Data Structures-LAB

Lab Manual # 02

Pointers, DMA and List Data Structure

Instructor: Engr. Khuram Shahzad

Semester Spring, 2022

Course Code: CL2001

Fast National University of Computer and Emerging Sciences Peshawar

Department of Computer Science

Data Structures-LAB

Table of Contents

C++ Dynamic Memory

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts −

- **The stack** − All variables declared inside the function will take up memory from the stack.
- **The heap** − This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory that was previously allocated by new operator.

new and delete operators in C++ for dynamic memory

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on Heap and non-static and local variables get memory allocated on Stack (Refer Memory Layout C Programs for details).
What are applications?

- One use of dynamically allocated memory is to allocate memory of variable size which is not possible with compiler allocated memory except variable length arrays.
- The most important use is flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need and whenever we don't need anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, Tree, etc.

How is it different from memory allocated to normal variables?
For normal variables like "int a", "char str[10]", etc, memory is automatically allocated and deallocated. For dynamically allocated memory like "int *p = new int[10]", it is programmers responsibility to deallocate memory when no longer needed. If programmer doesn't deallocate memory, it causes memory leak (memory is not deallocated until program terminates).
How is memory allocated/deallocated in C++?
C uses malloc() and calloc() function to allocate memory dynamically at run time and

uses free() function to free dynamically allocated memory. C++ supports these functions and also has two operators new and delete that perform the task of allocating and freeing the memory in a better and easier way.
This is all about new and delete operators.

new operator

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

- Syntax to use new operator: To allocate memory of any data type, the syntax is:

pointer-variable = new data-type;

- Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class. Example:

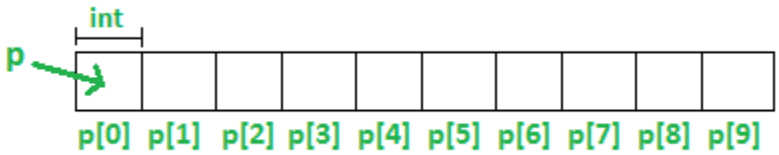
```
// Then request memory for the variable
int* p = NULL;
p = new int;

            OR

// Combine declaration of pointer
// and their assignment
int* p = new int;
```

Initialize memory: We can also initialize the memory for built-in data types using new operator. For custom data types a constructor is required (with the data-type as input) for initializing the value. Here's an example for the initialization of both data types :
pointer-variable = new data-type(value);

```
Example:
int* p = new int(25);
float* q = new float(75.25);

// Custom data type
struct cust
{
    int p;
    cust(int q) : p(q) { }
};

cust* var1 = new cust;    // Works fine, doesn't require constructor
OR
cust* var1 = new cust();        // Works fine, doesn't require
constructor

cust* var = new cust(25)        // Notice error if you comment this
line
```

int *p = new int[10]

- Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.



Normal Array Declaration vs Using new

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

What if enough memory is not available during runtime?

If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type std::bad_alloc, unless "nothrow" is used with the new operator, in which case it returns a NULL pointer (scroll to section "Exception handling of new operator" in this article). Therefore, it may be good idea to check for the pointer variable produced by new before using it program.

```
int* p = new (nothrow) int;
if (!p)
{
   cout << "Memory allocation failed\n";
}
```

Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.
Syntax:
// Release memory pointed by pointer-variable

delete pointer-variable;
Here, pointer-variable is the pointer that points to the data object created by new.
Examples:

delete p;

delete q;

To free the dynamically allocated array pointed by pointer-variable, use following form of delete:

```
// Release block of memory
// pointed by pointer-variable
Delete [] pointer-variable;
```

Example:

```
Example:
   // It will free the entire array
   // pointed by p.
   Delete [] p;
```

```cpp
// C++ program to illustrate dynamic allocation
// and deallocation of memory using new and delete
#include <iostream>
using namespace std;
int main()
{
    // Pointer initialization to null
    int* p = NULL;

    // Request memory for the variable
    // using new operator
    p = new (nothrow) int;
    if (!p)
        cout << "allocation of memory failed\n";
    else
    {
        // Store value at allocated address
        *p = 29;
        cout << "Value of p: " << *p << endl;
    }
    // Request block of memory
    // using new operator
    float* r = new float(75.25);

    cout << "Value of r: " << *r << endl;
    // Request block of memory of size n
    int n = 5;
    int* q = new (nothrow) int[n];

    if (!q)
        cout << "allocation of memory failed\n";
    else
    {
        for (int i = 0; i < n; i++)
            q[i] = i + 1;
        cout << "Value store in block of memory using index: ";
        for (int i = 0; i < n; i++)
            cout << q[i] << " ";
        cout << endl << "Value store in block of memory using ptr: ";

        for (int i = 0; i < n; i++)
        {
            cout << *q << " ";
            q = q + 1;
        }
    }
    // freed the allocated memory
    delete p;
    delete r;
    // freed the block of allocated memory
    delete[] q;
    return 0;
}
```

Output:

Value of p: 29

Value of r: 75.25

Value store in block of memory: 1 2 3 4 5

Dynamic Memory Allocation for Objects

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept −

```
#include <iostream>
using namespace std;
class Box
{
    public:
      Box()
    {
        cout << "Constructor called!" << endl;
    }
    ~Box()
    {
        cout << "Destructor called!" << endl;
    }
};
int main()
{
    Box* myBoxArray = new Box[4];
    delete[] myBoxArray; // Delete array

    return 0;
}
```

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times. If we compile and run above code, this would produce the following result −

```
Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
Destructor called!
Destructor called!
Destructor called!
```

C++ new and delete Operator for Arrays

```
// C++ Program to store GPA of n number of students and display it
// where n is the number of students entered by the user
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "Enter total number of students: ";
    cin >> num;
    float* ptr;
    // memory allocation of num number of floats
    ptr = new float[num];

    cout << "Enter GPA of students." << endl;
    for (int i = 0; i < num; ++i)
    {
        cout << "Student" << i + 1 << ": ";
        cin >> *(ptr + i);
    }
    cout << "\nDisplaying GPA of students." << endl;
    for (int i = 0; i < num; ++i)
    {
        cout << "Student" << i + 1 << " :" << *(ptr + i) << endl;
    }
    // ptr memory is released
    delete[] ptr;
    return 0;
}
```

Output

```
Enter total number of students: 4
Enter GPA of students.
Student1: 3.6
Student2: 3.1
Student3: 3.9
Student4: 2.9
Displaying GPA of students.
Student1 :3.6
Student2 :3.1
Student3 :3.9
```

```
Student4 :2.9
```

C++ new and delete Operator for Objects

```
#include <iostream>
using namespace std;
class Student
{
    int age;
    public:
    // constructor initializes age to 12
    Student() : age(12) { }

    void getAge()
    {
        cout << "Age = " << age << endl;
    }
};

int main()
{
    // dynamically declare Student object
    Student* ptr = new Student();
    // call getAge() function
    ptr->getAge();
    // ptr memory is released
    delete ptr;
    return 0;
}
```

Output

```
Age = 12
```

In this program, we have created a Student class that has a private variable age.

We have initialized age to 12 in the default constructor Student() and print its value with the function getAge().

In main(), we have created a Student object using the new operator and use the pointer ptr to point to its address.

The moment the object is created, the Student() constructor initializes age to 12.

We then call the getAge() function using the code:

C++ Dynamic Memory Allocation Example Arrays

```
/* C++ Dynamic Memory Allocation Example Program */
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;
int* rollno;    // declares an integer pointer
float* marks;   // declares a float pointer

int main()
{
    int size, i;
    cout << "How many elements for the array ? ";
    cin >> size;
    rollno = new int[size];    // dynamically allocate rollno array
    marks = new float[size];        // dynamically allocate marks
array
    // first check, whether the memory is available or not
    if ((!rollno) || (!marks))       // if rollno or marks is null
pointer
    {
        cout << "Out of Memory..!!..Aborting..!!\n";
        cout << "Press any key to exit..";
        getch();
        exit(1);
    }
    // read values in the array elements
    for (i = 0; i < size; i++)
    {
        cout << "Enter rollno and marks for student " << (i + 1) <<
"\n";
        cin >> rollno[i] >> marks[i];
    }
    // now display the array contents
    cout << "\nRollNo\t\tMarks\n";
    for (i = 0; i < size; i++)
    {
        cout << rollno[i] << "\t\t" << marks[i] << "\n";
    }
```

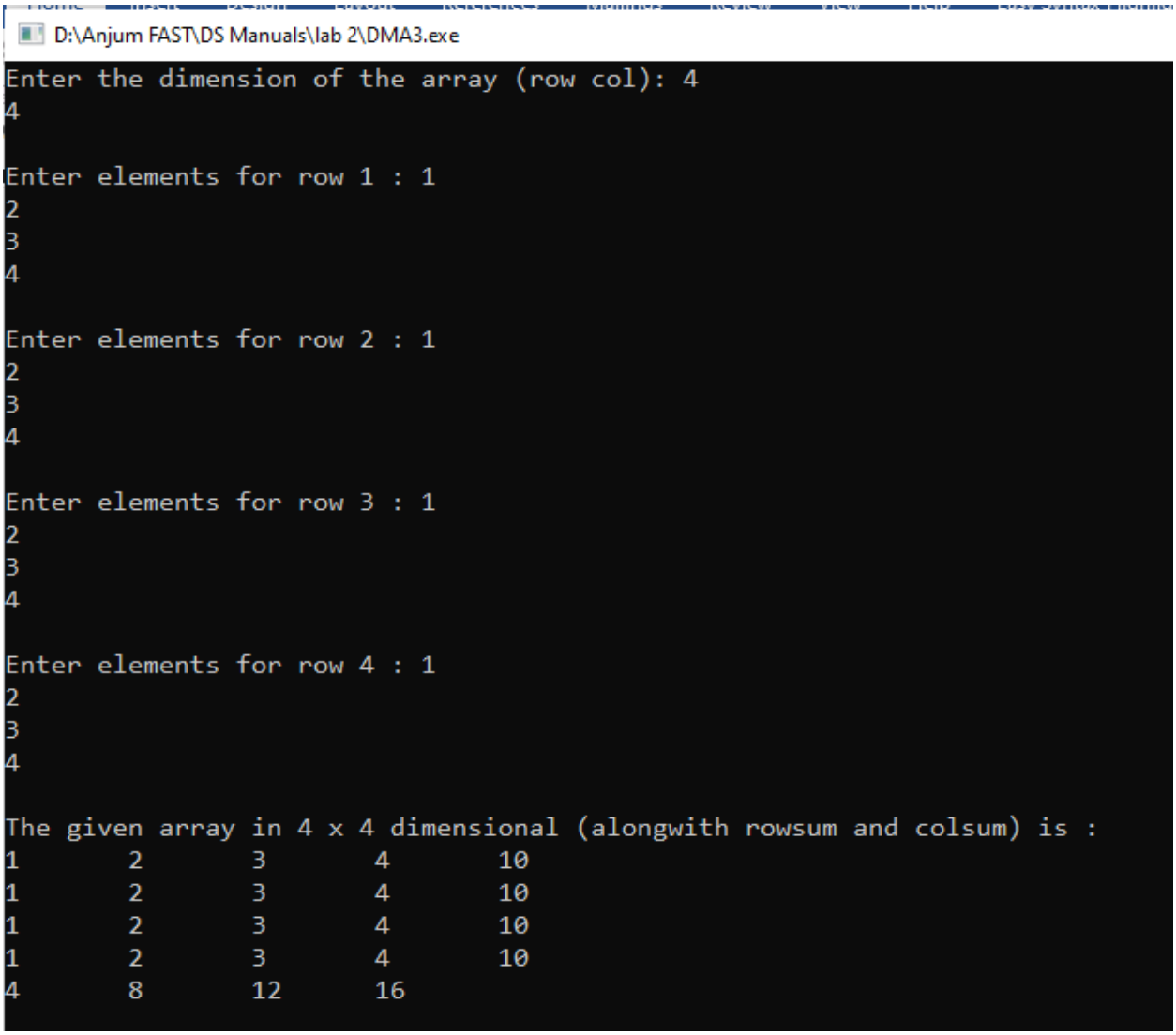
```
    delete[] rollno;    // deallocating rollno array
    delete[] marks;     // deallocating marks array
    getch();

}
```

C++ Dynamic Memory Allocation Example Multi-Dimensional Arrays

```
/* C++ Dynamic Memory Allocation Example Program
 * This is the same program as above, but this
 * program uses two-dimensional array to demonstrates
 * dynamic memory allocation in C++. This C++ program
 * also displays the rowsum and the colsum of the array */

#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;
int main()
{

    int* val, *rows, *cols;
    int maxr, maxc, i, j;
    cout << "Enter the dimension of the array (row col): ";
    cin >> maxr >> maxc;
    val = new int[maxr * maxc];
    rows = new int[maxr];
    cols = new int[maxc];

    for (i = 0; i < maxr; i++)
    {
        cout << "\nEnter elements for row " << i + 1 << " : ";
        rows[i] = 0;
        for (j = 0; j < maxc; j++)
        {
            cin >> val[i * maxc + j];
            rows[i] = rows[i] + val[i * maxc + j];
        }
    }

    for (j = 0; j < maxc; j++)
    {
        cols[j] = 0;
        for (i = 0; i < maxr; i++)
        {
            cols[j] = cols[j] + val[i * maxc + j];
        }
```

```
    }

    cout << "\nThe given array in 2 x 2 dimensional (alongwith rowsum and
colsum) is :\n";
    for (i = 0; i < maxr; i++)
    {
        for (j = 0; j < maxc; j++)
        {
            cout << val[i * maxc + j] << "\t";
        }
        cout << rows[i] << "\n";
    }

    for (j = 0; j < maxc; j++)
    {
        cout << cols[j] << "\t";
    }
    cout << "\n";
    delete[] val;
    delete[] rows;
    delete[] cols;
    getch();
    return 0;
}
```

```
}
```



Dynamic memory allocation in C++ for 2D and 3D array

This post will discuss dynamic memory allocation in C++ for multidimensional arrays.

1. Single Dimensional Array

```
#include <iostream>
#define N 10

// Dynamically allocate memory for 1D Array in C++
int main()
{
    // dynamically allocate memory of size `N`
    int* A = new int[N];
```

```
    // assign values to the allocated memory
    for (int i = 0; i < N; i++)
    {
        A[i] = i + 1;
    }

    // print the 1D array
    for (int i = 0; i < N; i++)
    {
        std::cout << A[i] << " ";   // or *(A + i)
    }

    // deallocate memory
    delete[] A;

    return 0;
}
```

2. 2-Dimensional Array

1. Using Single Pointer

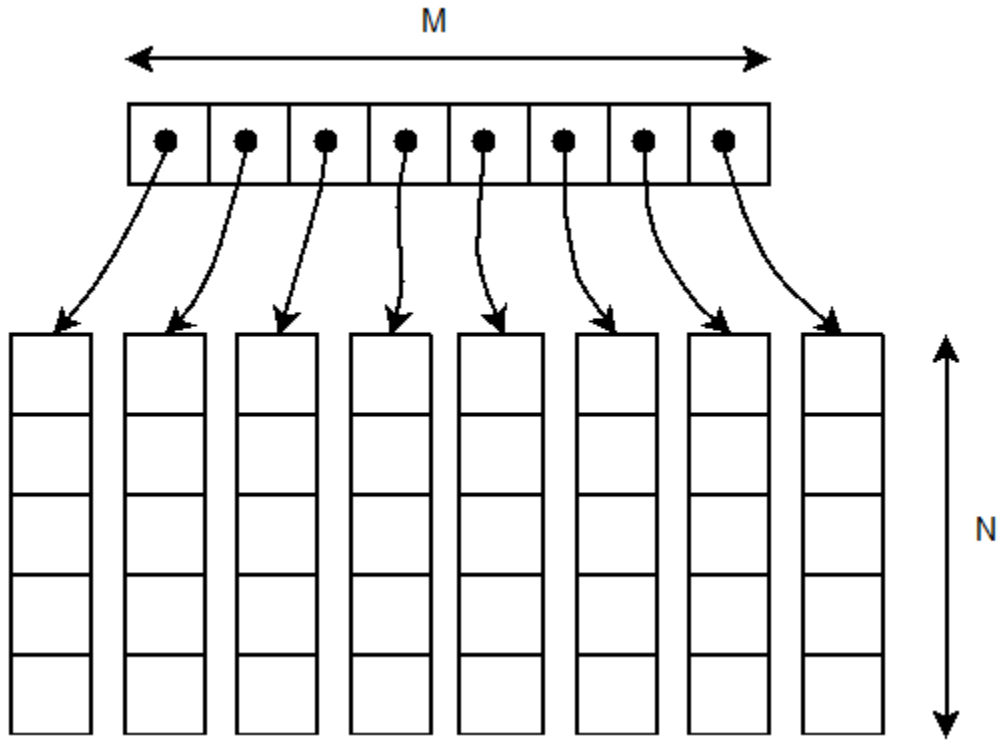
In this approach, we simply allocate one large block of memory of size `M × N` dynamically and assign it to the pointer. Then we can use pointer arithmetic to index the 2D array.

```
#include <iostream>

// `M Ã— N` matrix
#define M 4
```

```
#define N 5

// Dynamically allocate memory for 2D Array in C++
int main()
{
    // dynamically allocate memory of size `M Ã— N`
    int* A = new int[M * N];

    // assign values to the allocated memory
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
            *(A + i * N + j) = rand() % 100;
        }
    }

    // print the 2D array
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
            std::cout << *(A + i * N + j) << " ";       // or (A + i*N)[j])
        }
        std::cout << std::endl;
    }

    // deallocate memory
    delete[] A;

    return 0;
}
```

2. Using array of Pointers

We can dynamically create an array of pointers of size `M` and then dynamically allocate memory of size `N` for each row, as shown below:




```
#include <iostream>

// `M Ã— N` matrix
#define M 4
#define N 5

// Dynamic Memory Allocation in C++ for 2D Array
int main()
{
    // dynamically create an array of pointers of size `M`
    int** A = new int*[M];

    // dynamically allocate memory of size `N` for each row
    for (int i = 0; i < M; i++)
    {
        A[i] = new int[N];
    }

    // assign values to the allocated memory
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
            A[i][j] = rand() % 100;
        }
    }
```

```
    // print the 2D array
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
            std::cout << A[i][j] << " ";
        }
        std::cout << std::endl;
    }

    // deallocate memory using the delete[] operator
    for (int i = 0; i < M; i++)
    {
        delete[] A[i];
    }
    delete[] A;

    return 0;
}
```

3. 3-Dimensional Array

1. Using Single Pointer

As seen for the 2D array, we allocate memory of size $X \times Y \times Z$ dynamically and assign it to a pointer. Then we use pointer arithmetic to index the 3D array.

```
#include <iostream>
// `X Ã— Y Ã— Z` matrix
#define X 2
#define Y 3
#define Z 4

// Dynamic Memory Allocation in C++ for 3D Array
int main()
{
    // dynamically allocate memory of size `X Ã— Y Ã— Z`
    int* A = new int[X * Y * Z];

    // assign values to the allocated memory
    for (int i = 0; i < X; i++)
    {
        for (int j = 0; j < Y; j++)
        {
            for (int k = 0; k < Z; k++)
            {
                *(A + i * Y * Z + j * Z + k) = rand() % 100;
            }
        }
```

```
    }

    // print the 3D array
    for (int i = 0; i < X; i++)
    {
        for (int j = 0; j < Y; j++)
        {
            for (int k = 0; k < Z; k++)
            {
                std::cout << *(A + i * Y * Z + j * Z + k) << " ";
            }
            std::cout << std::endl;
        }
        std::cout << std::endl;
    }
    // deallocate memory
    delete[] A;

    return 0;
}
```

2. Using Triple Pointer

```
#include <iostream>
// `X Ã– Y Ã– Z` matrix
#define X 2
#define Y 3
#define Z 4
// Dynamically allocate memory for 3D Array in C++
int main()
{
    int*** A = new int**[X];

    for (int i = 0; i < X; i++)
    {
        A[i] = new int*[Y];
        for (int j = 0; j < Y; j++)
        {
            A[i][j] = new int[Z];
        }
    }

    // assign values to the allocated memory
    for (int i = 0; i < X; i++)
    {
        for (int j = 0; j < Y; j++)
        {
            for (int k = 0; k < Z; k++)
            {
                A[i][j][k] = rand() % 100;
```

```
            }
        }
    }

    // print the 3D array
    for (int i = 0; i < X; i++)
    {
        for (int j = 0; j < Y; j++)
        {
            for (int k = 0; k < Z; k++)
            {
                std::cout << A[i][j][k] << " ";
            }
            std::cout << std::endl;
        }
        std::cout << std::endl;
    }

    // deallocate memory
    for (int i = 0; i < X; i++)
    {
        for (int j = 0; j < Y; j++)
        {
            delete[] A[i][j];
            }
            delete[] A[i];
      }

      delete[] A;

      return 0;
}
```

C++ program for array implementation of List ADT

Concept: A linked list is a sequence of data structures, which are connected together via links. Linked List is a sequence of links which contains items. Each link contains a

connection to another link. Linked list is the second most-used data structure after array. A linked list is a sequence of data structures, which are connected together via links. Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

• **Link** − each link of a linked list can store a data called an element.

• **Next** − each link of a linked list contains a link to the next link called Next.

• **Linked List** − A Linked List contains the connection link to the first link called First.

```
#include<iostream>
#include<conio.h>
#include<process.h>

int length = 0;
void create();
void insert();
void deletion();
void search();
void display();
int size();
int a, b[20], n, d, e, f, i;
using namespace std;
int main()
{
    int c;

    cout << "\n Main Menu";
    cout << "\n 1.Create \n 2.Delete \n 3.Search \n 4.insert \n 5.Display \n
6.Exit";
    do
    {
        cout << "\n enter your choice:";
        cin >> c;
        switch (c)
        {
            case 1:
                create();
                break;
            case 2:
                deletion();
                break;
            case 3:
                search();
                break;
            case 4:
```

```
                insert();
                break;
            case 5:
                display();
                break;
            case 6:
                exit(0);
                break;
            default:
                cout << "The given number is not between 1-5\n";
        }
    } while (c <= 6);
    getch();
}
void create()
{
    cout << "\n Enter the number of elements you want to create: ";
    cin >> n;
    length = n;
    cout << "\nenter the elements\n";
    for (i = 0; i < n; i++)
    {
        cin >> b[i];
    }
}

void deletion()
{
    if (length == 0)
        cout << "Array Empty, Please Initilized it first.";
    else
    {

        cout << "Enter the number u want to delete \n";
        cin >> d;
        for (i = 0; i < n; i++)
        {
            if (b[i] == d)
            {
                b[i] = 0;
                cout << d << " deleted";
                length--;
            }
        }
    }
}
void search()
{
    cout << "Enter the number \n";
    cin >> e;
```

```
    for (i = 0; i < n; i++)
    {
        if (b[i] == e)
        {
            cout << "Value found the position\n" << i + 1;
        }
        else
        {
            cout << "The Value " << e << " found the position\n";
        }

    }
}
void insert()
{
    cout << "\nenter how many number u want to insert: ";
    cin >> f;
    cout << "\nEnter the elements\n";
    for (i = 0; i < f; i++)
    {
        cin >> b[n++];
        length++;
    }
}
void display()
{
    for (i = 0; i < n; i++)
    {
        cout << "\n" << b[i];
    }
}

int size()
{
    return length;
}
```