|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | **Programming Fundamentals** | |  | **(CL214)** | |  | **LABORATORY MANUAL** | |  | **Spring 2021** | |  | **C:\Users\Aamer\Desktop\nu-new.png**  **LAB 14** | |  | **Linked List, Template Functions and Classes** | |  | **Engr. Ibrar Khan**  **Engr. Sana Saleh** |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | STUDENT NAME | | ROLL NO | | | SEC | |  | | | | | | | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | | | | | | LAB ENGINEER SIGNATURE & DATE | | | | | | | **MARKS AWARDED: /10** | | | | | | |  | | | | | | | **NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES (NUCES), ISLAMABAD** | | | | | | |  | | | | | | | Last Edited by: | Engr. Sana Saleh | | Version: | 4.01 | | | Prepared by: | Engr. Sana Saleh | | Date: | 6 Dec, 2018 | | | Verified by: | Engr. Shahid Qureshi | | Date: | 29 April,2019 | | |

|  |  |
| --- | --- |
| **LAB14** | **Linked List & Template Functions and Classes** |

**Lab Objectives:**

1. To learn how to use Linked list, template functions and classes.
2. To learn how to delete a node in a linked list.

**Software Required:**

* Dev C++

**Introduction:**

1. **Linked List:**

A linked list is a data structure that can store an indefinite number of items. These items are connected using pointers in a sequential manner.

The elements of a linked list are called the **nodes**. A node has two fields i.e. **data** and **next**. The data field contains the data being stored in that specific node. It cannot just be a single variable. There may be many variables presenting the **data** section of a node. The **next** field contains the address of the next node. So, this is the place where the link between nodes is established.

Creating a structure for node:

|  |
| --- |
| struct nodeType  {  float info;  nodeType \*link;  }; |

No matter how many nodes are present in the linked list, the very first node is called **head**.

* 1. **Creating new list:**

Now, we need a class which will contain the functions to handle the nodes. This class should have head pointers. The constructer will make them **NULL** to avoid any garbage value.

|  |
| --- |
| class FloatList  {  private:  nodeType \*head; // List head pointer  public:  FloatList(void) // Constructor  {  head = NULL; } |

* 1. **Creating New node:**

The process of creating node is very simple. We need a pointer of a node type (which we defined) and we will insert the value in its data field. The next field of node would be declared as NULL as it would be the last node of linked list.

Now, the function will have a very special case that we want to know what would happen if the linked list is still empty? We will have to check it. Do you remember that the head points to the first node? It means if the head is equal to NULL then we can conclude that the linked list is empty.

If there is just one node (which we are going to create) in linked lists, then it is called head.

The creation of a new node at the end of linked list requires, linking of the newly created node with last node. Means passing the address of a new node to the link pointer of a last node.

* 1. **Insertion:**

Inserting a new node in the linked list is called insertion. A new node is created and inserted in the linked list. There are three cases considered while inserting a node:

* Insertion at the start.
* Insertion at the end.
* Insertion at a particular position.

C++ code for insertion at start and end is given above.

* 1. **Deletion:**

Linked lists provide us the great feature of deleting a node. The process of deletion is also easy to implement. The basic structure is to declare a temporary pointer which points the node to be deleted. Then a little bit of working on links of nodes. There are also three cases in which a node can be deleted:

* + 1. Deletion at the start
    2. Deletion at the end
    3. Deletion at a particular position

**Deletion of node:**

The C++ code for deleting a node having particular value, entered by user, is given below:

|  |
| --- |
| void deleteNode(float num)  {  nodeType \*nodePtr, \*previousNode;  // If the list is empty, do nothing.  if (head == NULL)  {  return;  }  // Determine if the first node is the one to be deleted  if (head->info == num)  {  nodePtr = head->link;  delete head;  head = nodePtr;  }  else  { // Initialize nodePtr to head of list  nodePtr = head;  // Skip all nodes whose info member is not equal to num.  while(nodePtr!= NULL && nodePtr->info != num)  {  previousNode = nodePtr;  nodePtr = nodePtr->link;  }  //if end of linked list has been reached, and info has not been found  if(nodePtr == NULL)  {  cout<<num<<" not found\n\n";  }  // Else if the info has been found  else  { // Link the previous node to the node after  // nodePtr, then delete nodePtr.  previousNode->link = nodePtr->link;  delete nodePtr;  }  }  } |

* 1. **Function Template:**

A function template starts with the keyword template followed by template parameter(s) inside <> which is followed by the function definition.

template <typename T>

T functionName(T parameter1, T parameter2, ...) {

// code

}

In the above code, T is a template argument that accepts different data types (int, float, etc.), and typename is a keyword. When an argument of a data type is passed to functionName(), the compiler generates a new version of functionName() for the given data type.

### Calling a Function Template

Once we've declared and defined a function template, we can call it in other functions or templates (such as the main() function) with the following syntax

functionName<dataType>(parameter1, parameter2,...);

For example, let us consider a template that adds two numbers:

template <typename T>

T add(T num1, T num2) {

return (num1 + num2);

}

We can then call it in the main() function to add int and double numbers.

int main() {

int result1;

double result2;

// calling with int parameters

result1 = add<int>(2, 3);

cout << result1 << endl;

// calling with double parameters

result2 = add<double>(2.2, 3.3);

cout << result2 << endl;

return 0;

}

1. **Class Template:**

Templates are a way of making your classes more abstract by letting you define the behavior of the class without actually knowing what datatype will be handled by the operations of the class. In essence, this is what is known as generic programming; this term is a useful way to think about templates because it helps remind the programmer that a templated class does not depend on the datatype (or types) it deals with. To a large degree, a templated class is more focused on the algorithmic thought rather than the specific nuances of a single datatype. Templates can be used in conjunction with abstract datatypes in order to allow them to handle any type of data. For example, you could make a templated stack class that can handle a stack of any datatype, rather than having to create a stack class for every different datatype for which you want the stack to function. The ability to have a single class that can handle several different datatypes means the code is easier to maintain, and it makes classes more reusable.

The basic syntax for declaring a templated class is as follows:

|  |  |
| --- | --- |
|  | template <class a\_type> class a\_class {...}; |

The keyword 'class' above simply means that the identifier a\_type will stand for a datatype. NB: a\_type is not a keyword; it is an identifier that during the execution of the program will represent a single datatype. For example, you could, when defining variables in the class, use the following line:

|  |  |
| --- | --- |
|  | a\_type a\_var; |

and when the programmer defines which datatype 'a\_type' is to be when the program instantiates a particular instance of a\_class, a\_var will be of that type.

When defining a function as a member of a templated class, it is necessary to define it as a templated function:

|  |  |
| --- | --- |
|  | template<class a\_type> void a\_class<a\_type>::a\_function(){...} |

When declaring an instance of a templated class, the syntax is as follows:

|  |  |
| --- | --- |
|  | a\_class<int> an\_example\_class; |

An instantiated object of a templated class is called a specialization; the term specialization is useful to remember because it reminds us that the original class is a generic class, whereas a specific instantiation of a class is specialized for a single datatype (although it is possible to template multiple types).

Usually when writing code it is easiest to precede from concrete to abstract; therefore, it is easier to write a class for a specific datatype and then proceed to a templated - generic - class. For that brevity is the soul of wit, this example will be brief and therefore of little practical application.

|  |  |
| --- | --- |
|  | class calc  {    public:      int multiply(int x, int y);   };  int calc::multiply(int x, int y)  {    return x\*y;  } |

We will define the first class to act only on integers.

We now have a perfectly harmless little class that functions perfectly well for integers; but what if we decided we wanted a generic class that would work equally well for floating point numbers? We would use a template.

|  |  |
| --- | --- |
|  | template <class A\_Type> class calc or // template<class A\_Type> // class calc  {    public:      A\_Type multiply(A\_Type x, A\_Type y);      A\_Type add(A\_Type x, A\_Type y);  };  template <class A\_Type> A\_Type calc<A\_Type>::multiply(A\_Type x,A\_Type y)  {    return x\*y;  } |

To understand the templated class, just think about replacing the identifier A\_Type everywhere it appears, except as part of the template or class definition, with the keyword int. It would be the same as the above class; now when you instantiate an  
object of class calc you can choose which datatype the class will handle.

|  |  |
| --- | --- |
|  | calc <double> a\_calc\_class; |

Templates are handy for making your programs more generic and allowing your code to be reused later.

**Practice Problems:**

1. Write generic functions (i.e., template functions) that can perform following operations on arrays of different data type i.e., char, int, float and double. Keep the size of array 5:

template<class T>

|  |  |
| --- | --- |
| ***Function*** | ***Prototype*** |
| **Sort**: Sort the data in ascending order. | void sort(T array1[], int size) |
| **Mode:** Finding the number that appeared most in the data. | T mode(T array1[]) |
| **Smallest Number**: Find the smallest number in the data. | T smallest(T array1[]) |
| **Display:** Print the array. | void Display(T array1[]) |

#include<iostream>

using namespace std;

template<typename T>

T mode(T array[]){

int z=0;

for(int i=0;i<5;i++){

for(int j=i+1;j<5;j++){

if(array[i]==array[j]){

z++;

}

}

cout<<"elements repititon order is"<<z<<"\n";

z=0;

}

}

template <typename T>

void Display(T array1[]){

cout<<"array is\n";

for (int i = 0; i < 5; i++) {

cout << array1[i]<<"\t";

}

}

template<typename T>

T smallest(T array1[],int size){

int temp=array1[0];

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

for(int j=i+1;j<size;j++)

{

if(array1[j]<array1[i])

temp=array1[i];

array1[i]=array1[j];

array1[j]=temp;

}

}

cout<<"\n\nMinimum Member is\t"<<temp<<"\n\n";

}

template<typename T>

void sort(T arr1[]) {

int temp;

for(int i=0;i<5;i++)

{

for(int j=i+1;j<5;j++)

{

if(arr1[i]>arr1[j])

{

temp =arr1[i];

arr1[i]=arr1[j];

arr1[j]=temp;

}

}

}

cout<<"\n\nsorted array is\t";

for (int i = 0; i < 5; i++) {

cout << arr1[i]<<"\t";

}

}

int main() {

int arr[5] = {10,45,10,67,9};

Display<int>(arr);

sort<int>(arr);

smallest<int>(arr,5);

mode<int>(arr);

}

1. Implement all the above task of part A, by creating a template class:

class template<T>

class calculator

{

private:

T no1[5];

public:

void input();

void sort();

T smallest();

T mode();

void display();

**#include<iostream>**

**using namespace std;**

**template <class T>**

**class Calculator**

**{**

**private:**

**T no1[5];**

**public:**

**void input(T no1[5]){**

**}**

**void sort(T no1[]) {**

**int temp;**

**for(int i=0;i<5;i++)**

**{**

**for(int j=i+1;j<5;j++)**

**{**

**if(no1[i]>no1[j])**

**{**

**temp =no1[i];**

**no1[i]=no1[j];**

**no1[j]=temp;**

**}**

**}**

**}**

**cout<<"\n\nsorted array is\t";**

**for (int i = 0; i < 5; i++) {**

**cout << no1[i]<<"\t";**

**}**

**}**

**T smallest(T no1[5]){**

**int temp=no1[0];**

**for(int i=0;i<5;i++){**

**for(int j=i+1;j<5;j++)**

**{**

**if(no1[j]<no1[i])**

**temp=no1[i];**

**no1[i]=no1[j];**

**no1[j]=temp;**

**}**

**}**

**cout<<"\n\nMinimum Member is\t"<<temp<<"\n\n";**

**};**

**T mode();**

**void display();**

**};**

**//void calculator::input(){**

**int main(){**

**Calculator <int>c1;**

**int arr[5]={1,10,5,20,4};**

**c1.input(arr);**

**c1.sort(arr);**

**c1.smallest(arr);**

**}**

1. Given a singly linked list containing N nodes, add following functions in linked list:

* Destructor for the linked list class.
* Overload assignment operator for the linked list.
* Display the linked list.

To implement this program, create a menu and ask user the operation that he/she wants to perform.

\*Create a class of “Linked\_List” for the implementation of this task.

**EXTRA TASK:**

1. Write a program, that can add, subtract, divide and multiply two numbers using class template.

|  |
| --- |
| class template<T>  class calculator  {  private:  T no1, no2;  public:  T multiply(T a,T b);  …..  } |