SE – COMP-A

Experiment No.5

1. **Title:** Demonstrate function template & Class template for selection sorting algorithm.

# Problem Statement:

Write a function template selection Sort. Write a program that inputs, sorts and outputs an integer array and a float array.

# Objectives:

* 1. To learn and understand templates.
  2. To demonstrate function template for selection sort.

# Outcomes:

* 1. Students will be able to learn and understand working and use of function template.
  2. Students will be able to demonstrate function template for selection sort.

# Software/Hardware/Other Requirements:

* Any CPU with Pentium Processor / similar, 256 MB RAM/ more, 1GB HDD / more.
* Operating System – ubuntu/Fedora 64bit OS
* Software: G++ compiler/ GCC compiler, Code Editor

# Theory:

1. **Template**

* Template allows a function or class to work on many different data types without being rewritten for each one Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.
* A template is a blueprint or formula for creating a generic class or a function.
* The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.
* There is a single definition of each container, such as vector, but we can define many different kinds of vectors for example, vector <int> or vector <string>.
* You can use templates to define functions as well as classes, let us see how do they work:
* Function Template: The general form of a template function definition is shown here:

template <typename type> ret-type function-name (parameter list)

{

// body of function

}

* Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.
* A function template behaves like a function except that the template can have arguments of many different types (see example). In other words, a function template represents a family of functions. The format for declaring function templates with type parameters is:

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

* Both expressions have the same meaning and behave in exactly the same way. The latter form was introduced to avoid confusion, since a type parameter need not be a class. (it can also be a basic type such as int or double.)
* For example, the C++ Standard Library contains the function template max (x, y) which returns the larger of x and y. That function template could be defined like this:

template <typename T> inline T max (T a, T b) {

return a > b? a: b;

}

* This single function definition works with many data types. The usage of a function template saves space in the source code file in addition to limiting changes to one function description and making the code easier to read.

# Selection Sort

* Selection sort is a simple sorting algorithm. This sorting algorithm is a in-place comparison-based algorithm in which the list is divided into two parts, sorted part at left end and unsorted part at right end. Initially sorted part is empty and unsorted part is entire list.
* Smallest element is selected from the unsorted array and swapped with the leftmost element and that element becomes part of sorted array. This process continues moving unsorted array boundary by one element to the right.
* This algorithm is not suitable for large data sets as its average and worst-case complexity are of O(n2) where n is no. of items.

# Flowchart Diagram:

1. **Algorithm:**
2. Define a function template with name input () and two arguments, one argument for array to sort and other for size of array
3. Accept size number of elements from user for sorting in input ()
4. Define other function template display () with two arguments, one pointer of array to sort and other number of elements to sort
5. Print sorted elements stored in array
6. Define another function template sort () to implement selection sort for generic datatype.
7. Declare a variable min to hold index position of smallest element in array to sort.
8. Define for loop with loop variable “i” from 0 to size, incremented by one.
9. Assign first index position to min variable i.e. 0.
10. Define second for loop with variable “j” starting from “i + 1” to size, incremented by one.
11. In inner for loop, check if element at index position “j” is smaller than element at index position “min”.
12. If condition is true, assign min as j else continue
13. In outer for loop, swap element at index position “i” with element at index position at “min”.
14. In main () method, declare variable size to accept size of array.
15. Input number of elements to sort and declare an array of given size.
16. Use int as datatype of array
17. Call input () function and pass array name and size as argument
18. Call sort () function and pass array name and size as argument to sort
19. Call display () function and pass array name and size as argument to display sorted array.
20. Repeat steps from 15 to 19 with datatype float
21. End program.

# Implementation:

#include<iostream> using namespace std; template<class T>

void input(T \*a, int size)

{

cout<<"\nEnter "<<size<<" elements in array"; for(int i=0;i<size;i++)

cin>>a[i];

}

template<typename T> void sorting(T \*a, int size)

{

int min;

for(int i=0;i<size-1;i++)

{

min=i;

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

{

if(a[j]<a[min])

min=j;

}

**C++ Program for implementation of Selection Sort using Function Template**

**Output:**

How many elements you want to sort?: 5 Enter 5 elements in array: 10 32 2 -32 11

Elements before sorting: 10 32 2 -32 11

Elements after sorting: -32 2 10 11 32

Enter 5 elements in array: 1.30 3.2 4.2 -4.32 1.1

Elements before sorting: 1.30 3.2 4.2 -4.32 1.1

Elements after sorting: -4.32 1.1 1.30 3.2 4.2

swap(a[i],a[min]);

}

}

template<typename T> void display(T \*a, int size)

{

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

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

}

int main()

{

cout<<"\nHow many elements you want to sort\n"; int size;

cin>>size; if(size <= 0)

{

cout<<"Array size cannot be negative."; return 1;

}

int a[size]; input(a,size);

cout<<"\nElements before sorting\n"; display(a,size);

cout<<"\nElements after sorting\n"; sorting(a,size);

display(a,size); float b[size]; input(b,size);

cout<<"\nElements before sorting\n"; display(b,size);

cout<<"\nElements after sorting\n"; sorting(b,size);

display(b,size); return 0;

}

1. **Conclusion:**

Hence, we demonstrated use of function template for selection sort.

# Review Questions & Exercises:

* 1. **Fill in the Blanks**
     1. When writing function or class templates, you use a(n) **Type Parameter** to specify a generic data type.
     2. The beginning of a template is marked by a(n) **Template Prefix** .
     3. When defining objects of class templates, the **Data Type** you wish to pass into the type parameter must be specified.
     4. A(n) **Specialized** template works with a specific data type.
     5. All type parameters defined in a **function template** must appear at least once in the function parameter list.

# Answer the following:

* + 1. When should we use templates?

Ans. **Templates are** powerful features of **C++** which allows **you** to write **generic** programs. In simple terms, **you can** create a single function or a class to work **with** different data types **using templates**. **Templates are** often **used** in larger codebase for the purpose of code reusability and flexibility of the programs.

* + 1. How does a template work?

Ans. 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.

* + 1. What are the advantages of templates over macros?

Ans. **Templates** are type-safe. **Templates** avoid some of the common errors found in code that makes heavy use of function-like **macros**. Perhaps most importantly, **templates** were designed to be applicable to much larger problems than **macros**. The definition of a function-like **macro** must fit **on** a single logical line of code.

# Algorithm Assignments:

* + 1. Write a function template that searches a any type of array for a specified value. The function should return the subscript of the element containing the value if it is found in the array. If the value is not found, the function should return negative value.

#include<bits/stdc++.h>

using namespace std;

inline void guts() {

#ifndef ONLINE\_JUDGE

freopen("input.txt", "r", stdin);

freopen("output.txt", "w", stdout);

freopen("debug.txt", "w", stderr);

#endif

}

template<typename T>

int linearSearch(T \*a, int n, T tar)

{

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

{

if (tar == a[i]) return i;

}

return -1;

}

signed main()

{

guts();

int a[5] = {2, 6, 4, 5, 7};

char b[5] = {'a', 'f', 'd', 'g', 'r'};

cout << linearSearch(a, 5, 5) << endl;

cout << linearSearch(b, 5, 'f') << endl;

return 0;

}

* + 1. Write a template for a function called total. The function should keep a running total of values entered by the user, then return the total. The argument sent into the function should be the number of values the function is to read.

#include <iostream>

#include <string>

**using** **namespace** std;

**template**<**class** **T**>

T total(int numValues){

T res = 0, value;

**while**(numValues > 0){

cin >> value;

res += value;

numValues--;

}

**return** res;

}

int main()

{

cout << "First try with 5 ints...**\n**";

cout << "Running total is: " << total<int>(5) << endl;

cout << "**\n**Now try with 4 chars...**\n**";

cout << "(Results are usually weird here...)**\n**";

cout << "Running total is: " << total<char>(4) << endl;

cout << "**\n**Finally try with 3 doubles...**\n**";

cout << "Running total is: " << total<double>(3) << endl;

**return** 0;

}

# Programming Assignments:

* + 1. Write templates for the two functions minimum and maximum. The minimum function should accept two arguments and return the value of the argument that is the lesser of the two. The maximum function should accept two arguments and return the value of the argument that is the greater of the two. Design a simple driver program that demonstrates the templates with various data types.

#include<bits/stdc++.h>

using namespace std;

inline void guts() {

#ifndef ONLINE\_JUDGE

freopen("input.txt", "r", stdin);

freopen("output.txt", "w", stdout);

freopen("debug.txt", "w", stderr);

#endif

}

template <typename T> inline T giveMax (T a, T b) {

return a > b ? a : b;

}

template <typename T> inline T giveMin (T a, T b) {

return b > a ? a : b;

}

signed main()

{

guts();

cout << giveMax(4, 7) << endl;

cout << giveMin(4, 7) << endl;

cout << giveMin('j', 'f') << endl;

return 0;

}

* + 1. Write the class template for Selection Sort algorithm. Which can accepts set of values of any type, displays the all values & sort function which sorts the all values in Descending order. Also Drive this class template on atleast 3 datatypes.

// ...

#include<bits/stdc++.h>

using namespace std;

inline void guts() {

#ifndef ONLINE\_JUDGE

freopen("input.txt", "r", stdin);

freopen("output.txt", "w", stdout);

freopen("debug.txt", "w", stderr);

#endif

}

template<class T>

void input(T \*a, int size)

{

cout << "\nEnter " << size << " elements in array";

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

cin >> a[i];

}

template<typename T>

void sorting(T \*a, int size)

{

int min;

for (int i = size - 1; i > 0; i--)

{

min = i;

for (int j = i - 1; j > -1; j--)

{

if (a[j] < a[min])

min = j;

}

swap(a[i], a[min]);

}

}

template<typename T>

void display(T \*a, int size)

{

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

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

}

int main()

{

guts();

cout << "\nHow many elements you want to sort\n";

int size;

cin >> size;

if (size <= 0)

{

cout << "Array size can not be negative.";

return 1;

}

int a[size];

input(a, size);

cout << "\nElements before sorting\n";

display(a, size);

cout << "\nElements after sorting\n";

sorting(a, size);

display(a, size);

float b[size];

input(b, size);

cout << "\nElements before sorting\n";

display(b, size);

cout << "\nElements after sorting\n";

sorting(b, size);

display(b, size);

char c[size];

input(c, size);

cout << "\nElements before sorting\n";

display(c, size);

cout << "\nElements after sorting\n";

sorting(c, size);

display(c, size);

return 0;

}

* + 1. Write a program that implements a template-based quick Sort.

#include<bits/stdc++.h>

using namespace std;

inline void guts() {

#ifndef ONLINE\_JUDGE

freopen("input.txt", "r", stdin);

freopen("output.txt", "w", stdout);

freopen("debug.txt", "w", stderr);

#endif

}

template<class T>

void swap(T\* a, T\* b)

{

T t = \*a;

\*a = \*b;

\*b = t;

}

template<class T>

int partition (T arr[], int low, int high)

{

T pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++)

{

if (arr[j] < pivot)

{

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

template<class T>

void quickSort(T arr[], int low, int high)

{

if (low < high)

{

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

template<class T>

void printArray(T arr[], int size)

{

int i;

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

cout << arr[i] << " ";

cout << endl;

}

int main()

{

guts();

int arr[] = {10, 7, 8, 9, 1, 5};

int n = sizeof(arr) / sizeof(arr[0]);

quickSort(arr, 0, n - 1);

cout << "Sorted array: \n";

printArray(arr, n);

char b[] = {'a', 'f', 'c', 'd', 'j', 'b'};

quickSort(b, 0, n - 1);

cout << "Sorted array: \n";

printArray(b, n);

return 0;

}

# References:

* 1. E Balagurusamy Object-Oriented Programming with C++.7th edition. McGraw-

Hill Publication, ISBN 10: 9352607996 ISBN 13: 9789352607990

* 1. K. R. Venugopal, B. Rajkumar, and T. RaviShankar Mastering C++ by K. R. Venugopal; with 2nd edition. McGraw-Hill Publication, ISBN13: 9781259029943.
  2. Yashavant P. Kanetkar Test Your C++ Skills, BPB Publications, ISBN: 9788176565547