Array

Topics:

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

Resources:

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

Introduction

An array is one of the well-known data structures. 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 that stores data in sequence. But first, we 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 array size. 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
- `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.

- `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 a capacity its default value; 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 add a new value to the array considering that the array has a static capacity. The insertion function is going to add a new value to the array; hence, you must initially check if there is room 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* indicates 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 *Id* parameter removed from its header and *id* replaced with 0 in its body. Furthermore, the second if statement header is removed 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 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 will be needed to be perform 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. Current, since the array is not sorted, the runtime of each of these functions are O(n) where n is 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 mak 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.