Lecture 3: Data Abstraction

Read: Chpt.3, Carrano

Q: How do we develop large programs?

Program:
Collection of modules

Program Development:
Use TDD with stepwise refinement to produce modular solutions.

A **modular program** is

- Easier to comprehend.
- Easier to write.
- Easier to modify.

Modularity allows us to

- Keeps the complexity of a large program manageable.
- Isolates errors.
- Eliminates redundancies.

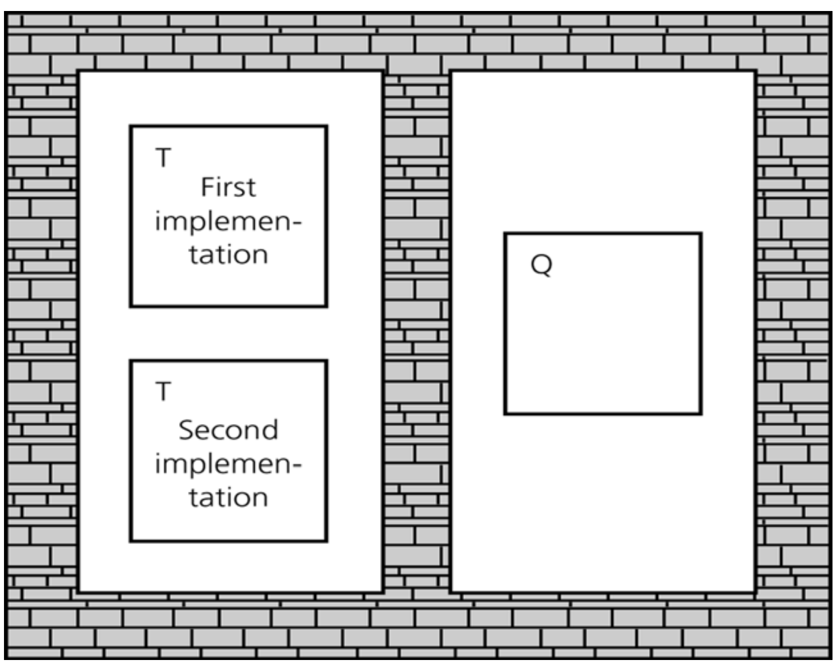
Designing & Constructing Modules:
Use functional and data abstraction

Functional Abstraction:
Allow us to separates the purpose and use of a module from its implementation.

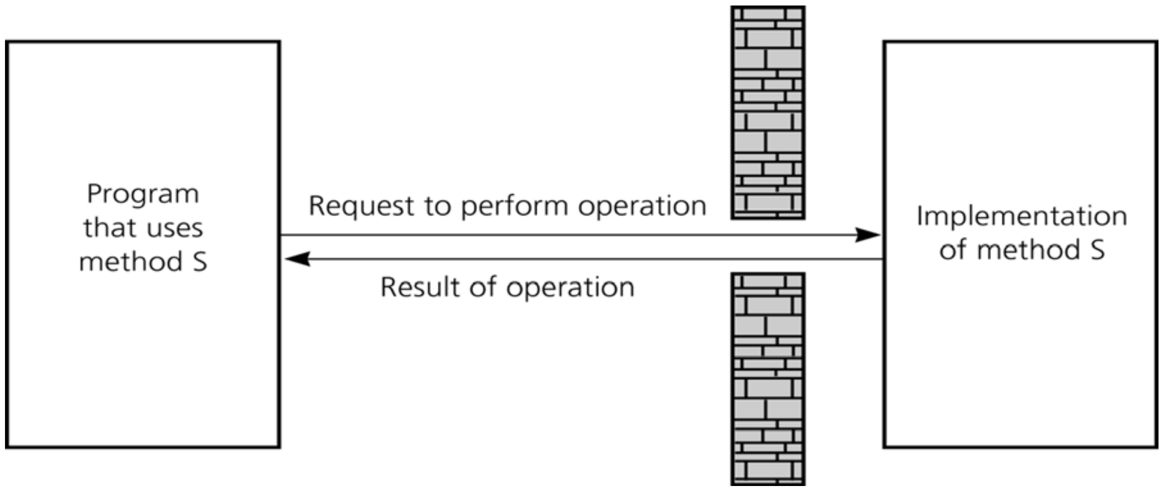
Module Specifications:
- Detail how the module behaves.
- Identify details that can be hidden within the module.

Information hiding in module construction:
- Certain implementation details should be hidden within a module.
- These implementation details should be inaccessible from outside the module.

Example: If task Q uses task T; implementation of T should not affect how Q is using T.



Using a Module:



Remark: The isolation of a module (by its walls) is not total. There needs to be a slit in the wall that allows task Q to interact with task T. How they interact with each other will depend on the specification (contract) of the function.

Data Abstraction:

- A natural extension of functional abstraction.
- Allow us to concentrate on ***what*** we can do to a collection of data instead of ***how*** we do it.
- Allow us to concentrate on the development of a solution to the problem in relative isolation from the implementation details of data organization.

Approach:

Use ***Abstract Data Types (ADTs)***

An **ADT** is collection of data together with a set of operations defined on the data.

- The ***specification*** of an ADT indicates only what the ADT operations do, but not how to implement them.
- The ***implementation*** of an ADT includes choosing a particular data structure and the implementation of each ADT operation in a programming language.

Designing an ADT:

- What data does a problem require?
- What operations does a problem require?

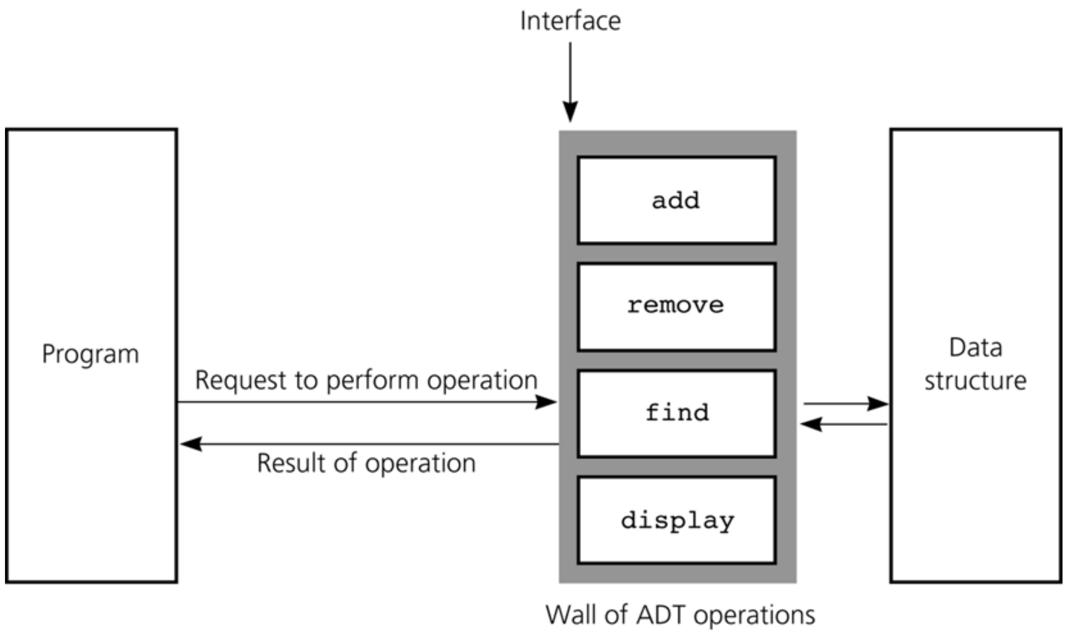
Remark: For complex ADTs, the behavior of the operations can be specified using a set of axioms, which are a set of mathematical rules.

Example: (aList.createList()).size() = 0

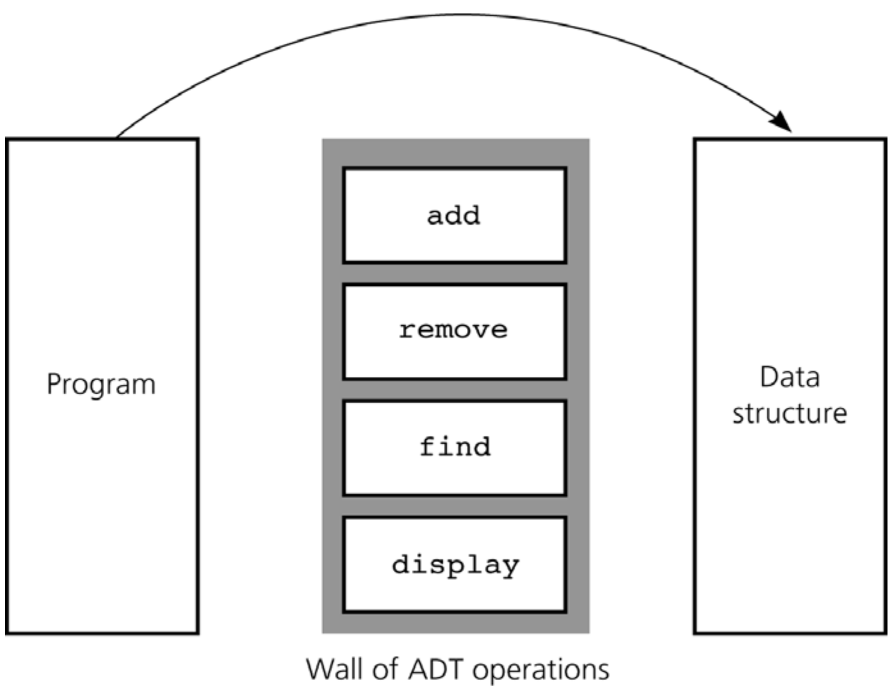
Implementing an ADT:

- Choose a data structure to represent the ADT's data objects.
- Implementation details should be hidden behind a wall of ADT operations such that a program would only be able to access the data structure using the ADT operations.

Example: A wall of ADT operations isolates a data structure from the program that uses it.



Violating the Wall of ADT Operations:

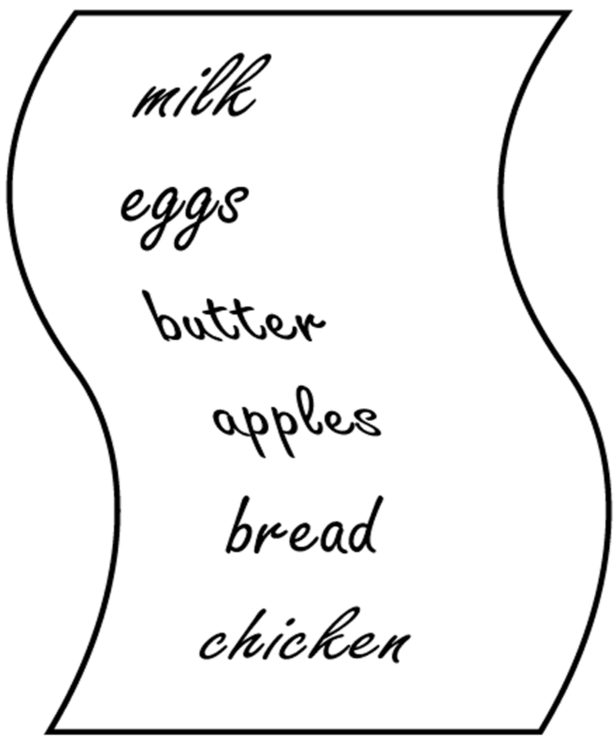


Q: How do we design an ADT?

Consider a general list.

Q: What is a list?

Let’s consider a list of grocery items.



Q: What are the properties of a (grocery) list?

- Except for the first and last items, each item has a unique predecessor and a unique successor.
- Items are referenced by their position within the list.

Roughly, a list is a collection of linearly ordered objects!

Q: What are the data objects and operations that can be defined on a list?

Data Objects:

- Grocery items
- List

ADT Operations:

In the ***specifications*** of the ADT operations, we need to define the contract for the ADT list. However, we do not need to specify how to store the data/list or how to perform the operations. Hence, ADT operations can be used in an application without the knowledge of how the operations will be implemented.

Typical List Operations:

- Create an empty list
- Destroy a list
- Determine whether a list is empty
- Determine the number of items on a list
- Insert an item at a given position in the list
- Delete an item from the list
- Retrieve an item at a given position in the list
- Combining two lists together

Remark: Any collection of these data objects together with a subset (superset) of these operations forms an ADT!

Specifying ADT using UML:

List
items
createList() *destroyList()* *isEmpty()* *getLength()* *insert()* *remove()* *retrieve()*

–items: ListItemType
+createList()
+destroyList()
+isEmpty(): boolean {query}
+getLength(): integer {query}
+insert(in index:integer, in newItem: ListItemType,
out success: boolean)
+remove(in index:integer, out success: boolean)
+retrieve(in index:integer, out dataItem: ListItemType,
out success: boolean) {query}

Using List Operations:
Creating an aList:

```
aList.creatList();
aList.insert(1, milk, success);
aList.insert(2, eggs, success);
aList.insert(3, butter, success);
aList.insert(4, apple, success);
aList.insert(5, bread, success);
aList.insert(6, chicken, success);
```

Inserting a New Item:

```
aList.insert(4, nuts, success);
```

Result:

aList = <milk, eggs, butter, nuts, apple, bread, chicken>

Removing an Item:

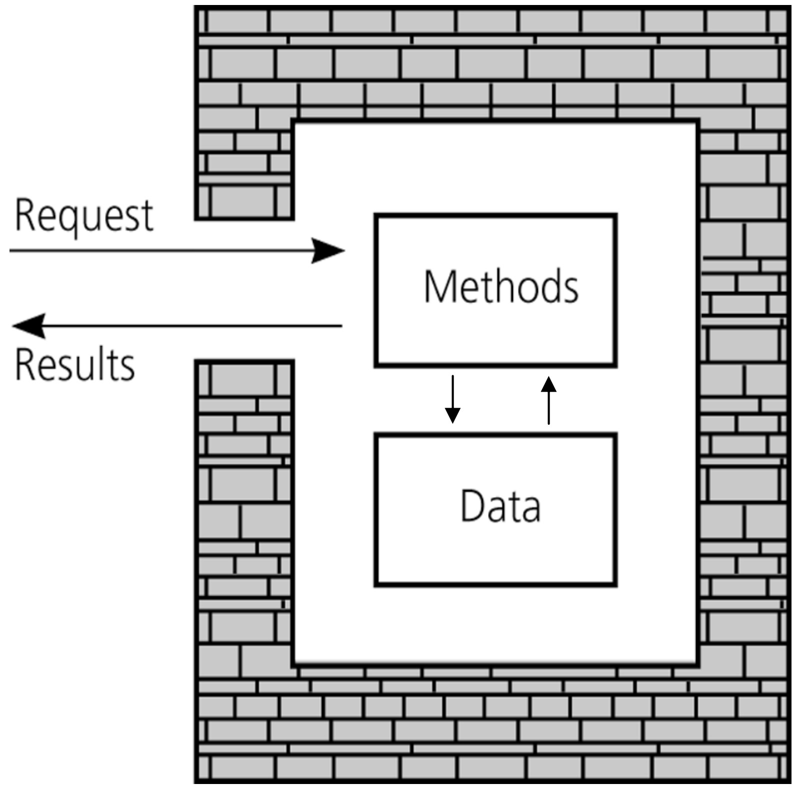
```
aList.remove(5, success);
```

Result:

aList = <milk, eggs, butter, nuts, bread, chicken>

Remark: Observe that both insert and remove operations require the specification of an index.

Using an ADT:



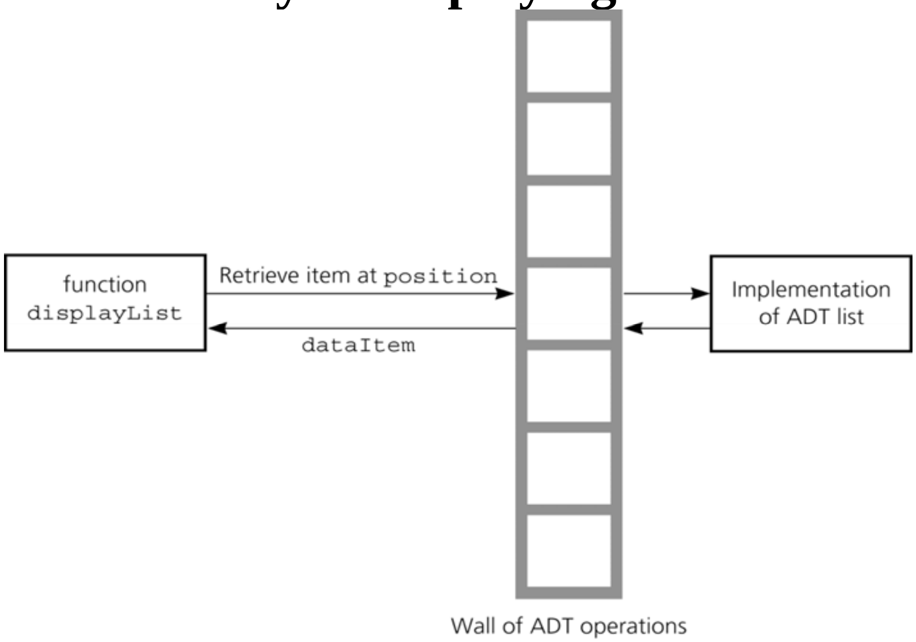
Remark: Private data members should only be accessed using the public methods in the ADT.

Q: What if the wall of an ADT is violated?

Consider a displayList Function for the List class that will

- Retrieve dataItem at a position
- Displace dataItem

Correct Way in Displaying a List:

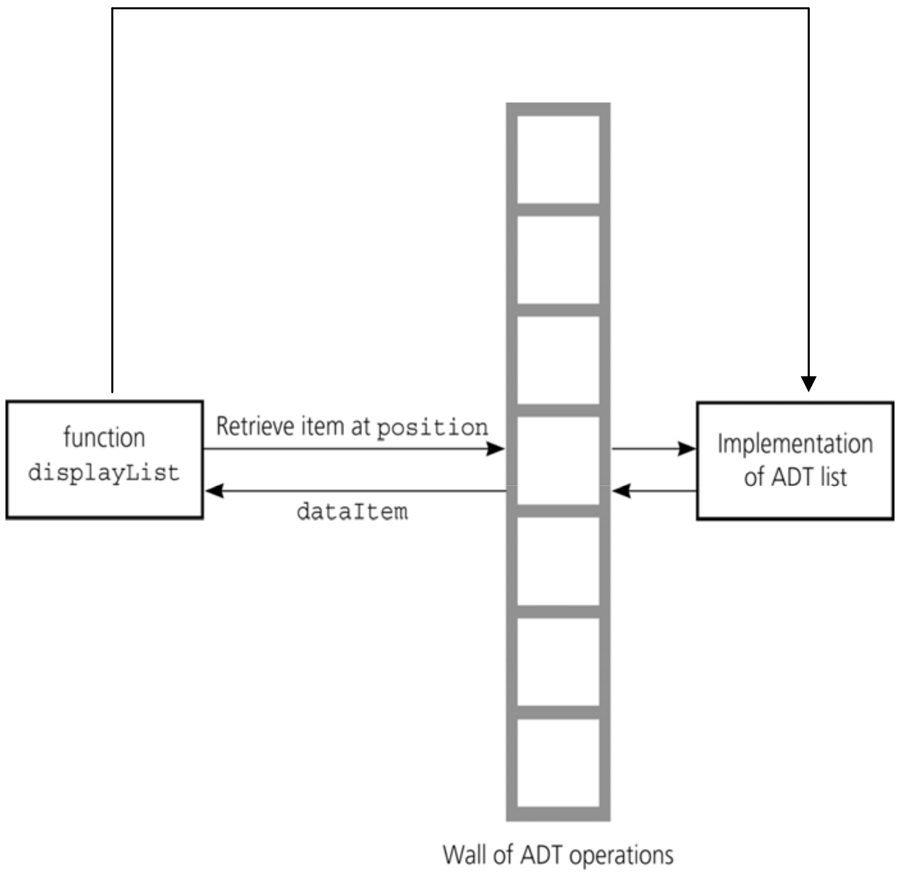


Algorithm for displayList Function:

```
displayList(in aList: List)
{
   for (position = 1 to aList.getlength()) do
         aList.retrieve(position, dataItem, success);
         display dataItem;
   endfor;
}
```

Remark: Independent of implementation of List, displayList algorithm is always the same!

Violating the Wall of ADT Operations:



Bad displayList Function:

```
displayList(in aList: List)
{
   for (position = 1 to aList.getlength()) do
         display aList[position];     // Implementation dependent
   endfor;
}
```

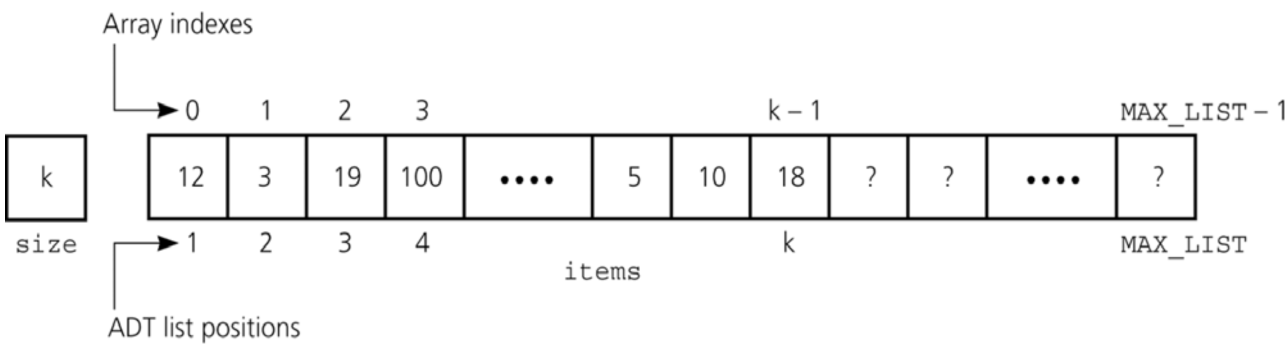
Implementing the ADT List:

1. **Array Implementation of List:**
    - Must declare maximum length of List.
    - All dataItem objects are of ListItemType.
    - Use array element to hold dataItem objects.
    - Use size to hold length of List.

Example: Consider List of integers.

Implementation:

```
const int MAX_LIST = 100;            // max length on list
typedef int ListItemType;            //data type of list items
ListItemType items[MAX_LIST];        //array of list items
int size;                            //length of list
```

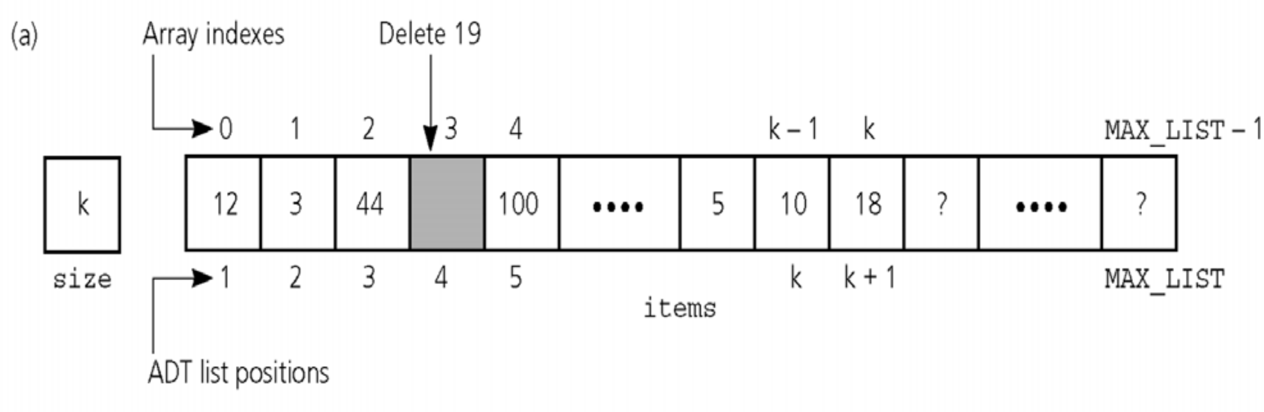


Warning: For any object in the List, its list position is one higher than its array index!

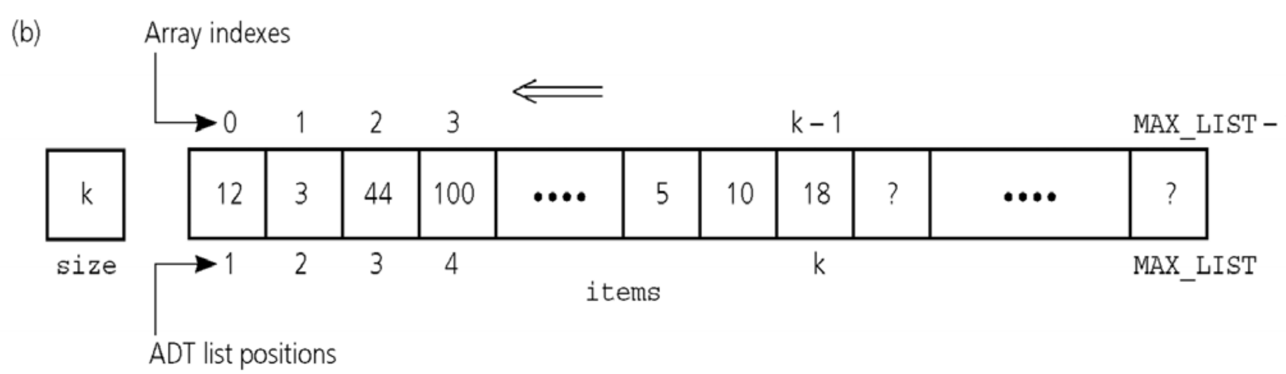
Q: Is array implementation a good way to implement a List class?

Must consider the efficiency of its operations!

Consider removing an item from an array:
remove(4,success):



Filling a Gap by Shifting:



Observations:

- Removing (inserting) a dataItem may require shifting the objects in the array.
- Shifting can be expensive. Hence, array is not a good data structure for implementing List when frequent insertions/deletions are required!

Implementing an ADT in C++:

C++ Classes:

- Encapsulation combines an ADT's data with its operations to form an object.
  - An object is an instance of a class.
  - A class contains data members and member functions.
- By default, all members in a class are private.
- Each class definition is placed in a header file.
  - *Classname*.h
- The implementation of a class's member functions are placed in an implementation file.
  - *Classname*.cpp

Constructors:

- A class can have several constructors.
- A constructor must have the same name as the class.
- Constructor has no return type, not even void
- It is used to create and initialize new instances of a class.
- If one is omitted, the compiler will generate a default constructor.
- A default constructor has no arguments.
- The implementation of a constructor (or any member function) is qualified with the scope resolution operator ::
  Sphere::Sphere(**double** initialRadius):
  theRadius(initialRadius)

Destructor:

- Each class has one destructor.
- It destroys an instance of an object when the object's lifetime ends.
- If one is omitted, the compiler will generate a default destructor.
- For many classes, the destructor can be omitted.

Header (ListA.h) file for List (Page 156):

```
const int MAX LIST = maximum-size-of-list;
typedef desired-type-of-list-item ListItemType;

class List
{
public:
        List( );                          // constructor
                                          // default destructor

// list operations:
   bool  isEmpty( ) const;
   // Determine whether a list is empty.
   // Precondition:  None.
   // Precondition:  Returns true if the list is empty;
   // otherwise returns false.
```

int getLength( ) const;
//Determines the length of a list.
//Precondition: None
//Postcondition: Returns the number of items
//that are currently in the list.

**void insert(int index, ListItemType newItem,
bool& success);**
//Inserts an item into the list at position index.
//Precondition: Index indicates the position at which
//the item should be inserted in the list.
//Postcondition: If insertion is successful, newItem is at
//position index in the list, and other items are renumbered
//accordingly, and success is true; otherwise success is
//false. Note: Insertion will not be successful if
//index < 1 or index > getLength( ) + 1.

void remove(int index, bool& success);
//Deletes an item from the list at a given positon.
//Precondition: Index indicates where the deletion
//should occur.
//Postcondition: If 1 < = index <= getLength( ),
//the item at position index in the list is
//deleted, other items are renumbered accordingly,
//and success is true; otherwise success is false.

```
void retrieve (int index, ListItemType& dataItem,
                                        bool& success) const;
   //Retrieves a list item by position.
   //Precondition:  index is the number of the item to
   //be retrieved.
   //Postcondition:  If 1 <= index <= getLength( ),
   //dataItem is the value of the desired item and
   //success is true; otherwise success is false.

private:
   ListItemType items[MAX LIST]; // array of list items
   int size;                       //number of items in list

   int translate(int index) const;
   //Converts the position of an item in a list to the
   //correct index within its array representation.

}; // end List class
// End of header file.
```

The implementations of the functions that the previous header file declares appear in the following ListA.cpp file:

Implementation (.cpp) file for List:

```
#include "ListA.h"                // include header file

List::List( )  :  size (0)
{
 } // end constructor

bool List::isEmpty( ) const
{
     return bool(size = = 0);
}  // end isEmpty

int List::getLength( ) const
{
     return size;
} // end getLength
```

```
void List::insert (int index, ListItemType newItem,
                                              bool & success)
   {
      success = bool( (index >= 1)   &&
                      (index <= size + 1) &&
                      (size < MAX LIST) );
      if (success)
      {  // make room for new item by shifting all items at
        //  positions >= index toward the end of the
        //  list (no shift if index == size + 1)
        for (int pos = size; pos >= index; ––pos)
           items[translate(pos + 1)] = items[translate(pos)];

      // insert new item
      items[translate(index)] = newItem;
      ++size;  // increase the size of the list by one
      }  // endif
   }  //  end insert
```

```
void List::remove(int index, bool & success)
{
   success = bool ( (index >= 1) && (index <= size) );

   if (success)
    {  // delete item by shifting all items at positions
       // > index toward the beginning of the list
       // (no shift if index == size)
       for (int fromPosition = index + 1;
                 fromPosition <= size; ++fromPosition)
           items[translate(fromPosition-1)] =
                                   items[translate(fromPosition)];
         ––size;  // decrease the size of the list by one
    }  //  end if

}  // end remove

void List::retrieve(int index, ListItemType& dataItem,
                          bool & success) const
{
   success = bool( (index >= 1)   &&
                         (index <= size) );
   if (success)
       dataItem = items[translate(index)];
} // end retrieve
```

```
int List::translate(int index) const
{
        return index – 1;
} // end translate

// End of implementation file.
```

Q: Is List a good data structure if one has to search frequently for dataItems?

No. May need to examine many items in the List!

Possible Improvement:

Sorted List!

The ADT sorted list

- Maintains items in sorted order.
- Inserts and deletes items by their values, not their positions.

ADT: SortedList.

SortedList
items
+createSortedList()
+destroySortedList()
+sortedIsEmpty(): boolean {query}
+sortedGetLength(): integer {query}
+sortedInsert(in newItem: ListItemType, out success: boolean) // Insert newItem into its sorted position in the sorted list.
+sortedRemove(in anItem: ListItemType, out success boolean)

+sortedRetrieve(in index: integer, out dataItem:
ListItemType, out success: boolean) {query}
//If 1 <= index <= sortedGetLength(), set dataItem to the
//item at position index of the sorted list.

+locatePosition(in anItem: ListItemType, out isPresent: boolean): integer {query} //Return position of anItem in the sorted list.

More on ADTs Designs:

The design of an ADT is an evolutionary process

- During initial design and implementation
- Throughout the life cycle

Example: Create a **Circle** class

A circle is characterized by its center and radius.

Circle
–centerX: double –centerY: double –radius: double
+createCircle +move(in newX: double, in newY: double) +area() {query} : double +distanceTo(in c: Circle) {query} : double +distanceTo(in x: double, in y: double) {query} : double +onCircle(in x: double, in y: double) {query} : Boolean …

Consider a new class, say Line:

A line is characterized by any two points on it.

Line
–x1: double –y1: double –x2: double –y2: double
+createLine +length( ) {query} : double +move(in newX1: double, in newY1: double, in newX2: double, in newY2: double) …

Remark: Notice that the concept of point is not yet captured in these classes!

Discovering a Missing Class:

Example: Create a **Point** class

Point
–x: double –y: double
+createPoint +distanceTo(in p: Point) {query} : double +getX() : double +getY() : double +setX(in : double) +setY(in : double)

Then our **Circle** and **Line** classes becomes

Circle
–center: Point –radius: double
+createCircle +move(in newC: Point) +area() {query} : double +distanceTo(in c: Circle) {query} : double +distanceTo(in p: Point) {query} : double +onCircle(in p: Point) {query} : boolean …

Line
–p1: Point –p2: Point
+createLine +length() {query} : double +move(in newP1: Point, in newP2: Point) …

```
// Point.h

#include <iostream>

class Point
{
    public:
        // Constructors:
        Point(); // default constructor sets (x,y) to (0,0)
        Point(const Point& p); // copy constructor
        Point(double initX, double initY);
        // constructor sets (x,y) to given values

        // General public instance methods
        double distanceTo(Point p);
        double getX( );
        double getY( );
        void setX(double newX);
        void setY(double newY);

    private:
        double x;
        double y;
};
```

```
// Point.c++

#include <cmath>
using namespace std;

#include "Point.h"

// Constructors:

// Default constructor:
Point::Point() : x(0.0), y(0.0)
{
}

// Copy constructor:
Point::Point(const Point& p) : x(p.x), y(p.y)
{
}

// Constructor to set initial coordinates to given values
Point::Point(double initX, double initY) : x(initX), y(initY)
{
}
```

```
// general public instance methods

double Point::distanceTo(Point p)
{
	double xDiff = x - p.x;
	double yDiff = y - p.y;
	return sqrt(xDiff*xDiff + yDiff*yDiff);
}

double Point::getX()
{
	return x;
}

double Point::getY()
{
	return y;
}

void Point::setX(double newX)
{
	x = newX;
}

void Point::setY(double newY)
{
	y = newY;
}
```

```
// Circle.h
#include <iostream>
#include "Point.h"
class Circle
{
    public:
        // Constructors
        Circle(); // default; makes a unit circle centered at origin
        Circle(const Circle& c); // copy constructor
        Circle(Point c, double r);
        Circle(double cx, double cy, double r);

        // general public instance methods
        double distanceTo(Circle c);
        double getArea( );
        Point  getCenter( );
        void   getCenter(double& cx, double& cy);
        double getDiameter( );
        double getRadius( );
        void setCenter(Point C);
        void setCenter(double cx, double cy);
        void setDiameter(double d);
        void setRadius(double r);

    private:
        Point center;
        double radius;
};
```

```
// Circle.c++

#include <cmath>
using namespace std;

#include "Circle.h"

// Default constructors
Circle::Circle() : center(0.0,0.0), radius(1.0)
{
}

// Copy constructor
Circle::Circle(const Circle& c) : center(c.center),
radius(c.radius)
{
}

// Constructor with explicit center and
Circle::Circle(Point c, double r) : center(c), radius(r)
{
}

// Constructor with explicit center and radius
Circle::Circle(double cx, double cy, double r) : center(cx,cy),
radius(r)
{
}
// general public instance methods
```

```
double Circle::distanceTo(Circle c)
{
// delegate part of the job to the Point::distanceTo method
	double dCenterToCenter = center.distanceTo(c.center);
	return dCenterToCenter - radius - c.radius;
}

double Circle::getArea( )
{
	return M_PI * radius * radius;
}

Point Circle::getCenter( )
{
	return center;
}

void Circle::getCenter(double& cx, double& cy)
{// We have direct access to "Center", but not to its x and y
	cx = center.getX();
	cy = center.getY();
}

double Circle::getDiameter( )
{
	return 2.0 * radius;
}
double Circle::getRadius( )
{
```

```
    return radius;
}

void Circle::setCenter(Point C)
{
    center = C;
}

void Circle::setCenter(double cx, double cy)
{
    center.setX(cx);
    center.setY(cy);
}

void Circle::setDiameter(double d)
{
    radius = 0.5 * d;
}

void Circle::setRadius(double r)
{
    radius = r;
}
```

Brief Introduction to Namespaces:

C++ does not require that all variables and functions be declared as members of a class. It provides a mechanism to group together a collection of logically related classes and declarations such as global constants and functions in a region called ***namespace***.

Reasons:

- Allows us to directly declare that the collection does in fact have significance to the program. That is, it is a declaration that they are logically related.
- Helps to prevent accidental reuse and blocking of names.

Declaring a namespace:

```
namespace nameSpaceName
{
    declarations
}
```

As with classes, the implementations may be inside, or outside, the namespace declaration.

Example: Given a game program and its GUI:

- Module 1 uses a global state variable called lastTip to remember the last tool tip that was used in its GUI.
- Module 2 uses a global state variable called lastTip to remember the last clue that was given to the player in the game being played.

```
namespace GUI
{
    string lastTip;
    void showlastTip()
    {
        cout << “Last GUI tip was: ” << lastTip
            << endl;
    }
}

namespace Game
{
    string lastTip;
    void showLastTip();
}

void Game::showLastTip()
{
    cout << “Last Game tip was: ” << lastTip
        << endl;
}
```

Client Access:

```
#include "GUI.h"
#include "Game.h"


void show1()
{
    GUI::showLastTip();    // using scope resolution operator
}


void show2()
{
    using namespace Game;
    showLastTip();          // access thro using namespace
}
```

There is also an unnamed global namespace.

- Anything not declared in an explicit namespace is a part of the global namespace.
- For example, if there was a third version of showLastTip:

```
void showLastTip()
{
    cout << "Blah\n";
}
void show3()
```

```
{
    using namespace Game;
    GUI::showLastTip();    // invokes the 'GUI' one
    showLastTip();         // invokes the one in 'Game'
    ::showLastTip();       // invokes the one in the global NS
}
```

Example:

```
#include <iostream.h>

namespace N1
{   int j = 11;
    //  double a, b;

    namespace sub
    {
        double x = 16.8;
    }
};
```

```
namespace N2
{   int j = 22;
    //  double c, d;
    namespace sub
    {
        double y = 26.8;
    }
}

int main()
{
    using namespace N1;
    using namespace N2;
    //  a = 1;
    //  d = 2;
    cout << N2::j << ‘ ’ << N1::sub::x << endl;    // legal!
    return 0;
}
```

Remarks:

- Illegal to replace N2::j with ::j.
- Is it legal to have the comment statements activated?

Brief Introduction to C++ Exceptions

Exception is a run-time error caused by some abnormal conditions.

Consider we need to return the square root of a real.

```
double sqrt(double x)
//   Computing the square root of x
//   Precondition:   Real number x > 0
//   Postcondition: Return the square root of x
{
  return sqrt(x);
}
```

Q: What if $x < 0$?

- Normally runtime exceptions crash your program.
- In C++, exception handling mechanism allows program to continue after catching exception(s).

Exception Handling:

- Catching an exception
- Throwing an exception
- Claiming an exception

1. Catching (Handling) an Exception:

- Use try-catch block.

  Syntax:

```
try
{
    statements;
}
catch (ExceptionClass1  identifier)
{
    statements;
}
    …
catch (ExceptionClassk  identifier)
{
    statements;
}
```

- Statement(s) that may cause an exception will be placed inside a try-block.
- Each try-block must immediately be followed by one or more catch-blocks.
- Each catch-block must indicate the type of exception it will handle.
- The ordering of catch-blocks is not important.

- When a statement in try-block causes an exception, the remaining statements of the try-block will be skipped and control will be passed on to the catch-block corresponding to the type of exception thrown (detected).
- After the execution of a catch-block, all remaining catch-blocks (associated with the same try-block) will be skipped.
- If a corresponding catch-block can not be found, program will (usually) abort.

2. Throwing an Exception:

When an exception is detected within a method, we use a throw-statement to indicate that an exception has occurred. Once the ***throw-statement*** is executed, the remaining statements in the method will be skipped.

Syntax:

```
throw ExceptionClass(stringArgument);
```

3. **Claiming an Exception:**

To specify the type of exception(s) that can be thrown by a method, a ***throw-clause*** must be included with the method's header.

Syntax:

```
functionDeclaration throw (exceptionClass1,
                                      …, exceptionClassk)
```

Example:

```
double sqrt(double x) throw (NegativeNumberException)
{   if (x < 0.0)
        throw NegativeNumberException("negative
                                          num in sr");
    return sqrt(x);
}
```

Consider the following caller functions.

```
void compute()    // caller
{
    double s1 = sqrt(4.0);
    cout << "First: " << s1 << endl;
    double s2 = sqrt(-3.2);
    cout << "Second: " << s2 << endl;
}
```

Output:

First: 2

Abort (It will not catch the exception!)

To "catch" the exception, we use:

```
void compute()
{
	try
	{
		double s1 = sqrt(4.0);
		cout << "First: " << s1 << endl;
	}
	catch (NegativeNumberException n)
	{
		cout << "Can't do sqrt(4.0)\n";
	}

	try
	{
		double s2 = sqrt(-3.2);
		cout << "Second: " << s2 << endl;
	}
	catch (NegativeNumberException n)
	{
		cout << "Can't do sqrt(-3.2)\n";
	}
}
```

Output:

```
First: 2
Can't do sqrt(-3.2)
```

We can also have one try-catch block that includes multiple things that might fail.

```
void compute()
{
    try
    {
        double s1 = sqrt(4.0);
        cout << "First: " << s1 << endl;
        double s2 = sqrt(-3.2);
        cout << "Second: " << s2 << endl;
    }
    catch (NegativeNumberException n)
    {
        cout << "You tried to do something bad!!\n";
    }
}
```

Q: What is this "NegativeNumberException" thing?

- This is an exception class that we can construct.
- There is a C++ class called *exception* that is the base class of all exceptions that may be thrown.
- Subclasses include *runtime_error*, *logic_error*, and others.

Example:

```
class NegativeNumberException: public runtime_error
{
    public:
        NegativeNumberException(const string& m = "")
                                    : runtime_error(m.c_str())
        {
        }
};
```

Warning: Note that not all compilers have a constructor for the base class *exception* that takes a character string. Indeed, neither Metrowerks nor g++ does.

Remark: See implementation of ADT List using exceptions in Carrano. Also, read Appendix A on Exceptions.