Lecture review

Generic Programming The generic programming pattern generalizes the algorithm with the help of templates in C++ so that it can be used along with different data types. In templates, we specify a placeholder instead of the actual data type, and that placeholder gets replaced with the data type used during the compilation. So, if the template function were being called for integer, character, and float, the compiler would produce 3 copies of the function. This is possible because of the static-type nature of C++.
The method of Generic Programming is implemented to increase the efficiency of the code. Generic Programming enables the programmer to write a general algorithm which will work with all data types. It eliminates the need to create different algorithms if the data type is an integer, string or a character.
The advantages of Generic Programming are

1. Code Reusability
2. Avoid Function Overloading
3. Once written it can be used for multiple times and cases.

Generics can be implemented in C++ using Templates . T

Templates

The simple idea is to pass the data type as a parameter so that we don't need to write the same code for different data types. emplate is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

Generic functions using Template

Examples

```
#include<iostream>
using namespace std;
template <typename T>
T my_max(T a, T b){
    if(a>b)
    return a;
    else
    return b;
}
int main(){
    cout<<my_max(4,5)<<endl;
    cout<<my_max(4.5,2.5)<<endl;
    cout<<my_max('a', 'b')<<endl;
}
```

```cpp
#include<iostream>
using namespace std;
template <class T>
T my_sum(T a, T b){
    return a + b;
}
int main(){
    cout<<my_sum(4,5)<<endl;
    cout<<my_sum(4.2,5.3)<<endl;
    cout<<my_sum('a','b')<<endl;
    cout<<my_sum(string("IBA "), string("Karachi"))<<endl;
}
```

```cpp
#include<iostream>
using namespace std;
template <class T>
class sum{
    public:
        T operate(T a, T b){
            return a + b;
        }
};
int main(){
    sum <int> obj;
    cout<<obj.operate(4,5)<<endl;
    sum <float> obj2;
    cout<<obj2.operate(4.7,5.8)<<endl;
    sum <string> obj3;
    cout<<obj3.operate(string("IBA"),string(" Karachi"))<<endl;
}
```

```cpp
#include<iostream>
using namespace std;
template <class T, int size>
class array{
    private:
        T arr[size];
    public:
        T& print (int index){
            return arr[index];
        }
};

int main(){
    array <int, 3> obj;
    for (int i = 0; i<3; i++){
        obj.print(i) = i + 2;
    }
    cout << "array's elements:\t";
    for (int i = 0; i < 3; i++) {
        cout << obj.print(i) << " ";
    }
```

```
    cout << endl;
    return 0;
}
```

```
#include<iostream>
using namespace std;
template<class T, int size>
class array{
    private:
        T arr[size];
    public:
        T & operator [](int index){
            return arr[index];
        }
};
int main(){

    array <int, 3> obj;
    array <double, 5> obj2;

    for(int i=0; i<3; i++){
        obj[i]=i*2;
         cout << "intArray[" << i << "] = " << obj[i] << endl;
    }

    cout<<"double array values are :\n";

    for(int i=0; i<5; i++){
        obj2[i]=i+2;
         cout << "doubleArray[" << i << "] = " << obj2[i] << endl;
    }
}
```

Lab exercises

Exercise 1

Write a generic function max_value() that takes three arguments of any type and returns the maximum of the three values.

Exercise 2

Write a generic function swap_values() that takes two arguments of any type and swaps their values.

Exercise 3

Finding element frequency.

Write a generic function element_frequency() that takes an array of any type and its size, and a value to search for, and returns the number of times the value appears in the array.

Exercise 4

sorting any type of array.

Write a generic function sort_array() that takes an array of any type and its size, and sorts the array in ascending order. The same program must contain a function reverse_array that takes the same array, and reverses the order of elements in the array. There must also be a generic function binary_search() that takes your sorted array, and a value to search for, and returns the index of the value in the array (or -1 if not found) using binary search.

Exercise 5

Stack Implementation

Implement a generic stack class Stack that can store elements of any type, with push, pop, and top operations.

Exercise 6

Queue implementation.

Implement a generic Queue class that can store elements of any type, with enqueue, dequeue, and front operations.

Exercise 7

LinkList implementation.

Implement a generic LinkedList class that can store elements of any type, with operations for insertion, deletion, and traversal.

Exercise 8

Implement a generic SparseMatrix class that can represent a sparse matrix of any type, with operations for addition, subtraction, and multiplication.

Exercise 9

Write a generic function matrix_operation that takes two 2D arrays of any type, their dimensions, and a binary operation function as arguments. The function should apply the operation element-wise on the two matrices and return the resulting matrix.

function signature.

```
template <typename T, typename BinaryOperation>
std::vector<std::vector<T>> matrix_operation(const std::vector<std::vector<T>>& matrix1,
                                             const std::vector<std::vector<T>>& matrix2,
                                             size_t rows, size_t cols,
                                             BinaryOperation op);
```

To perform matrix addition using this function, you can pass it a lambda function that adds two elements:

```
auto add = [](const T& a, const T& b) { return a + b; };
auto result = matrix_operation(matrix1, matrix2, rows, cols, add);
```

Exercise 10 .....

PriorityQueue Class.

Implement a generic PriorityQueue class that can store elements of any type, with operations for insertion and retrieval of the highest priority element. function signature.

```
template <typename T, typename PriorityFunc = std::less<T>>
class PriorityQueue {
public:
    void insert(const T& element);
    T getHighestPriority();
    // Add any additional functions you need for your PriorityQueue implementation

private:
    std::priority_queue<T, std::vector<T>, PriorityFunc> pq;
};
```

```
    int main() {
    // Example usage with integers and a custom priority function
    PriorityQueue<int, MyPriorityFunction> pq;

    // Inserting elements
    pq.insert(10);
    pq.insert(30);
    pq.insert(20);

    // Retrieving and removing the highest priority element
    int highestPriorityElement = pq.getHighestPriority();
    std::cout << "Highest priority element: " << highestPriorityElement << std::endl;

    // Add more operations as needed

    return 0;
}
```

****