
Arrays

Data Structures – CS2001 – Fall 2021

Dr. Syed Ali Raza

School of Computer Science

National University of Computing & Emerging Sciences

Karachi Campus

Agenda


- Static Array (Built-in Array)
- Dynamic Array
- DynamicSafeArray class for 1D array
- Exception Handling
- Problem Solving using DynamicSafeArray

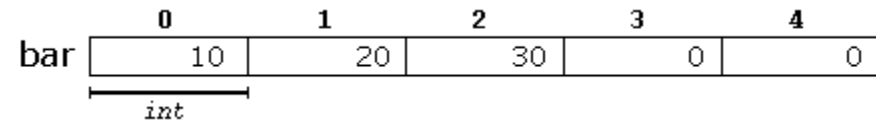
Arrays

- It is used to store elements of same type in contiguous memory locations.
- Contiguous memory locations make it possible to access an element using its position.

Built-in Array (C++)

Different array initializations:

- `int foo [5] = { 16, 2, 77, 40, 12071 };`
- `int bar [5] = { 10, 20, 30 };` 
- `int baz [5] = { };`
- `int foo[] = { 10, 20, 30 };`
- `int foo[] { 10, 20, 30 };`
- `int foo[5]; foo[2] = 75;`
- `Int foo[]; // invalid: array size unknown`



Some valid operations:

```
foo[0] = a;  
foo[a] = 75;  
b = foo [a+2];  
foo[foo[a]] = foo[2] + 5
```

Built-in Array (C++) vs pointers

- An array can be of type pointer so its elements can hold addresses (like a normal pointer).
- An array (say, `int arr[4];`) has a special type which is pointer to whole array. So `&arr + 1` will skip the array and gives address of location next to last element of the array.
- Dynamically allocated array (using pointers) can't be initialized at declaration (`char* cpt=new char[3];` //No way to be initialized here.)
- Their assignments have different assembly codes.

Limitations of Built-in Array (C++)

- Fixed size: Needs size at the time of compilation
- There is no bound checking: It is possible to refer to an element which does not exist in the array by using an index outside the valid range (exceeding the array bounds). It can cause the following problems.
- It may return a wrong (garbage) data which can be harmful for your program.
- Also, it may try to access a segment of memory which is not owned by your program (causes the segmentation fault). For example, it is used by some other application.

Dynamic Array

- Sometimes we don't know the size of an array beforehand. Instead, the number of data elements which are stored in the array become known at the run time.
- For this purpose, static arrays can't be used because the array size is needed at the time of declaration.
- A dynamically-allocated array provides a partial solution.
- In a dynamically-allocated array, memory is allocated dynamically (at run time) using pointer. Such as,

```
double * nums = new double[size];
```

OR

```
double * nums;  
nums = new double[size];
```

Dynamic Array

- Array size can be taken as an input at the run-time and a dynamically-allocated array can be initialized afterwards.
- However, once that array is initialized, it's size is cannot be changed.
- This issue is solved by a dynamic array.
- A dynamic array (or resizable array) is one whose size can be changed dynamically (at the run-time) when needed.

Dynamic Array

- The main idea in a dynamic array is to use a pointer to point to the dynamically-allocated array.
- To expand, replace it with a new dynamically-allocated array with bigger size.
- And make the pointer point to this new array.

Dynamic Array

Dynamic Array ADT:

- Abstract data type with the following operations (at a minimum):
 - Get(i): returns element at location i
 - Set(i, val): Sets element i to val
 - PushBack(val): Adds val to the end
 - Remove(i): Removes element at location i
 - Size(): the number of elements.

Dynamic Array

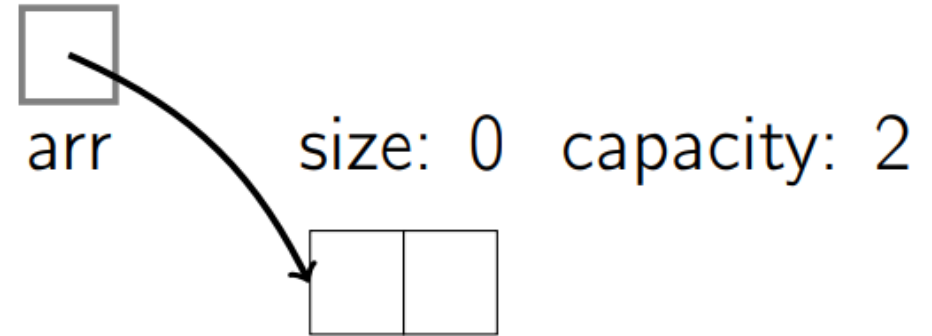
Implementation details:

Data to use:

- **arr**: dynamically-allocated array.
- **capacity**: size of the dynamically-allocated array.
- **size**: number of elements currently in the array.

Dynamic Array Resizing

- Lets say we have a dynamically-allocated array of size zero and capacity of two.
- What happens when the following operations are performed?
 - PuchBack(a)
 - PuchBack(b)
 - PuchBack(c)
 - PuchBack(d)
 - PuchBack(e)



Pseudo Code for PushBack Function

PushBack(val)

if size = capacity:

 allocate new_arr[2 × capacity]

 for i from 0 to size – 1:

 new_arr[i] ← arr[i]

 free arr

 arr ← new_arr; capacity ← 2 × capacity

arr[size] ← val

size ← size + 1

Pseudo Code for Get and Set Functions

Get(i)

if $i < 0$ or $i \geq \text{size}$:

 ERROR: index out of range

return arr[i]

Set(i, val)

if $i < 0$ or $i \geq \text{size}$:

 ERROR: index out of range

arr[i] = val

Pseudo Code for Remove and Size Functions

Remove(i)

if $i < 0$ or $i \geq \text{size}$:

 ERROR: index out of range

for j from i to $\text{size} - 2$:

$\text{arr}[j] \leftarrow \text{arr}[j + 1]$

$\text{size} \leftarrow \text{size} - 1$

Size()

return size

Runtimes

Get(i)	$O(1)$
Set(l, val)	$O(1)$
PushBack(val)	$O(n)$
Remove(i)	$O(n)$
Size()	$O(1)$

Other Member Functions

- Constructor: to initialize the dynamic array using the user's size and a default capacity.
- Copy constructor: to make a copy of the array using another array.
- Assignment operator: to make a copy of the array using another array.
- Access operator: to overload the index operator [] so that we can index our array just like normal arrays. In other words, it must produce a left-hand value as well as a right-hand value such as `int x = da[5];` and `da[6] = 12;`
- Destructor: to clean-up any memory allocation when the object is destroyed.

Alternate Implementation of Access Operator

It is safe as the user's index can be a number larger than current capacity. The array will be expended to accommodate the user's index.

```
int& operator[](int index) {  
    int *new_arr;           // pointer for new array  
    if (index >= capacity) { // is element in the array?  
        new_arr = new int[index + SPARE_CAPACITY]; // allocate a bigger array  
        for (int i = 0; i < size; i++) // copy old values  
            new_arr[i] = arr[i];  
        for (int j = size; j < index + SPARE_CAPACITY; j++) // initialize remainder  
            new_arr[j] = 0;  
        capacity = index + 10; // set length to bigger size  
        delete [] arr; // delete the old array  
        arr = new_arr; // reassign the new array  
    }  
    if (index > size) // set nextIndex past index  
        size = index + 1;  
    return *(arr + index); // a reference to the element  
}
```

Exercise

- Write overriding for the following equality operators
- ==
- !=

Bounds Errors Handling

- Make sure it is safe to refer to an element using an index.
- Make accessing your array safe by checking if the index being used is within the lower and upper bounds of your array.
- The lower bound is zero and the upper bound is size -1.
- If index is exceeding the bounds, throw a 'out of bound' exception.
- Any function which takes index as a parameter and access or modify an element using the index should check for bounds and throw an exception.
- Such functions include get, set, remove, access operator overload, etc.

Bounds Errors Handling

- In the user's code, it is important to handle an exceptions such as 'out of bound.'
- Use try and catch: use try to call functions which throw an exception and catch to catch the exception and report.

```
// remove an element
try{
    array_name.remove(10);
}
catch(const char* msg){
    cout<<msg<<endl;
}
```

Templates in C++

- Template is a format of writing a c++ class or function such that you don't have to write it for different data types.
- Template allows to pass data type as a parameter.
- A function template is a generic function which can be used for different data types.

```
template <typename T>
T sqr(T x) {
    return x*x;
}
```

Templates in C++

- Similarly, a class can be implemented as a template, the passed data type is used to modify the class's implementation to work for that data type.

```
template <typename Type>
class Complex {
    private:
        Type re, im;
    public:
        Complex( Type r=0, Type i=0) {re = r; im = i;}
        Type real() { return re; }
        Type imag() { return im; }
```