Object Oriented Programming - Lab

LAB 02

Topic: Dynamic Memory Allocation in C++

Course Code: CL1004

Semester: Spring 2025

Instructor: Engr. Muhammad Qasim

Email: Muhammad.qasim@nu.edu.pk

Department of Computer Science, National University of Computer and Emerging Sciences FAST Peshawar

Memory Allocation

There are essentially two types of memory allocation:

1. Static - Done by the compiler automatically (implicitly).
2. Dynamic - Done explicitly by the programmer.

Static:

• Global variables or objects (memory is allocated at the start of the program, and freed when program exits; alive throughout program execution and can be access anywhere in the program)

• Local variables (memory is allocated when the routine starts and freed when the routine returns and cannot be accessed from another routine)

• Allocation and free are done implicitly. No need to explicitly manage memory is nice (easy to work with), but has limitations!

Dynamic:

1. Programmer explicitly requests the system to allocate memory and return starting address of memory allocated (what is this?). This address can be used by the programmer to access the allocated memory.

2. When done using memory, it must be explicitly freed.

Dynamic Memory Allocation:

Dynamic allocation requires two steps:

1. Creating the dynamic space.

2. Storing its address in a pointer (so that the space can be accessed)

To dynamically allocate memory in C++, we use the **new** operator.

De-allocation:

o Deallocation is the "clean-up" of space being used for variables or other data storage

o Compile time variables are automatically deallocated based on their known extent (this is the same as scope for "automatic" variables)

o It is the programmer's job to deallocate dynamically created space

o To de-allocate dynamic memory, we use the **delete** operator

Allocating space with new: To allocate space dynamically, use the unary operator new, followed by the type being allocated. new int; // dynamically allocates an int

new double; // dynamically allocates a double

If creating an array dynamically, use the same form, but put brackets with a size after the type: **new int[40];** // dynamically allocates an array of 40

int new double[size]; // dynamically allocates an array of size double and the size can be a variable

These statements above are not very useful by themselves, because the allocated spaces have no names! BUT, the new operator returns the starting address of the allocated space, and this address can be stored in a pointer:

int * p; // declare a pointer p

p = new int; // dynamically allocate an int and load address into p

double * d; // declare a pointer d

d = new double; // dynamically allocate a double and load address into d

// we can also do these in single line statements

int x = 40;

int * list = new int[x];

float * numbers = new float[x+10];

Accessing dynamically created space:

So once the space has been dynamically allocated, how do we use it? For single items, we go through the pointer. Dereference the pointer to reach the dynamically created target:

int * p = new int;

***p = 10;**

cout << *p; // dynamic integer, pointed to by p // assigns 10 to the dynamic integer

For dynamically created arrays, you can use either pointer-offset notation, or treat the pointer as the array name and use the standard bracket notation:

double * numList = new double[size];

for (int i = 0; i < size; i++)   // initialize array elements to 0

numList[i] = 0;

numList[5] = 20;   // bracket notation

***(numList + 7) = 15;   // pointer-offset notation. means same as numList[7]**

Deallocation of dynamic memory

To deallocate memory that was created with new, we use the unary operator delete. The one operand should be a pointer that stores the address of the space to be deallocated:

```
int * ptr = new int; // dynamically created int // ...

 delete ptr; // deletes the space that ptr points to
```

Note that the pointer ptr still exists in this example. That's a named variable subject to scope and extent determined at compile time. It can be reused:

ptr = new int[10]; // point p to a brand new array

To deallocate a dynamic array, use this form:

```
delete [] name_of_pointer;
```

Example:

```
 int * list = new int[40]; // dynamic array

delete [] list; // deallocates the array

list = 0; // reset list to null pointer
```

After deallocating space, it's always a good idea to reset the pointer to null unless you are pointing it at another valid target right away.

To consider: So what happens if you fail to deallocate dynamic memory when you are finished with it? (i.e. why is deallocation important?)

Example

```
#include <iostream>

using namespace std;

int main () {

double* pvalue  = NULL; // Pointer initialized with null

pvalue  = new double;   // Request memory for the variable


   *pvalue = 29494.99;     // Store value at allocated address

   cout << "Value of pvalue : " << *pvalue << endl;
   delete pvalue;         // free up the memory.

   return 0;
}
```

Output:

```
Value of pvalue : 29495
```

Example

```
// Dynamically Allocate Memory for 1D Array in C++

#include <iostream>

#define N 10

int main()
{
 // dynamically allocate memory of size N
 int* A = new int[N];

 // assign values to 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;
}
```

Output: 1 2 3 4 5 6 7 8 9 10

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.

Application Example: Dynamically resizing an array

If you have an existing array, and you want to make it bigger (add array cells to it), you cannot simply append new cells to the old ones. Remember that arrays are stored in consecutive memory, and you never know whether or not the memory immediately after the array is already allocated for something else.   For that reason, the process takes a few more steps. Here is an example using an integer array. Let's say this is the original array:

```
int * list = new int[size];
```

I want to resize this so that the array called list has space for 5 more numbers (presumably because the old one is full). There are four main steps.

1. Create an entirely new array of the appropriate type and of the new size. (You'll need

another pointer for this).

```
int * temp = new int[size + 5];
```

2. Copy the data from the old array into the new array (keeping them in the same positions).

This is easy with a for-loop.

```
for (int i = 0; i < size; i++)
temp[i] = list[i];
```

3. Delete the old array -- you don't need it anymore! (Do as your Mom says, and take out

the garbage!)

```
delete [] list;  // this deletes the array pointed to by "list"
```

4. Change the pointer. You still want the array to be called "list" (its original name), so

change the list pointer to the new address.

```
list = temp;
```

That's it! The list array is now 5 larger than the previous one, and it has the same data in it that the original one had. But, now it has room for 5 more items.

Example

```
#include<iostream>

using namespace std;

int main()

{

int max = 5;           // no longer const

int* a = new int[max];  // allocated on heap

int n = max;

char check;

//--- Read into the array

 cout<<"Enter 5 values: "<<endl;

 for(int i= 0; i<max; i++)

 cin>>a[i];


 do{

  cout<<endl<<"Wanna enter more values??(y/n)  ";

  cin>>check;


             if (check == 'y' || check == 'Y') {

         max = max +1;            // increment in the previous size

          int* temp = new int[max]; // create new bigger array.

          for (int i=0; i<n; i++) {

              temp[i] = a[i];       // copy values to new array.

          }


          delete [] a;              // free old array memory.

          a = temp;                 // now a points to new array.
          cout<<"Enter another value. ";
          cin>>a[n];
          n++;
            }
           else break;
 } while (check=='y'|| check =='Y');

 //--- Write out the array etc.
 cout<<endl<<”Array Values:  “;
 for(int i=0;i<n;i++)
 cout<<a[i]<<" ";


}
```

Output:

```
Enter 5 values:
1
2
3
4
5

Wanna enter more values??(y/n)  y
Enter another value. 6

Wanna enter more values??(y/n)  y
Enter another value. 7
Wanna enter more values??(y/n)  n
Array Values:  1 2 3 4 5 6 7
```

TASKS

1. Write a program in which you should input a string without space from the user, and the program should perform the following operations using pointers:
   - Check if the string is palindrome.
   - Count the frequency of a certain character.
2. Dynamic 2D Array Manipulation
   Write a C++ program that performs the following steps:

1. **Dynamic 2D Array Creation**:
   - Ask the user to input the number of rows and columns for a 2D integer array.
   - Dynamically allocate memory for the 2D array using new.
2. **Array Input**:
   - Prompt the user to enter values for the 2D array.
3. **Array Manipulation**:
   - Find and display the **sum** of all elements in the 2D array.
   - Find and display the **maximum** and **minimum** values in the 2D array.
   - Transpose the 2D array and display the transposed array.
4. **Dynamic Memory Deallocation**:
   - Deallocate the memory allocated for the 2D array using delete.
5. **Output**:
   - Display the results (sum, maximum, minimum, and transposed array) to the user.

Sample Input/Output:

Input:

```
Enter the number of rows: 2
Enter the number of columns: 3
Enter the elements of the array:
1 2 3
4 5 6
```

Output:

```
Sum of array elements: 21
Maximum value in the array: 6
Minimum value in the array: 1
Transposed array:
1 4
2 5
3 6
```

Steps to Implement:

1. Use new to allocate memory for the 2D array based on the user-provided rows and columns.
2. Use nested loops to take input for the 2D array elements.
3. Use nested loops to calculate the sum, maximum, and minimum values.
4. Transpose the 2D array and display it.
5. Use delete to free the dynamically allocated memory.
6. Display the results.