Session 4

```
//What are Arrays ? Memory addressing for Arrays{both in Simple program, OOP, Focusing
Memory Addressing }
Program-1
```

```
int main()
{
    int a[10] = { 1,2,3,4 };
    for (int i = 0; i < 10; i++)
        cout << "Array index Member are =" << a[i] << endl;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int a1;
    int a[10] = { 1,2,3,4 };
    for (int i = 0; i < 10; i++)
        cout << "Array index Member are =" << a[i] << endl;
cout << "Reference Address  of a =" << &a << endl;
int &p = a[0];//Take the referece addrress and throug pointer move to the 4th position

cout << "Reference Content  of a =" << p << endl;

cout << "Reference Address  of a =" << &p << endl;

}
```

Sorted Task

```
int main()
{
    int a1,temp;
    temp = 0;
    int i,j,t ;
    int a[10] = { 5,6,7,2,1};
    for (i = 0; i <=4; i++)
    {
        for (j = i+1; j <=4; j++)
        {
            if (a[i] > a[j])
            {
                temp=a[j];
                a[j] = a[i];
                a[i] =temp;
            }

        }

    }

    for (t = 0; t <=4; t++)
    {
        cout << "Sorted String is =" << a[t] << endl;
    }
}
```

Task

```


int* p=a;
    for (t = 0; t <=4; t++)
    {
        cout << "Sorted String is =" << *p++ << endl;

    }
```

```
#include <iostream>
using namespace std;
class Shahbaz {
public :
    Shahbaz() {

    };
//    ~Shahbaz() {};
    int* test()
    {
        int a[]= {5,6,7,2,1};
        int temp;
        temp = 0;
        int i, j, t;
        for (i = 0; i <= 4; i++)
        {

            for (j = i + 1; j <= 4; j++)
            {
                if (a[i] > a[j])
                {
                    temp = a[j];
                    a[j] = a[i];
                    a[i] = temp;
                }

            }

        }
        int* p1 = a;
        for (int i = 0; i <= 4;i++)
        {
            cout <<" Array Value" << a[i] << "Addresss Aspace"<< p1 <<"Address
Content"<<*p1<<endl;
            p1++;
        }
        return a;
    }

};
int main()
{
    Shahbaz *a1=new Shahbaz(); //Pointer type Object
    cout << "Memory Reference" << &a1 << endl;

    //int *p=&a1 //Logical Error
    int *p;
    p=a1->test();
    for (int b = 0; b < 4; b++)
    {
        cout << " " << p << " "<< endl;
            p++;
    }

}
```

Limitation of Built-in Arrays in C/C++, Index bounding, safe access, fixed length declaration etc.

```
int main()
{

int b1[2] = { 0,1};
cout << b1[-1];
}
```

No error

```
int main()
{

    Shahbaz *a1=new Shahbaz(); //Pointer type Object
    int b1[2] = { 0,1,3,4,5};
cout << b1[-1];
}
```

This is also run

Pointer vs. arrays

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
   int  var[MAX] = {10, 100, 200};
   int  *ptr;

   // let us have array address in pointer.
   ptr = var;

   for (int i = 0; i < MAX; i++) {
      cout << "Address of var[" << i << "] = ";
      cout << ptr << endl;

      cout << "Value of var[" << i << "] = ";
      cout << *ptr << endl;

      // point to the next location
      ptr++;
   }

   return 0;
}
```

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
   int  var[MAX] = {10, 100, 200};

   for (int i = 0; i < MAX; i++) {
      *var = i;     // This is a correct syntax
      var++;        // This is incorrect.
   }

   return 0;
}
```

Defining class of DynamicSafeArray in one dimension

Vector

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array. Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

Iterators

1. begin() – Returns an iterator pointing to the first element in the vector
2. end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector
3. rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
4. rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)
5. cbegin() – Returns a constant iterator pointing to the first element in the vector.
6. cend() – Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.
7. crbegin() – Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
8. crend() – Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

```cpp
#include <iostream>
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Output of begin and end: ";
    for (auto i = g1.begin(); i != g1.end(); ++i)
        cout << *i << " ";

    cout << "\nOutput of cbegin and cend: ";
    for (auto i = g1.cbegin(); i != g1.cend(); ++i)
        cout << *i << " ";

    cout << "\nOutput of rbegin and rend: ";
    for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)
        cout << *ir << " ";

    cout << "\nOutput of crbegin and crend : ";
    for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)
        cout << *ir << " ";

    return 0;
}
```

Output:

```
Output of begin and end: 1 2 3 4 5

Output of cbegin and cend: 1 2 3 4 5

Output of rbegin and rend: 5 4 3 2 1

Output of crbegin and crend : 5 4 3 2 1
```

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Size : " << g1.size();
    cout << "\nCapacity : " << g1.capacity();
    cout << "\nMax_Size : " << g1.max_size();

    // resizes the vector size to 4
    g1.resize(4);

    // prints the vector size after resize()
    cout << "\nSize : " << g1.size();

    // checks if the vector is empty or not
    if (g1.empty() == false)
        cout << "\nVector is not empty";
    else
        cout << "\nVector is empty";

    // Shrinks the vector
    g1.shrink_to_fit();
    cout << "\nVector elements are: ";
    for (auto it = g1.begin(); it != g1.end(); it++)
        cout << *it << " ";

    return 0;

}
```

Output:

```
Size : 5

Capacity : 8

Max_Size : 4611686018427387903

Size : 4

Vector is not empty

Vector elements are: 1 2 3 4
```

Element access:

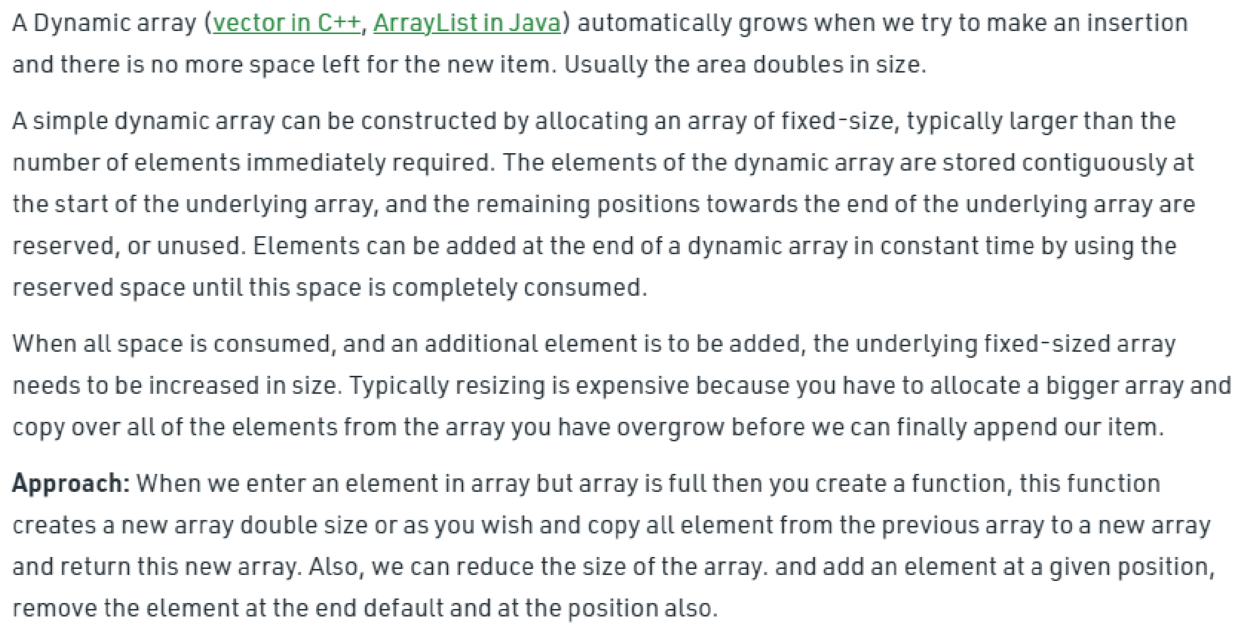
1. reference operator [g] – Returns a reference to the element at position 'g' in the vector
2. at(g) – Returns a reference to the element at position 'g' in the vector
3. front() – Returns a reference to the first element in the vector
4. back() – Returns a reference to the last element in the vector
5. data() – Returns a direct pointer to the memory array used internally by the vector to store its owned elements.

```
#include <vector>

using namespace std;
int main()
{
    vector<int> g1;

    for (int i = 1; i <= 10; i++)
        g1.push_back(i * 10);

    cout << "\nReference operator [g] : g1[2] = " << g1[2];

    cout << "\nat : g1.at(4) = " << g1.at(4);

    cout << "\nfront() : g1.front() = " << g1.front();

    cout << "\nback() : g1.back() = " << g1.back();

    // pointer to the first element
    int* pos = g1.data();

    cout << "\nThe first element is " << *pos;
    return 0;
}
```

```
Reference operator [g] : g1[2] = 30
at : g1.at(4) = 50
front() : g1.front() = 10
back() : g1.back() = 100
The first element is 10
```

Modifiers:

1. assign() – It assigns new value to the vector elements by replacing old ones
2. push_back() – It push the elements into a vector from the back
3. pop_back() – It is used to pop or remove elements from a vector from the back.
4. insert() – It inserts new elements before the element at the specified position
5. erase() – It is used to remove elements from a container from the specified position or range.
6. swap() – It is used to swap the contents of one vector with another vector of same type. Sizes may differ.
7. clear() – It is used to remove all the elements of the vector container
8. emplace() – It extends the container by inserting new element at position
9. emplace_back() – It is used to insert a new element into the vector container, the new element is added to the end of the vector

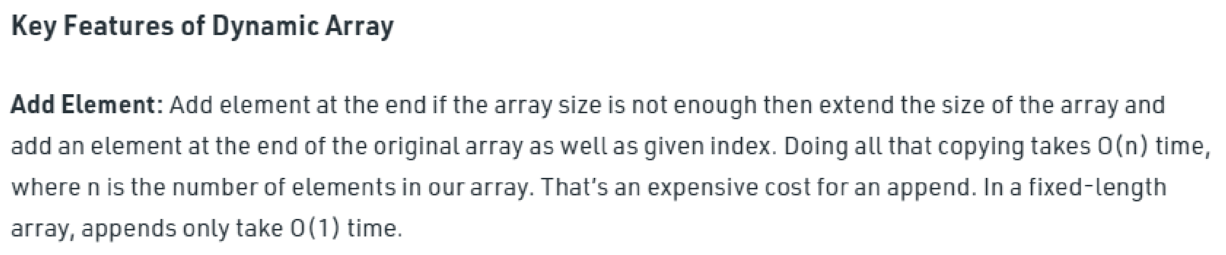


A Dynamic array (vector in C++, ArrayList in Java) automatically grows when we try to make an insertion and there is no more space left for the new item. Usually the area doubles in size.

A simple dynamic array can be constructed by allocating an array of fixed-size, typically larger than the number of elements immediately required. The elements of the dynamic array are stored contiguously at the start of the underlying array, and the remaining positions towards the end of the underlying array are reserved, or unused. Elements can be added at the end of a dynamic array in constant time by using the reserved space until this space is completely consumed.

When all space is consumed, and an additional element is to be added, the underlying fixed-sized array needs to be increased in size. Typically resizing is expensive because you have to allocate a bigger array and copy over all of the elements from the array you have overgrow before we can finally append our item.

Approach: When we enter an element in array but array is full then you create a function, this function creates a new array double size or as you wish and copy all element from the previous array to a new array and return this new array. Also, we can reduce the size of the array. and add an element at a given position, remove the element at the end default and at the position also.

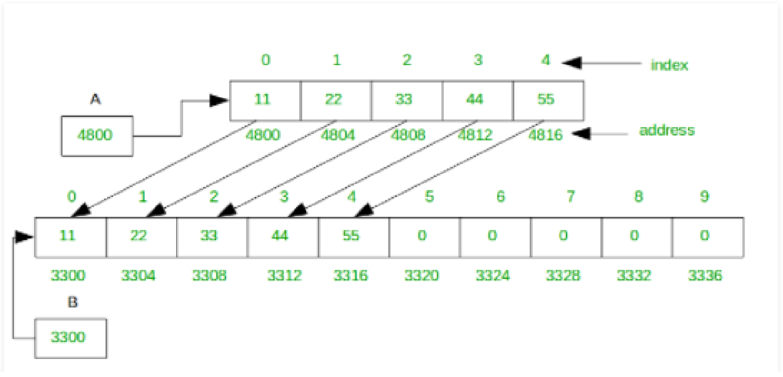


Key Features of Dynamic Array

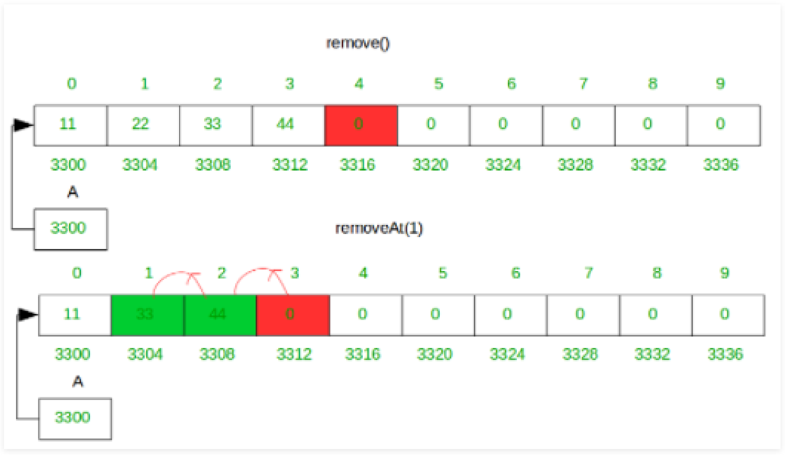
Add Element: Add element at the end if the array size is not enough then extend the size of the array and add an element at the end of the original array as well as given index. Doing all that copying takes O(n) time, where n is the number of elements in our array. That's an expensive cost for an append. In a fixed-length array, appends only take O(1) time.

But appends take O(n) time only when we insert into a full array. And that is pretty rare, especially if we double the size of the array every time we run out of space. So in most cases appending is still O(1) time, and sometimes it's O(n) time.

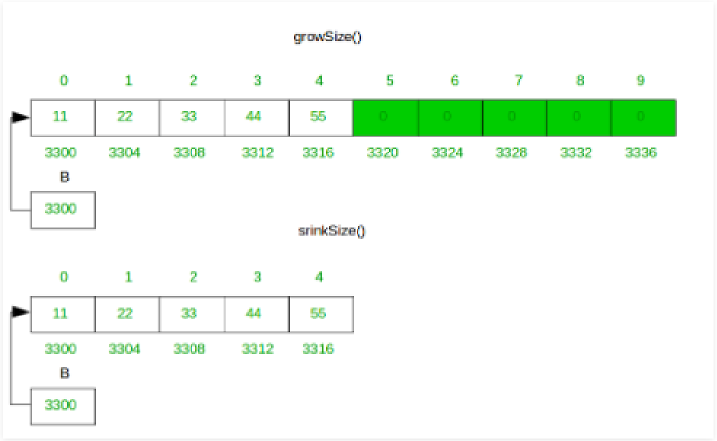
In dynamic array you can create fixed-size array when required added some more element in array then use this approach:



Delete Element: Delete an element from array, default remove() method delete an element from end, simply store zero at last index and you also delete element at specific index by calling removeAt(i) method where i is index. removeAt(i) method shift all right element in the left side from the given index.



Resize of Array Size: When the array has null/zero data(exclude add by you) at the right side of the array whose take unrequited memory, the method srinkSize() free extra memory. When all space is consumed, and an additional element is to be added, then the underlying fixed-size array needs to increase size. Typically resizing is expensive because you have to allocate a bigger array and copy over all of the elements from the array you have overgrow before we can finally append our item.



Simple code for dynamic array. In below code then the array will become full of size we copy all element to new double size array(variable size array).sample code is below

```
#include <cstdlib>
#include<string.h>
using namespace std;
class atype {
       int ncols;
       int* dynamicArray;
public:          atype()
{
       ncols = 0;
       dynamicArray = new int[ncols];
}
         atype(int col) {
                ncols = col;
                dynamicArray = new int[ncols];
         }
         ~atype() {
                delete[] dynamicArray;
         }

//user inserting elements in 2d array
     fillArray()  {
                    (     in=0;in<ncols;++in)
                {              value;
                cout<<"enter value";
                cin>>value;
                dynamicArray[in] = value;
                }
////bound checking-safe array implementation
int &operator [](int i)
          {
                 if(i<0 || i>  ncols-1 )
                 {
                        cout << "Boundary Error\n";
                 exit(1);
}
                 return dynamicArray[i];
          }

////////////////////////////////////////////////////////////////////////////////////
atype& operator=(const atype& rhs)    //assignment operator
          {
                 if (this == &rhs)
                        return *this;
                 delete[] dynamicArray;
          ncols = rhs.ncols;
          dynamicArray = new int[ncols];
          memcpy(dynamicArray, rhs.dynamicArray, sizeof(int)* ncols);
          return *this;

}


};


int main() {
       int columns;
       cout << "enter cols" << endl;
       cin >> columns;
       atype ob1(columns);
       ob1.fillArray();
       atype ob2 = ob1;
       atype ob3;
       ob3 = ob1;
       cout << ob1[1] << endl;
       cout << ob1[2] << endl;     //checking bounds of array

}
```