Array

Topics:

+ Generic Array Object
+ Array Data Structure
+ Insertion Functions
+ Removal Functions
+ Searching Functions
+ Resizing Function
+ Vector Data Structure Object

Resources:

- `Array.h`
- `Repository.h`
- `Vector.h`
- `main.cpp`

Introduction

An array is one of the well-known data structure. It has been used from the beginning of your studies in computer science. In this lecture, we implement insertion, removal, search and other modification function of an array that stores data in sequence. But first, we will construct the data structure to simplify implementations.

Generic Array Object

When dealing with an array, we will often have to access its size; hence, we defined the generic **Array** class that has a method that accesses the size of the array. Its methods are

- `Array()` - it initializes the capacity of the array to 20 and assigns the default value of the type to each element of the array.
- `Array(`*cp*`)` - it initializes the capacity of the array to *cp* if positive and assigns the default value of the type to each element of the array; otherwise, it does the same thing as the default constructor.
- `Array(`*obj*`)` - it performs a deep copy of *obj*.
- `operator=(`*obj*`)` - it performs a deep copy of *obj*.

- ~`Array()` - it deallocates the array.
- `operator[](`*idx*`)` - it returns the element of the array with index *idx* if *idx* is valid; otherwise, it throws an error.
- `Length()` - it returns the capacity of the array.
- `Resize(`*cp*`)` - it resizes the array to the capacity *cp* if *cp* is a positive value. The elements of the array are assigned the default value of the type.
- `ToString()` - it returns a string of the elements of the array separated by commas enclosed in square brackets.
- `operator<<(`*out*`,`*obj*`)` - it displays the string as `ToString()`.

Array Data Structure

Typically, whenever you use an array data structure, you populate it sequentially. Thus, you will need an index that indicates how many elements of the array have been actually filled. This index differs from the capacity of an array. Therefore, we will first define the array structure

```
template <class T>
struct array
{
 Array<T> data;
 int size;
};
```

This structure starts with the default capacity; however, you can always adjust the size with the `Resize()` method. The following is a list of helper functions.

- `Initialize(`*obj*`,`*cp*`)` - it resizes *data* to the capacity *cp* if *cp* positive and assigns 0 to *size*.

Insertion Functions

When you are inserting data in an array, you can insert it anywhere by using an index as seen in the function below.

```
template<typename T>
void Insert(array<T>& obj,int idx,const T& item)
{
 if(obj.size < obj.data.Length())
 {
  if(idx >= 0 && idx <= obj.size)
  {
   for(int i = obj.size;i > idx;i -= 1)
   {
    obj.data[i] = obj.data[i-1];
   }
   obj.data[idx] = item;
   obj.size += 1;
  }
 }
}
```

Now, let us breakdown the function. First, the first if statement determines if you can add a new value to the array considering that the array has a fixed capacity. Since the insertion function is going to add a new value to the array, you must initially check if there is space for it.

Second, the second if statement determines if the index *idx* is valid. The new value is suppose to be placed in an existing position in the array or at the end of the array. This means that *idx* must be between 0 and *size* inclusively since *size* indicates the size of the array. Furthermore, *size* is also the next index in the array.

Third, the for loop shifts the data in the array over one element until the element with index *idx*. It begins at the end of the array. Then for each instance of its body, it assigns the current element the value of the element that immediately precedes it. For instances, if *data* = [a,b,c,d,e] (*size* = 5) and *idx* = 2, the changes to the array for each run is as follows

Run	Array
1	[a,b,c,d,e,e]
2	[a,b,c,d,d,e]
3	[a,b,c,c,d,e]

Last, the two lines following the for loop adds *item* to the array and increment *size* by 1 respectively. At this the point, the array includes the new value and is in a consistent state. This function has a worst-case scenario O(n) (linear) runtime where n refers to *size*.

If you were to only insert at the end of the array, you can write an overloaded function that has a O(1) (constant) runtime. It would be defined as follows

```
template<typename T>
void Insert(array<T>& obj,const T& item)
{
 if(obj.size < obj.data.Length())
 {
  obj.data[obj.size] = item;
  obj.size += 1;
 }
}
```

However, a function that inserts at the beginning of the array has a O(n) runtime. It is the worst case scenario of the original function. Its function definition is the definition of the orginal function with the *idx* parameter removed from its header and *idx* replaced with 0 in its body. Furthermore, the second if statement header and its body braces are removed.

Removal Functions

Like the insertion functions, the removal functions can remove values from the array from any valid position in the array. Its definition is as follows

```
template<typename T>
void Remove(array<T>& obj,int idx)
{
 if(obj.size > 0)
 {
  if(idx >= 0 && idx < obj.size)
  {
   obj.size -= 1;

   for(int i = idx;i < obj.size;i += 1)
   {
    obj.data[i] = obj.data[i+1];
   }
  }
 }
}
```

This function is performing a removal by assigning the value of each element starting with the element with the index *idx* the value of its following element. First, the first if statement determines if the array is empty. If there are no elements in the array, there is no reason to continue.

Second, the second if statement determines if *idx* is a valid index of the array. To be considered valid it needs to be equal to an index of any occupied space in the array; hence, it must be between 0 inclusively and *size* exclusively since no element is located at the index *size*.

Third, we decrement *size* by 1 so that we do not get a out of bound error when we traverse the array in the following for loop. Last, the for loop, as stated, assigns each element the value of the element immediately following it starting with the element with index *idx*. For instances, if *data* = [a,b,c,d,e] (*size* = 5) and *idx* = 2, the changes to the array for each run is as follows

Run	Array
1	[a,b,d,d,e]
2	[a,b,d,e,e]

Notice that the last value is still in the array; however, you do not have to worry about it because it is unreachable. When an insertion is perform it will be overridden. This function has a worst-case scenario O(n) runtime where n refers to *size*.

Like with inserting a value at the end of an array, the function to remove a value from the end of an array can be done in O(1) runtime as follows

```
template<typename T>
void Remove(array<T>& obj)
{
 if(obj.size > 0)
 {
  obj.size -= 1;
 }
}
```

Furthermore, the function to remove a value from the beginning of an array has a O(n) runtime and is the worst-case scenario of the original function. The same type of changes you needed to perform for the insertion function must be performed for the removal function.

Searching Functions

It is common to want to search an array either to find the index of a value, to determine if a value is present in an array, or to determine how many instances of a value is present in an array. Currently, since the array is not sorted, the runtime of each of these functions are O(n) where n refers to *size*. In fact, there definitions are all similar.

First, the definition of a function that returns the index of the first occurrence of a value in an array starting at a specific index is as follows

```
template<typename T>
int Search(array<T>& obj,int idx,const T& item)
{
 for(int i = idx;i >= 0 && i < obj.size;i += 1)
 {
  if(obj.data[i] == item)
  {
   return i;
  }
 }
 return -1;
}
```

This function traverses the array starting at *idx*. The condition of the loop is compound to make sure that *idx* is within the proper bound of the array. Similarly, the function that searches the array starting from the first index is as follows

```
template<typename T>
int Search(array<T>& obj,const T& item)
{
 for(int i = 0;i < obj.size;i += 1)
 {
  if(obj.data[i] == item)
  {
   return i;
  }
 }
 return -1;
}
```

This function simply removed the *idx* parameter from the original function, replaced all occurences of *idx* in the body with 0 and simplified the condition because it would be redundant.

Second, sometimes it is neccessary to search an array in reverse. The changes to the reverse search functions only occur in the for loop header as follows

```
template<typename T>
int ReverseSearch(array<T>& obj,int idx,const T& item)
{
 for(int i = obj.size - 1;i >= 0 && i >= idx;i -= 1)
 {
  if(obj.data[i] == item)
  {
   return i;
  }
 }
 return -1;
}
```

```
template<typename T>
int ReverseSearch(array<T>& obj,const T& item)
{
 for(int i = obj.size - 1;i >= 0;i -= 1)
 {
  if(obj.data[i] == item)
  {
   return i;
  }
 }
 return -1;
}
```

The function on the left searches the array in reverse and ends at a specified index *idx*; while, the right function searches the array in reverse and ends at the beginning of the array. For both functions, the loop starts at the last element in the array; and then, the counter is decremented for each instance until it reaches its destination.

Next, sometimes it is necessary to determine if an value is in an array. The definition of the function that does this is identical to the second search function, but the arguments of the return statements become boolean as follows

```
template<typename T>
bool Contains(array<T>& obj,const T& item)
{
 for(int i = 0;i < obj.size;i += 1)
 {
  if(obj.data[i] == item)
  {
   return true;
  }
 }
 return false;
}
```

Once this function finds an element in the array that matches *item*, it returns true. However, if the loop terminates, this means no element matches *item*; thus, the function returns false.

Last, there are times when you may need to count the number of instances of a value that appears in an array. The definition of this function is

```
template<typename T>
bool Count(array<T>& obj,const T& item)
{
 int cnt = 0;

 for(int i = 0;i < obj.size;i += 1)
 {
  if(obj.data[i] == item)
  {
   cnt += 1;
  }
 }
 return cnt;
}
```

The function structure is similar to the previous function `Contains()` except it has a single return statement and a variable for counting named *cnt*. The function begins by initializing *cnt* to 0. Afterwards, it traverses through the loop checking if any element of the array matches *item*. Whenever an element is found, *cnt* is incremented by 1. Last, once the loop terminates, *cnt* is returned. Moverover, the objective of all the above functions is to traverse through an interval of the array and compare its elements with some value.

Resizing Function

Whenever you are dealing with dynamic array problems, you will need to resize the array; however, you must maintain the current data when you do so. A good analogy of this process is moving to a new home. When you are moving, your objects are placed in storage until you can situate in the new place. Once all preparations are completed at the new place, you move your objects from storage into the new place. This is exactly what you do when you resize an array. For clarity let us list the steps.

1. Move objects to a temporary storage.
2. Modify the array [deallocate and reallocate the array].
3. Move objects from the temporary storage to the modified array.

Now, let us look at its implementation where we double the size of the array

```
template<typename T>
void Resize(array<T>& obj)
{
 int sz = obj.data.Length();
 T* tmp = new T[sz];

 for(int i = 0;i < sz;i += 1)
 {
  tmp[i] = obj.data[i];
 }
 obj.data.Resize(sz * 2);

 for(int i = 0; i < sz;i += 1)
 {
  obj.data[i] = tmp[i];
 }
 delete[] tmp;
}
```

First, a temporary array is created with a size equal to the currrent capacity of the array. Next, the first loop copies the data of the array to the temporary array. Then, the array is resized to twice its original capacity. Afterwards, the second loop copies the data of the temporary array to the array. Last, the temporary array is deallocated. Once the function terminates, the array will have the same data as before the execution of the function, but its capacity will be doubled. Last, this function has a $O(n)$ runtime where n refers to *sz*.

Vector Data Structure Object

Currently, we looked at an array data structure that has a fixed size. However, there are situations were you need an array that resizes on the fly so that you do not have to remember to manually do so yourself. This type of data structure is called a *vector*. It has all the functionaities of an array as well as most of the functionalities we discussed in previous sections. The following is a list of the methods of a generic type vector class.

- `Vector()` - it initializes the capacity of the array to 20 and assign 0 to its size.
- `Vector(`*sz*`)` - it initializes size to *sz* and its capacity to twice its size, and assigns the default value of the type to each element of the array upto size.
- `Vector(`*obj*`)` - it performs a deep copy of *obj*.
- `operator=(`*obj*`)` - it performs a deep copy of *obj*.
- ~`Vector()` - it deallocates the array.

- `operator[](idx)` - it returns the element of the array with index *idx* if *idx* is valid; otherwise, it throws an error.
- `Length()` - it returns the size of the array.
- `IsEmpty()` - it returns true if the size of the array is 0; otherwise, it returns false.
- `Insert(idx,itm)` - it inserts *itm* into the array if *idx* is a valid index.
- `Insert(itm)` - it inserts *itm* at the end of the array.
- `Remove(idx)` - it removes the element located at the index *idx* from the array if *idx* is valid.
- `Remove()` - it removes the last element of the array.
- `Search(idx,itm)` - it returns the index of the first element that matches *itm* starting at *idx*; otherwise, it returns the size of the array.
- `Search(itm)` - it returns the index of the first element that matches *itm*; otherwise, it returns the size of the array.
- `Contains(itm)` - it returns true if an element of the array matches *itm*; otherwise, it returns false.
- `ToString()` - it returns a string of the elements of the array separated by commas enclosed in square brackets.
- `operator<<(out,obj)` - it displays the string as `ToString()`.

The vector class also has a private `Resize()` method. It is called in the insertion function whenever the size of the array is equal to the capacity of the array.