Example: Array-based List

List : A collection of elements of the same type.

Operations on a list
1. Create a list (empty list)
2. Determine if the list is empty.
3. Determine if the list is full.
4. Find the size (length) of the list.
5. Destroy, or clear, the list.
6. Determine whether an item is contained in the list.
7. Insert an item (at the default location or a given location).
8. Remove an item (from the default location, or a given location, or given value).
9. Retrieve an item from the list (at the default location, or the specified location).
10. Search the list for a given item.

Implementation consideration 1:
How to store the list in the computer's memory?

- In this example, we store the list items using an array.

Implementation consideration 2:
We want the program codes to be reusable for different data types, e.g. `int`, `double`, `fraction`, and etc.

- Develop generic codes using **`template`**
- In Java, this is called generic class (or generic programming)

```
// filename: arrayListType.h

#ifndef ARRAY_LIST_TYPE_H
#define ARRAY_LIST_TYPE_H

template<class elemType>
class arrayListType
{
protected:
   elemType *list; //array to hold list elements
   int length;     //no. of elements in the list
   int maxSize;    //physical size of array

public:
   arrayListType(int size = 100);
   //constructor, default size = 100

   arrayListType(const arrayListType<elemType>& other);
   //copy constructor

   ~arrayListType();
   //destructor, deallocate the memory occupied by the array
   //A user defined destructor is necessary if the object
   //instance contains dynamically allocated memory.

   //accessor functions
   bool isEmpty() const;
   bool isFull() const;
   int listSize() const;
   int maxListSize() const;
   void print() const;
   virtual int search(const elemType& item);
   void retrieve(int loc, elemType& item);
   arrayListType<elmType>& operator=
                         (const arrayListType<elmType>& other);

   //mutator functions
   void clearList();
   virtual void insert(const elemType& item);
   virtual void remove(const elemType& item);
   virtual void removeAt(int loc);

//search, insert, remove, removeAt are virtual functions.
//Their implementations can be overridden in derived classes.

};  //end of class definition
```

```
//Implementations of the functions in the template class
//should be put in the .h file of the template class

template<class elemType>
arrayListType<elemType>::arrayListType(int size)
{
   if (size < 0)
   {
      cerr << “List size must be positive” << endl;
      maxSize = 100;
   }
   else
      maxSize = size;

   list = new elemType[maxSize]; //dynamic memory allocation

   assert (list != NULL); //For development (debugging) phase.
                          //Assertion test is usually turned
                          //off during production phase.
}

//overload the copy constructor
template<class elemType>
arrayListType<elemType>::arrayListType
                      (const arrayListType<elemType>& other)
{
   //The copy constructor is called when an object is passed
   //as a value parameter to a function.

   maxSize = other.maxSize;
   length = other.length;

   //make a copy of the array
   list = new elemType[maxSize];
   assert(list != NULL);

   for (int i = 0; i < length; i++)
      list[i] = other.list[i];
}


template<class elemType>
arrayListType<elemType>::~arrayListType()
{
   delete [] list; //free the memory occupied by the array
```

```
}

//overload (redefine) the assignment operator
template<class elemType>
arrayListType<elemType>& arrayListType<elmType>::operator=
                      (const arrayListType<elemType>& other)
{
   if (this != &other)
   {
      length = other.length;

      if (maxSize != other.maxSize)
      {
         maxSize = other.maxSize;
         delete [] list;
         list = new elemType[maxSize];
         assert(list != NULL);
      }

      for (int i = 0; i < length; i++)
         list[i] = other.list[i];
   }
   return *this;
}

template<class elemType>
bool arrayListType<elemType>::isEmpty() const
{
   return length == 0;
}

template<class elemType>
bool arrayListType<elemType>::isFull() const
{
   return length >= maxSize;
}

template<class elemType>
int arrayListType<elemType>::listSize() const
{
   return length;
}

template<class elemType>
int arrayListType<elemType>::maxListSize() const
{
   return maxSize;
}

template<class elemType>
void arrayListType<elemType>::print() const
{
   for (int i = 0; i < length; i++)
      cout << list[i] << “ ”;

   cout << endl;
}

template<class elemType>
int arrayListType<elemType>::search(const elemType& item)
{
   for (int i = 0; i < length; i++)
      if (list[i] == item)
         return i;

   return -1; //item not found
}

template<class elemType>
void arrayListType<elemType>::
                           retrieve(int loc, elemType& item)
{
   if (loc < 0 || loc >= length)
      cerr << “Error: Index out of bound” << endl;
   else
      item = list[loc];
}

template<class elemType>
void arrayListType<elemType>::clearList()
{
   length = 0;
}

template<class elemType>
void arrayListType<elemType>::insert(const elemType& item)
{
   if (!isFull())
      list[length++] = item;
}
```

```
template<class elemType>
void arrayListType<elemType>::remove(const elemType& item)
{
   int i = 0;
   bool done = false;

   while (i < length && !done)
   {
      if (list[i] == item)
      {
         list[i] = list[--length];
         done = true;
      }
      else
         i++;
   }
}

template<class elemType>
void arrayListType<elemType>::removeAt(int loc)
{
   if (loc < 0 || loc >= length)
      cerr << "Error: Index out of bound" << endl;
   else
      list[loc] = list[--length];
}


#endif //end of the .h file
```

```
//Example program that makes use of arrayListType

int main()
{
   arrayListType<int> intList(100);

   arrayListType<fraction> fractionList(50);

   for (int i = 1; i < 20; i++)
      intList.insert(i);

   intList.print();

   for (int t = 1; t < 10; t++)
   {
      fraction *f;

      f = new fraction(t, t+1);
      fractionList.insert(*f);
   }

   fractionList.print();

   ...
}
```

Remark about the compiler regarding the `operator=`

```
arrayListType<int> list_1(20);

for (int i = 0; i < 20; i++)
   list_1.insert(i);

arrayListType<int> list_2;
//list_2 created by default constructor

list_2 = list_1;  //use the overloaded operator=


arrayListType<int> list_3 = list_1; //initialization
//list_3 is created by direct copying of the member variables
//of list_1
//The overloaded operator= is NOT used !!
```

Elements of the array-based list in the above example are unordered. We can implement an ordered list using inheritance.

```
#ifndef ORDERED_ARRAY_LIST_TYPE_H
#define ORDERED_ARRAY_LIST_TYPE_H

#include "arrayListType.h"

template <class elemType>
class ordered_arrayListType : public arrayListType<elemType>
{
   //no additional variables required in this example

public:
   ordered_arrayListType() : arrayListType()
   { }

   ordered_arrayListType(int n) : arrayListType(n)
   { }

   //override the virtual functions in base class
   int search(const elemType& item);
   void insert(const elemType& item);
   void remove(const elemType& item);
   void removeAt(int loc);
};


template<class elemType>
int ordered_arrayListType<elemType>::
                                    search(const elemType& item)
{  //use binary search
   int low = 0;
   int high = length-1;
   while (low <= high)
   {
      int mid = (low + high)/2;
      if (list[mid] == item)
         return mid;
      if (list[mid] < item)
         low = mid + 1;
      else
         high = mid - 1;
   }

   return -1; //item not found
}
//mutator functions should maintain the ordering of elements
//in the list

template<class elemType>
void ordered_arrayListType<elemType>::
                                   insert(const elemType& item)
{
   if (isFull())
      return;

   int i;  // i must be declared outside the for statement !!
   for (i = length-1; i >= 0 && list[i] > item; i--)
      list[i+1] = list[i];

   list[i+1] = item;
   length++;
}


template<class elemType>
void ordered_arrayListType<elemType>::
                                  remove(const elemType& item)
{
   int k = search(item);
   if (k >= 0)
   {
      for (int i = k; i < length-1; i++)
         list[i] = list[i+1];
      length--;
   }
}


template<class elemType>
void ordered_arrayListType<elemType>::removeAt(int loc)
{
   if (loc >= 0 && loc < length)
   {
      for (int i = loc; i < length-1; i++)
         list[i] = list[i+1];
      length--;
   }
}

#endif
```

Remarks on array-based list

1. The size of the physical array is fixed during execution. The physical size of the array and the logical length of the list are two different attributes.
2. New items can be added to the array only if there is room.
3. Expansion of an array is possible – create a new array and copy the contents of the old array to the new array. But this operation is time consuming.
4. If the array is ordered, then searching can be done more efficiently using binary search.
5. Insertion and deletion operation on an ordered array is time consuming, i.e. requires shifting of the array elements to maintain the ordering.