNodePtr->item = item ;
    prevPtr = NULL ;
    currPtr = head ;
    while ( currPtr != NULL && item > currPtr->item )
    {
        prevPtr = currPtr ;           // advance both pointers
        currPtr = currPtr->link ;
    }
    newNodePtr->link = currPtr ;       // insert new node here
    if ( prevPtr == NULL )
        head =newNodePtr ;
    else
        prevPtr->link = newNodePtr;
}

```

```

void SortedList :: DeleteTop ( /* out */ ItemType& item )
// Pre:      list is not empty && list elements in ascending order
// Post:      item == element of first list node @ entry
//            && node containing item is no longer in linked list
//            && list elements in ascending order
{
    NodePtr tempPtr = head ;
                                // obtain item and advance head
    item = head->item;
    head = head->link ;
    delete tempPtr ;
}

```

```

void SortedList :: Delete ( /* in */ ItemType item )
// Pre:      list is not empty && list elements in ascending order
//            && item == component member of some list node
// Post:      item == element of first list node @ entry

```

```

//      && node containing first occurrence of item is no longer
//      in linked list && list elements in ascending order
{   NodePtr delPtr ;
    NodePtr currPtr ;           // Is item in first node?
    if ( item == head->item )
    {   delPtr = head ;           // If so, delete first node
        head = head->link ;
    }
    else {                       // search for item in rest of list
        currPtr = head ;
        while ( currPtr->link->item != item )
            currPtr = currPtr->link ;
        delPtr = currPtr->link ;
        currPtr->link = currPtr->link->link ;
    }
    delete delPtr ;
}

int main()
{

    SortedList list;
    ItemType mainItem;

    list.Insert(352);
    list.Insert(48);
    list.Insert(12);
    list.Print();

    if (!list.IsEmpty())
    {
        list.DeleteTop(mainItem); // delete the first node
        cout << "node delete was " << mainItem << endl << endl;
    }

    cout << "\nprint out list after delete" << endl;
    list.Print();

    list.Insert(1); // insert at the top of the list
    list.Insert(500); //insert at the bottom of the list
    list.Insert(77); // insert in the middle
    cout << "\nprint the list after inserting nodes" << endl;
    list.Print();

    list.Delete(48); // delete in the middle
    cout << "\nprint the list deleting a middle node" << endl;
    list.Print();
}

```

```

        list.Delete(1); // delete the first node
        cout << "\nprint the list deleting the first node" << endl;
        list.Print();

        list.Delete(500); // delete the last node
        cout << "\nprint the list deleting the last node" << endl;
        list.Print();
    }

```

### Output

```

12
48
352
node delete was 12

```

### // linkedList2

```

#ifndef LINKEDLIST_H_
#define LINKEDLIST_H_

```

```

#include <iostream>
using namespace std;

```

//Definition of the node

```

template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *link;
};

```

```

template<class Type>
class linkedListType
{
public:
    const linkedListType<Type>& operator=
        (const linkedListType<Type>&);
    //Overload the assignment operator
    void initializeList();
    //Initialize the list to an empty state
    //Post: first = NULL, last = NULL
    bool isEmptyList();
    //Function returns true if the list is empty;
    //otherwise, it returns false

```

```

void print();
    //Output the data contained in each node
    //Pre: List must exist
    //Post: None
int length();
    //Return the number of elements in the list
void destroyList();
    //Delete all nodes from the list
    //Post: first = NULL, last = NULL
void retrieveFirst(Type& firstElement);
    //Return the info contained in the first node of the list
    //Post: firstElement = first element of the list
void search(const Type& searchItem);
    //Outputs "Item is found in the list" if searchItem is in
    //the list; otherwise, outputs "Item is not in the list"
void insertFirst(const Type& newItem);
    //newItem is inserted in the list
    //Post: first points to the new list and the
    //      newItem inserted at the beginning of the list
void insertLast(const Type& newItem);
    //newItem is inserted in the list
    //Post: first points to the new list and the
    //      newItem is inserted at the end of the list
    //      last points to the last node in the list
void deleteNode(const Type& deleteItem);
    //if found, the node containing deleteItem is deleted
    //from the list
    //Post: first points to the first node and
    //      last points to the last node of the updated list
linkedListType();
    //default constructor
    //Initializes the list to an empty state
    //Post: first = NULL, last = NULL
linkedListType(const linkedListType<Type>& otherList);
    //copy constructor
~linkedListType();
    //destructor
    //Deletes all nodes from the list
    //Post: list object is destroyed

protected:
    nodeType<Type> *first; //pointer to the first node of the list
    nodeType<Type> *last;  //pointer to the last node of the list
};

#endif /* LINKEDLIST_H_ */

```



.....

```
#include "linkedlist.h"
```

```
template<class Type>
bool linkedListType<Type>::isEmptyList()
{
    return(first == NULL);
}
```

```
template<class Type>
linkedListType<Type>::linkedListType() // default constructor
{
    first = NULL;
    last = NULL;
}
```

```
template<class Type>
void linkedListType<Type>::destroyList()
{
    nodeType<Type> *temp; //pointer to deallocate the memory
                          //occupied by the node
    while(first != NULL) //while there are nodes in the list
    {
        temp = first; //set temp to the current node
        first = first->link; //advance first to the next node
        delete temp; //deallocate memory occupied by temp
    }
    last = NULL; //initialize last to NULL; first has already
                //been set to NULL by the while loop
}
```

```
template<class Type>
void linkedListType<Type>::initializeList()
{
    destroyList(); //if the list has any nodes, delete them
}
```

```
template<class Type>
void linkedListType<Type>::print()
{
    nodeType<Type> *current; //pointer to traverse the list
```

```

    current = first; //set current so that it points to
                      //the first node
    while(current != NULL) //while more data to print
    {
        cout<<current->info<<" ";
        current = current->link;
    }
} //end print

```

```

template<class Type>
int linkedListType<Type>::length()
{
    int count = 0;
    nodeType<Type> *current; //pointer to traverse the list

    current = first;

    while (current!= NULL)
    {
        count++;
        current = current->link;
    }

    return count;
} // end length

```

```

template<class Type>
void linkedListType<Type>::retrieveFirst(Type& firstElement)
{
    firstElement = first->info; //copy the info of the first node
} //end retrieveFirst

```

```

template<class Type>
void linkedListType<Type>::search(const Type& item)
{
    nodeType<Type> *current; //pointer to traverse the list
    bool found;

    if(first == NULL) //list is empty
        cout<<"Cannot search an empty list. "<<endl;
    else
    {

```

```

        current = first; //set current pointing to the first
                           //node in the list

        found = false; //set found to false

        while(!found && current != NULL) //search the list
            if(current->info == item) //item is found
                found = true;
            else
                current = current->link; //make current point to
                                         //the next node

        if(found)
            cout<<"Item is found in the list."<<endl;
        else
            cout<<"Item is not in the list."<<endl;
    } //end else
} //end search

template<class Type>
void linkedListType<Type>::insertFirst(const Type& newItem)
{
    nodeType<Type> *newNode; //pointer to create the new node

    newNode = new nodeType<Type>; //create the new node
    newNode->info = newItem; //store the new item in the node
    newNode->link = first; //insert newNode before first
    first = newNode; //make first point to the
                     //actual first node

    if(last == NULL) //if the list was empty, newNode is also
                     //the last node in the list
        last = newNode;
}

template<class Type>
void linkedListType<Type>::insertLast(const Type& newItem)
{
    nodeType<Type> *newNode; //pointer to create the new node

    newNode = new nodeType<Type>; //create the new node
    newNode->info = newItem; //store the new item in the node
    newNode->link = NULL; //set the link field of new node
                          //to NULL
}

```

```

    if(first == NULL) //if the list is empty, newNode is
                      //both the first and last node
    {
        first = newNode;
        last = newNode;
    }
    else //if the list is not empty, insert newNode after last
    {
        last->link = newNode; //insert newNode after last
        last = newNode; //make last point to the actual last node
    }
} //end insertLast

```

```

template<class Type>
void linkedListType<Type>::deleteNode(const Type& deleteltem)
{
    nodeType<Type> *current; //pointer to traverse the list
    nodeType<Type> *trailCurrent; //pointer just before current
    bool found;

    if(first == NULL) //Case 1; list is empty.
        cout<<"Can not delete from an empty list.\n";
    else
    {
        if(first->info == deleteltem) //Case 2
        {
            current = first;
            first = first ->link;
            if(first == NULL) //list had only one node
                last = NULL;
            delete current;
        }
        else //search the list for the node with the given info
        {
            found = false;
            trailCurrent = first; //set trailCurrent to point to
                                //the first node
            current = first->link; //set current to point to the
                                //second node

            while((!found) && (current != NULL))
            {
                if(current->info != deleteltem)
                {
                    trailCurrent = current;

```

```

        current = current-> link;
    }
    else
        found = true;
} // end while

if(found) //Case 3; if found, delete the node
{
    trailCurrent->link = current->link;

    if(last == current)    //node to be deleted was
        //the last node
    last = trailCurrent;    //update the value of last
    delete current;        //delete the node from the list
}
else
    cout<<"Item to be deleted is not in the list."<<endl;
} //end else
} //end else
} //end deleteNode

```

```

template<class Type>
LinkedListType<Type>::~LinkedListType() // destructor
{
    NodeType<Type> *temp;

    while(first != NULL) //while there are nodes left in the list
    {
        temp = first;    //set temp point to the current node
        first = first->link; //advance first to the next node
        delete temp;    //deallocate memory occupied by temp
    } //end while

    last = NULL; //initialize last to NULL; first is already null
} //end destructor

//copy constructor
template<class Type>
LinkedListType<Type>::LinkedListType(
    const LinkedListType<Type>& otherList)
{
    NodeType<Type> *newNode; //pointer to create a node
    NodeType<Type> *current; //pointer to traverse the list

    if(otherList.first == NULL) //otherList is empty

```

```

{
    first = NULL;
    last = NULL;
}
else
{
    current = otherList.first; //current points to the
                                //list to be copied

    //copy the first node
    first = new nodeType<Type>; //create the node
    first->info = current->info; //copy the info
    first->link = NULL;         //set the link field of
                                //the node to NULL
    last = first;               //make last point to the
                                //first node
    current = current->link;     //make current point to the
                                //next node

    //copy the remaining list
    while(current != NULL)
    {
        newNode = new nodeType<Type>; //create a node
        newNode->info = current->info; //copy the info
        newNode->link = NULL;         //set the link of
                                //newNode
        to NULL

        last->link = newNode;         //attach newNode after last
        last = newNode;               //make last point to
                                //the actual last
        node

        current = current->link; //make current point to
                                //the next node

    } //end while
    } //end else
} //end copy constructor

//overload the assignment operator
template<class Type>
const linkedListType<Type>& linkedListType<Type>::operator=(
    const linkedListType<Type>&
otherList)
{
    nodeType<Type> *newNode; //pointer to create a node
    nodeType<Type> *current; //pointer to traverse the list.

```

```

if(this != &otherList) //avoid self-copy
{
    if(first != NULL) //if the list is not empty, destroy the list
        destroyList();

    if(otherList.first == NULL) //otherList is empty
    {
        first = NULL;
        last = NULL;
    }
    else
    {
        current = otherList.first;    //current points to the
                                        //list to be copied

        //copy the first element
        first = new nodeType<Type>;    //create the node
        first->info = current->info;    //copy the info
        first->link = NULL;            //set the link field of
                                        //the node
        to NULL

        last = first;    //make last point to the first node
        current = current->link; //make current point to the next
                                //node of the list being
        copied

        //copy the remaining list
        while(current != NULL)
        {
            newNode = new nodeType<Type>;
            newNode->info = current->info;
            newNode->link = NULL;
            last->link = newNode;
            last = newNode;
            current = current->link;
        } //end while
    } //end else
} //end else

return *this;
}

int main()
{

```

```

2      linkedListType<int> list1, list2;           //Line 1
      int num;                                     //Line

      cout<<"Line 3: Enter numbers ending with -999"
          <<endl;
      //Line 3
      cin>>num;                                     //Line

4

      while(num != -999)                           //Line 5
      {
          list1.insertLast(num);                   //Line 6
          cin>>num;                                 //Line

7      }

      cout<<endl;                                   //Line

8

      cout<<"list 1: ";                             //Line 9
      list1.print();                               //Line 10
      cout<<endl;                                   //Line

11     cout<<"list1 length is " << list1.length() <<endl;

      list2 = list1;    //test the assignment operator Line 13

      cout<<"list 2: ";
      list2.print();
      cout<<endl;                                   //Line 15

16     cout<<"list 2 length is " <<list2.length() <<endl;

      cout << "All the nodes in list 2 are destroyed" << endl;
      list2.initializeList(); // destroy nodes in list 2

      if (list2.isEmptyList())
          cout << "It has been verified that list2 is empty" << endl;

      int firstInt;
      list1.retrieveFirst(firstInt);
      cout <<"The first node in list 1 is " << firstInt << endl;

      int searchInt;
      cout << "Enter a number to search" << endl;

```



```

        cin >> searchInt;
        list1.search(searchInt);

        cout << "Enter another number to search" << endl;
        cin >> searchInt;
        list1.search(searchInt);

        cout << "Enter a number to add at the begining of the list" << endl;
        cin >> firstInt;
        list1.insertFirst(firstInt);

        int endInt;
        cout << "Enter a number to add at the end of the list" << endl;
        cin >> endInt;
        list1.insertLast(endInt);

        int deleteInt;
        cout << "Enter a number to delete from the list" << endl;
        cin >> deleteInt;
        list1.deleteNode(deleteInt);

        cout << "Enter another number to delete from the list" << endl;
        cin >> deleteInt;
        list1.deleteNode(deleteInt);

        //print the list
        cout<<"list 1: ";
        list1.print();
        return 0;
}

```

## Output

Line 3: Enter numbers ending with -999

1

3

4

7

8

-999

list 1: 1 3 4 7 8

list1 length is 5

list 2: 1 3 4 7 8

list 2 length is 5

All the nodes in list 2 are destroyed

It has been verified that list2 is empty

The first node in list 1 is 1

Enter a number to search

1

Item is found in the list.

Enter another number to search

4

Item is found in the list.

Enter a number to add at the beginning of the list

5

Enter a number to add at the end of the list

8

Enter a number to delete from the list

4

Enter another number to delete from the list

**//linkedlist3**

```
#ifndef H_doublyLinkedList
#define H_doublyLinkedList
```

```
#include <iostream>
```

```
using namespace std;
```

```
//Definition of the node
```

```
template <class Type>
```

```
struct nodeType
```

```
{
```

```
    Type info;
```

```
    nodeType<Type> *next;
```

```
    nodeType<Type> *back;
```

```
};
```

```
template <class Type>
```

```
class doublyLinkedList
```

```
{
```

```
public:
```

```
    void initializeList();
```

```
        //Initialize list to an empty state
```

```
        //Post: first = NULL
```

```
    bool isEmptyList();
```

```
        //Function returns true if the list is empty;
```

```
        //otherwise, it returns false
```

```
    void destroy();
```

```
        //Delete all nodes from the list
```

```

        //Post: first = NULL
void print();
        //Output the info contained in each node
int length();
        //Function returns the number of nodes in the list
void search(const Type& searchItem);
        //Outputs "Item is found in the list" if searchItem
        //is in the list; otherwise, outputs "Item not in the list"
void insertNode(const Type& insertItem);
        //newItem is inserted in the list
        //Post: first points to the new list and the
        // newItem is inserted at the proper place in the list
void deleteNode(const Type& deleteItem);
        //If found, the node containing the deleteItem is deleted
        //from the list
        //Post: first points to the first node of the
        // new list
doublyLinkedList();
        //Default constructor
        //Initialize list to an empty state
        //Post: first = NULL
doublyLinkedList(const doublyLinkedList<Type>& otherList);
        //copy constructor
~doublyLinkedList();
        //Destructor
        //Post: list object is destroyed

private:
    nodeType<Type> *first; //pointer to the list
};

```

```

template<class Type>
doublyLinkedList<Type>::doublyLinkedList()
{
    first= NULL;
}

```

```

template<class Type>
bool doublyLinkedList<Type>::isEmptyList()
{
    return(first == NULL);
}

```

```

template<class Type>
void doublyLinkedList<Type>::destroy()
{
    nodeType<Type> *temp; //pointer to delete the node

    while(first != NULL)
    {
        temp = first;
        first = first->next;
        delete temp;
    }
}

```

```

template<class Type>
void doublyLinkedList<Type>::initializeList()
{
    destroy();
}

```

```

template<class Type>
int doublyLinkedList<Type>::length()
{
    int length = 0;
    nodeType<Type> *current; //pointer to traverse the list

    current = first; //set current to point to the first node

    while(current != NULL)
    {
        length++; //increment length
        current = current->next; //advance current
    }

    return length;
}

```

```

template<class Type>
void doublyLinkedList<Type>::print()
{
    nodeType<Type> *current; //pointer to traverse the list

    current = first; //set current to point to the first node
}

```

```

        while(current != NULL)
        {
            cout<<current->info<<" "; //output info
            current = current->next;
        }//end while
    }//end printList

```

```

template<class Type>
void doublyLinkedList<Type>::search(const Type& searchItem)
{
    bool found;
    nodeType<Type> *current; //pointer to traverse the list

    if(first == NULL)
        cout<<"Cannot search an empty list"<<endl;
    else
    {
        found = false;
        current = first;

        while(current != NULL && !found)
            if(current->info >= searchItem)
                found = true;
            else
                current = current->next;

        if(current == NULL)
            cout<<"Item not in the list"<<endl;
        else
            if(current->info == searchItem) //test for equality
                cout<<"Item is found in the list"<<endl;
            else
                cout<<"Item not in the list"<<endl;
    }
}

```

```

template<class Type>
void doublyLinkedList<Type>::insertNode(const Type& insertItem)
{
    nodeType<Type> *current; // pointer to traverse the list
    nodeType<Type> *trailCurrent; // pointer just before current
    nodeType<Type> *newNode; // pointer to create a node
    bool found;

    newNode = new nodeType<Type>; //create the node

```

```

newNode->info = insertItem; //store new item in the node
newNode->next = NULL;
newNode->back = NULL;

if(first == NULL) //if list is empty, newNode is the only node
    first = newNode;
else
{
    found = false;
    current = first;

    while(current != NULL && !found) //search the list
        if(current->info >= insertItem)
            found = true;
        else
        {
            trailCurrent = current;
            current = current->next;
        }

    if(current == first) //insert new node before first
    {
        first->back = newNode;
        newNode->next = first;
        first = newNode;
    }
    else
    {
        //insert newNode between trailCurrent and current
        if(current != NULL)
        {
            trailCurrent->next = newNode;
            newNode->back = trailCurrent;
            newNode->next = current;
            current ->back = newNode;
        }
        else
        {
            trailCurrent->next = newNode;
            newNode->back = trailCurrent;
        }
    }
} //end else
} //end insertNode

template<class Type>

```

```

void doublyLinkedList<Type>::deleteNode(const Type& deleteltem)
{
    nodeType<Type> *current; // pointer to traverse the list
    nodeType<Type> *trailCurrent; // pointer just before current

    bool found;

    if(first == NULL)
        cout<<"Cannot delete from an empty list"<<endl;
    else
        if(first->info == deleteltem) // node to be deleted is the // first node
        {
            current = first;
            first = first->next;

            if(first != NULL)
                first->back = NULL;

            delete current;
        }
        else
        {
            found = false;
            current = first;

            while(current != NULL && !found) //search the list
                if(current->info == deleteltem)
                {
                    found = true;
                    current = current->next;
                }

            if(current == NULL)
                cout<<"Item to be deleted is not in the list"<<endl;
            else
                if(current->info == deleteltem) //check for equality
                {
                    trailCurrent = current->back;
                    trailCurrent->next = current->next;

                    if(current->next != NULL)
                        current->next->back = trailCurrent;

                    delete current;
                }
            else

```

```

        cout<<"Item to be deleted is not in list."<<endl;
    } //end else
} //end deleteNode

```

```

template<class Type>
doublyLinkedList<Type>::~~doublyLinkedList()
{
    // cout<<"Needs to be written"<<endl;
}
#endif
//linkedList3.cpp
//Program to test the various operations on a doubly linked list

```

```

// #include <iostream>
// #include "link8.h"

```

```

//using namespace std;

```

```

#ifndef H_doublyLinkedList
#define H_doublyLinkedList

```

```

#include <iostream>

```

```

using namespace std;

```

```

//Definition of the node

```

```

template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *next;
    nodeType<Type> *back;
};

```

```

template <class Type>
class doublyLinkedList
{
public:
    void initializeList();
        //Initialize list to an empty state
        //Post: first = NULL
    bool isEmptyList();
        //Function returns true if the list is empty;
        //otherwise, it returns false

```



```

void destroy();
    //Delete all nodes from the list
    //Post: first = NULL
void print();
    //Output the info contained in each node
int length();
    //Function returns the number of nodes in the list
void search(const Type& searchItem);
    //Outputs "Item is found in the list" if searchItem
    //is in the list; otherwise, outputs "Item not in the list"
void insertNode(const Type& insertItem);
    //newItem is inserted in the list
    //Post: first points to the new list and the
    //  newItem is inserted at the proper place in the list
void deleteNode(const Type& deleteItem);
    //If found, the node containing the deleteItem is deleted
    //from the list
    //Post: first points to the first node of the
    //  new list
doublyLinkedList();
    //Default constructor
    //Initialize list to an empty state
    //Post: first = NULL
doublyLinkedList(const doublyLinkedList<Type>& otherList);
    //copy constructor
~doublyLinkedList();
    //Destructur
    //Post: list object is destroyed

private:
    nodeType<Type> *first; //pointer to the list
};

```

```

template<class Type>
doublyLinkedList<Type>::doublyLinkedList()
{
    first= NULL;
}

```

```

template<class Type>
bool doublyLinkedList<Type>::isEmptyList()
{
    return(first == NULL);
}

```

```
}
```

```
template<class Type>
void doublyLinkedList<Type>::destroy()
{
    nodeType<Type> *temp; //pointer to delete the node

    while(first != NULL)
    {
        temp = first;
        first = first->next;
        delete temp;
    }
}
```

```
template<class Type>
void doublyLinkedList<Type>::initializeList()
{
    destroy();
}
```

```
template<class Type>
int doublyLinkedList<Type>::length()
{
    int length = 0;
    nodeType<Type> *current; //pointer to traverse the list

    current = first; //set current to point to the first node

    while(current != NULL)
    {
        length++; //increment length
        current = current->next; //advance current
    }

    return length;
}
```

```
template<class Type>
void doublyLinkedList<Type>::print()
{
    nodeType<Type> *current; //pointer to traverse the list
```

```

    current = first; //set current to point to the first node

    while(current != NULL)
    {
        cout<<current->info<<" "; //output info
        current = current->next;
    }//end while
} //end printList

```

```

template<class Type>
void doublyLinkedList<Type>::search(const Type& searchItem)
{
    bool found;
    nodeType<Type> *current; //pointer to traverse the list

    if(first == NULL)
        cout<<"Cannot search an empty list"<<endl;
    else
    {
        found = false;
        current = first;

        while(current != NULL && !found)
            if(current->info >= searchItem)
                found = true;
            else
                current = current->next;

        if(current == NULL)
            cout<<"Item not in the list"<<endl;
        else
            if(current->info == searchItem) //test for equality
                cout<<"Item is found in the list"<<endl;
            else
                cout<<"Item not in the list"<<endl;
    } //end else
} //end search

```

```

template<class Type>
void doublyLinkedList<Type>::insertNode(const Type& insertItem)
{
    nodeType<Type> *current; // pointer to traverse the list
    nodeType<Type> *trailCurrent; // pointer just before current
    nodeType<Type> *newNode; // pointer to create a node
    bool found;

```

```

newNode = new nodeType<Type>; //create the node
newNode->info = insertItem; //store new item in the node
newNode->next = NULL;
newNode->back = NULL;

if(first == NULL) //if list is empty, newNode is the only node
    first = newNode;
else
{
    found = false;
    current = first;

    while(current != NULL && !found) //search the list
        if(current->info == insertItem)
            found = true;
        else
        {
            trailCurrent = current;
            current = current->next;
        }

    if(current == first) //insert new node before first
    {
        first->back = newNode;
        newNode->next = first;
        first = newNode;
    }
    else
    {
        //insert newNode between trailCurrent and current
        if(current != NULL)
        {
            trailCurrent->next = newNode;
            newNode->back = trailCurrent;
            newNode->next = current;
            current ->back = newNode;
        }
        else
        {
            trailCurrent->next = newNode;
            newNode->back = trailCurrent;
        }
    }
}
} //end else
} //end insertNode

```

```

template<class Type>
void doublyLinkedList<Type>::deleteNode(const Type& deleteltem)
{
    nodeType<Type> *current; // pointer to traverse the list
    nodeType<Type> *trailCurrent; // pointer just before current

    bool found;

    if(first == NULL)
        cout<<"Cannot delete from an empty list"<<endl;
    else
        if(first->info == deleteltem) // node to be deleted is the
                                     // first node
        {
            current = first;
            first = first->next;

            if(first != NULL)
                first->back = NULL;

            delete current;
        }
        else
        {
            found = false;
            current = first;

            while(current != NULL && !found) //search the list
                if(current->info >= deleteltem)
                    found = true;
                else
                    current = current->next;

            if(current == NULL)
                cout<<"Item to be deleted is not in the list"<<endl;
            else
                if(current->info == deleteltem) //check for equality
                {
                    trailCurrent = current->back;
                    trailCurrent->next = current->next;

                    if(current->next != NULL)
                        current->next->back = trailCurrent;

                    delete current;
                }
        }
    }

```

```

    }
    else
        cout<<"Item to be deleted is not in list."<<endl;
    }//end else
} //end deleteNode

```

```

template<class Type>
doublyLinkedList<Type>::~doublyLinkedList()
{
    // cout<<"Needs to be written"<<endl;
}
#endif
int main()
{
    doublyLinkedList<int> intlist;
    int num;

    cout<<"Enter a list of positive integers ending "
        <<"with -999: "<<endl;
    cin>>num;

    while(num != -999)
    {
        intlist.insertNode(num);
        cin>>num;
    }

    cout<<endl;
    cout<<"List in ascending order: ";
    intlist.print();
    cout<<endl;

    return 0;
}

```

output

Enter a list of positive integers ending with -999:

```

5
7
3
9
11
-999

```

```

// class 3.28
#include <string>
#include <iostream>
using namespace std;
typedef string Elem;
class CNode {
private:
    Elem elem;
    CNode* next;

    friend class CircleList;
};

// class 3.29
class CircleList {
public:
    CircleList();
    ~CircleList();
    bool empty() const;
    const Elem& front() const;
    const Elem& back() const;
    void advance();
    void add(const Elem& e);
    void remove();
private:
    CNode* cursor;
};

// constructor 3.30
CircleList::CircleList()
    : cursor(NULL) { }
CircleList::~~CircleList()
    { while (!empty()) remove(); }

// methods 3.31
bool CircleList::empty() const
    { return cursor == NULL; }
const Elem& CircleList::back() const
    { return cursor->elem; }
const Elem& CircleList::front() const
    { return cursor->next->elem; }
void CircleList::advance()
    { cursor = cursor->next; }

// add 3.32
void CircleList::add(const Elem& e) {
    CNode* v = new CNode;
    v->elem = e;
    if (cursor == NULL) {

```

```

        v->next = v; // v points to itself
        cursor = v; // cursor points to v
    }
    else { // list is nonempty?
        v->next = cursor->next; // link in v after cursor
        cursor->next = v;
    }
}
// remove 3.33

void CircleList::remove() { // remove node after cursor
    CNode* old = cursor->next; // the node being removed
    if (old == cursor) // removing the only node?
        cursor = NULL; // list is now empty
    else
        cursor->next = old->next; // link out the old node
    delete old; // delete the old node
}

// test 3.34

int main() {
    CircleList playList; // []
    playList.add("Stayin Alive"); // [Stayin Alive*]
    playList.add("Le Freak"); // [Le Freak, Stayin Alive*]
    playList.add("Jive Talkin"); // [Jive Talkin, Le Freak, Stayin Alive*]

    playList.advance(); // [Le Freak, Stayin Alive, Jive Talkin*]
    playList.advance(); // [Stayin Alive, Jive Talkin, Le Freak*]
    playList.remove(); // [Jive Talkin, Le Freak*]
    playList.add("Disco Inferno"); // [Disco Inferno, Jive Talkin, Le Freak*]
    cout << playList.front() << endl; //
    cout << playList.back() << endl;
    return 0;
}

```

**Output**  
Disco Inferno  
Le Freak

**// recur1.cpp**

// Finding the Sum of the Numbers from 1 to n using recursion  
//

```

#include <iostream>
using namespace std;
int Summation ( /* in */ int n );

int main()
{
    cout << Summation(11) << endl;
}

```



```
int Summation ( /* in */ int n )
// Computes the sum of the numbers from 1 to n by
// adding n to the sum of the numbers from 1 to (n-1)
// Precondition:  n is assigned  &&  n > 0
// Postcondition:
// Function value == sum of numbers from 1 to n
{
    if ( n == 1)                                // base case
        return 1 ;
    else                                         // general case
        return ( n + Summation ( n - 1 ) ) ;
}
```

output

66

```

// recur2.cpp
//
#include <iostream>
using namespace std;

int Factorial ( int number );

int main()
{
    cout << Factorial(10) << endl;
}

int Factorial ( int number )
// Pre: number is assigned and number >= 0.
{
    if ( number == 0)           // base case
        return 1 ;
    else                         // general case
        return number * Factorial ( number - 1 ) ;
}
output
3628800

```

```
// recur3.cpp/
```

```
    #include <iostream>
using namespace std;
```

```
int Power ( int x, int n );
```

```
int main()
{
    cout << Power (3,6) << endl;
}
```

```
int Power ( int x, int n )
```

```
// Pre:  n >= 0.  x, n are not both zero
// Post: Function value == x raised to the power n.
```

```
{
    if ( n == 0 )
        return 1;           // base case
    else
        return ( x * Power ( x , n-1 ) ); // general case
}
```

```
output
729
```

```
// recur4.cpp
```

```
#include <iostream>
using namespace std;
```

```
double Power ( double x, int n );
```

```
int main()
{
    cout << Power (10,-4) << endl;
}
```

```
double Power ( /* in */ double x, /* in */ int n )
```

```
// Precondition: x != 0 && Assigned(n)
// Postcondition: Function value == x raised to the power n.
```

```
{
    if ( n == 0 )                // base case
        return 1;

    else if ( n > 0 )            // first general case
        return ( x * Power ( x , n - 1 ) );

    else                          // second general case
        return ( 1.0 / Power ( x , - n ) );
}
```

```
output
```

```
0.0001
```

```
// recur5.cpp
```

```
//
```

```
#include <iostream>
using namespace std;
```

```
void PrintStars ( /* in */ int numberOfStars );
```

```
int main()
{
    PrintStars(33);
}
```

```
    return 0;
}
```

```
void PrintStars( /* in */ int numberOfStars )
// Prints n asterisks, one to a line
// Precondition:   numberOfStars is assigned
// Postcondition:
//               IF numberOfStars > 0, n stars have been printed, one to a line
//
{
    if (numberOfStars > 0)
    {
        cout << '*';
        PrintStars (numberOfStars-1);
    }
}
```

output

\*\*\*\*\*

**// recur6.cpp**

// print an array in reverse using recursion

**#include <iostream>**

**using namespace std;**

**void PrintRev( const int data[ ], int first, int last );**

**int main()**

{

**int** data[10];

**for** (**int** index=0 ; index < 10; index++)

data[index]=index;

PrintRev(data, 0, 2) ;

cout << **endl**;

PrintRev(data, 3, 9) ;

}

**void PrintRev ( /\* in \*/ const int data [ ], // Array to be printed  
/\* in \*/ int first , // Index of first element  
/\* in \*/ int last ) // Index of last element**

// Prints array elements data [ first. . last ] in reverse order

// Precondition: first assigned && last assigned

// && if first <= last then data [first . . last ] assigned

{

**if** ( first <= last ) // general case

{

cout << data [ last ] << " "; // print last element

PrintRev ( data, first, last - 1 ); // then process the rest

}

// Base case is empty else-clause

}

**output**

2 1 0

9 8 7 6 5 4 3

**// recur7.cpp**

// A recursive function for a function having one parameter that

// generates the nth Fibonacci number.

// f(i+2)=f<sub>i</sub>+f(i+1)

**#include <iostream>**

**#include <cmath>**

**using namespace std;**

// The full recursive version:

**unsigned long Fib1( int n );**

**int main()**

{

**char** ans;

```

int N;
do
{
cout << "I will display fibonacci numbers 0-N." << endl;
cout << "Enter an limit, please. Be patient! This recursive"
<< endl << "Fibonacci routine will take about 17 "
<< endl << "seconds for N = 45 alone " << endl;
cin >> N;
for ( int i = 0; i < N; i++ )
cout << Fib1(i) << endl;
cout << "Y/y to continue, anything else quits" << endl;
cin >> ans;
} while ( 'Y' == ans || 'y' == ans );
}

unsigned long Fib1( int n )
{
if (n == 0 || n == 1)
return 1;
return Fib1( n - 1 ) + Fib1( n - 2 );
}

```

### Output

I will display fibonacci numbers 0-N.  
Enter an limit, please. Be patient! This recursive  
Fibonacci routine will take about 17  
seconds for N = 45 alone

45

1  
1  
2  
3  
5  
8  
13  
21  
34  
55  
89  
144  
233  
377  
610  
987  
1597  
2584  
4181  
6765  
10946  
17711  
28657

46368  
75025  
121393  
196418  
317811  
514229  
832040  
1346269  
2178309  
3524578  
5702887  
9227465  
14930352  
24157817  
39088169  
63245986  
102334155  
165580141  
267914296  
433494437  
701408733  
1134903170  
Y/y to continue, anything else quits