EC-200 Data Structures

LAB MANUAL # 03

**Course Instructor:** Lecturer Anum Abdul Salam

**Lab Engineer:** Lab Engineer Ayesha Batool

**Student Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Degree/ Syndicate: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Trait** | **Obtained Marks** | **Maximum Marks** |
| **R1** | **Application Functionality 20%** |  | 20 |
| **R2** | **Specification & Data structure implementation**  **30%** |  | 30 |
| **R3** | **Reusability**  **10%** |  | 10 |
| **R4** | **Input Validation**  **10%** |  | 10 |
| **R5** | **Efficiency**  **20%** |  | 20 |
| **R6** | **Delivery**  **10%** |  | 10 |
| **R7** | **Plagiarism above 80%** |  | 1 |
|  | **Total** |  | 10 |

**Total Marks = O**𝒃𝒕𝒂𝒊𝒏𝒆𝒅 𝑴𝒂𝒓𝒌𝒔 (∑6𝟏 𝑹𝒊 ∗ 𝑹7)

# Lab #03

# Pointers, dynamic array and templates

# Lab Objective:

To practice templates & copy constructors

# Dynamic memory

Till now, we have worked on program where all memory needs were determined before program execution by defining the variables needed. But there may be cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory, for which the C++ language integrates the operators new and delete.

## Operators new and new[]

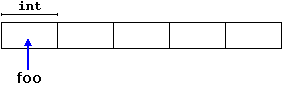
Dynamic memory is allocated using operator new. new is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated. Its syntax is:

pointer = new type  
pointer = new type [number\_of\_elements]

The first expression is used to allocate memory to contain one single element of type type. The second one is used to allocate a block (an array) of elements of type type, where number\_of\_elements is an integer value representing the amount of these. For example:

|  |  |
| --- | --- |
| int \* foo;  foo = new int [5]; |  |

In this case, the system dynamically allocates space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to foo (a pointer). Therefore, foo now points to a valid block of memory with space for five elements of type int.



Here, foo is a pointer, and thus, the first element pointed to by foo can be accessed either with the expression foo[0] or the expression \*foo (both are equivalent). The second element can be accessed either with foo[1] or \*(foo+1), and so on...

There is a substantial difference between declaring a normal array and allocating dynamic memory for a block of memory using new. The most important difference is that the size of a regular array needs to be a *constant expression*, and thus its size has to be determined at the moment of designing the program, before it is run, whereas the dynamic memory allocation performed by new allows to assign memory during runtime using any variable value as size.

## Operators delete and delete[]

In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of operator delete, whose syntax is:

|  |  |
| --- | --- |
| delete pointer;  **delete[] pointer**; |  |

The first statement releases the memory of a single element allocated using new, and the second one releases the memory allocated for arrays of elements using new and a size in brackets ([]).  
The value passed as argument to delete shall be either a pointer to a memory block previously allocated with new, or a *null pointer* (in the case of a *null pointer*, delete produces no effect).

|  |  |  |
| --- | --- | --- |
| // rememb-o-matic  #include <iostream>  #include <new>  using namespace std;  int main ()  {  int i,n;  int \* p;  cout << "How many numbers would you like to type? ";  cin >> i;  p= new int[i];  if (p == nullptr)  cout << "Error: memory could not be allocated";  else  {  for (n=0; n<i; n++)  {  cout << "Enter number: ";  cin >> p[n];  }  cout << "You have entered: ";  for (n=0; n<i; n++)  cout << p[n] << ", ";  delete[] p;  }  return 0;  } | How many numbers would you like to type? 5  Enter number : 75  Enter number : 436  Enter number : 1067  Enter number : 8  Enter number : 32  You have entered: 75, 436, 1067, 8, 32, |  |

Notice how the value within brackets in the new statement is a variable value entered by the user (i), not a constant expression:

|  |  |  |
| --- | --- | --- |
|  | p= new int[i]; |  |

There always exists the possibility that the user introduces a value for i so big that the system cannot allocate enough memory for it. For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program, and I got the text message we prepared for this case (Error: memory could not be allocated).  
It is considered good practice for programs to always be able to handle failures to allocate memory, either by checking the pointer value (if nothrow) or by catching the proper exception.

# Templates:

Generic Programming (one template->multiple data types)

There are two types of templates

* Function templates
* Class templates

## Function Templates

Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using template parameters. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.  
The format for declaring function templates with type parameters is:

*template <class identifier> function\_declaration;  
template <typename identifier> function\_declaration;*

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

|  |  |  |
| --- | --- | --- |
|  | template <class myType>  myType GetMax (myType a, myType b) {  return (a>b?a:b);  } |  |

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this still-undefined type.

To use this function template we use the following format for the function call:  
function\_name <type> (parameters);

For example, to call GetMax to compare two integer values of type int we can write:

|  |  |  |
| --- | --- | --- |
|  | int x,y;  GetMax <int> (x,y); |  |

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

## Function template Example:

#include "stdafx.h"

#include <iostream>

using namespace std;

template <class T>

T min(T a, T b)

{

if (a > b)

return a;

else

return b;

}

template <class A ,class B>

B fun(A a, B b){

return b;

}

void main(){

cout << "welcome" << endl;

cout << min(3, 5) << endl;

cout << min(4.0, 5.0) << endl;

cout << min('A', 'B') << endl;

cout << fun(3, 3.5) << endl;

int a;

cin >> a;

}

## Class Templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

|  |  |  |
| --- | --- | --- |
|  | template <class T>  class mypair {  T values [2];  public:  mypair (T first, T second)  {  values[0]=first; values[1]=second;  }  }; |  |

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

|  |  |  |
| --- | --- | --- |
|  | mypair<int> myobject (115, 36); |  |

this same class would also be used to create an object to store any other type:

|  |  |  |
| --- | --- | --- |
|  | mypair<double> myfloats (3.0, 2.18); |  |

The only member function in the previous class template has been defined inline within the class declaration itself. In case that we define a function member outside the declaration of the class template, we must always precede that definition with the template <...> prefix:

|  |
| --- |
| // class templates  #include <iostream>  using namespace std;  template <class T>  class mypair {  T a, b;  public:  mypair (T first, T second)  {a=first; b=second;}  T getmax ();  };  template <class T>  T mypair<T>::getmax ()  {  T retval;  retval = a>b? a : b;  return retval;  }  int main () {  mypair <int> myobject (100, 75);  cout << myobject.getmax();  return 0;  } |

Notice the syntax of the definition of member function getmax:

|  |  |  |
| --- | --- | --- |
|  | template <class T>  T mypair<T>::getmax () |  |

Confused by so many T's? There are three T's in this declaration: The first one is the template parameter. The second Trefers to the type returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

# LAB TASK

## TASK 01:

1. Create a class to swap two characters. your class should contain data member that points to char type variable.

Provide: `

* **default constructor( )**
* **void setValue(charmem1, charmem2)** Points the pointer to some value
* **char getValue()** return the values repective pointer is pointing to
* **char swapValue()** to swap the value of both the characters (i-e one passed as parameter)
* **Destructor()**
* **void Display()** to display new and old values

Your class should allocate memory dynamically and there should be no memory leak and dangling pointer.

1. Write a program to convert kilogram to grams by passing pointers as arguments to the function.

Provide:

* **default constructor( )**
* **void setValue(intval)** Points the pointer to value
* **int getValue()** return the value pointer is pointing to
* **float convert()** to covert from kg to g (i-e input value passed as parameter)
* **Destructor ()**
* **void Display()** to display the resultant value

Your class should allocate memory dynamically and there should be no memory leak and dangling pointer.

## TASK 02:

* Write a template function that receives an array with generic data type and returns average of all elements of the array. Pass array and size of array as arguments to the function. (Exercise with int, double and float data types.)
* Write template function that receives an array with generic data type. Function should reverse the elements in array.

## Test Plan:

|  |  |
| --- | --- |
| template<class Avg>  Avg avgNumber(Avg num1[])  {  //logic} |  |
| double arr[5];  cout << "Enter five numbers:\n"; |  |
| cin >> arr[i];  //get array elements form the user |  |
| Cout<<avgNumber(arr,)<<" is the average" << endl; |  |
| template<class Rever>  Rever revArray(Rever num1[],int size)  {//logic for reversing the elements of array{  }  cout << "The new array order is " << endl;  //print array elements in new order |  |
| Void main()  {float arr[5];  cout << "Enter five numbers:\n";} |  |
| cin >> arr[i];  //get array elements form the user |  |

## TASK 03:

**Create a class template where each object of template class can hold a dynamic array of any size and any type given by user and passed in overloaded constructor**.

**For Example**

<int> Template\_class(5); // will create an array of size 5 of type int

<flaot> Template\_class(10);// will create an array of size 10 of type float

<double> Template\_class(20);// will create an array of size 10 of type double

Provide

* 1. Default, copy and overloaded constructors
  2. SetValues() function that sets value to array by taking input from user
  3. ShowArray() to display the array
  4. ResizeArray(int nSize) to resize array
  5. Destructor

**Note: Your program should have no memory leakage & dangling pointers. Your program should be validated properly and display warnings and errors to user as well.**

## Test Plan:

Perform following operations to your program and write the results if visible on console and status (Operation successful, unsuccessful) in case if no result is visible on console.

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.** | **Operations** | **Values** | **Results/Status** |
| **1.** | An array of default size **def**  Templateclass <int> Template\_class1(); |  |  |
| **2.** | An array of size from user **iuserDef**  Templateclass <int> Template\_class2(6); | size=6  [0,0,0,0,0,0] |  |
|  | An array of size from user of type float **fuserDef**  Templateclass <float> Template\_class3(4); | size = 4  [0.0,0.0,0.0,0.0] |  |
| **3.** | An array, copy of userDef array **(cUserDef)** | size=6  [0,0,0,0,0,0] |  |
|  | Set values to user define array **fuserDef**.  Template\_class3.setValue(); | [2.2,4.5,7.4,8.5] |  |
| **4.** | Set values to **def** array  Template\_class1.setValues(); | [2,4,6,7,8,9] |  |
| **5.** | Assign values to **cUserDef** array  Template\_class2.setValues(); | [7,5,6,9,0,1] |  |
| **6.** | Display **cUserDef** array of type int |  |  |
| **7.** | Call again set values function with **iuserDef** array. \* Template.class2.setValue(); | [5,7,8,9,0,1] |  |
| **8.** | Display **fuserDef** Array of type float |  |  |
| **9.** | Display **iuserDef** array of type int |  |  |
| **10.** | Resize **cUserDef** array \* | size=4  Warning: Loss of data?  [7,5,6,9] |  |
| **11.** | Display **cUserDef** array |  |  |