

Day 9 (10-12-2025)

Function template

- Function template is a blueprint for creating functions that can operate with any data type.
- It allows you to write a single function definition that can work with different data types without needing to overload the function for each type.
- The syntax for defining a function template involves using the `template` keyword followed by the type parameter enclosed in angle brackets (`<` and `>`).
- The type parameter can be named anything, but `T` is commonly used as a convention.
- When you call a function template, the compiler generates a specific version of the function for the data type you provide.

Syntax:

```
template <typename T>
T functionName(T parameter1, T parameter2) {
    // function body
}
```

Example:

```
#include <iostream>
using namespace std;

// function template
template <typename T>
T add(T a, T b){           //typename
    return a + b;
}

// if same type -> it can be used

//normal function
int add(int a, int b){
    return a + b;
}

//method overloading
double add(double a, double b){
    return a + b;
}

// 10.0 15.5

//string
```

```
//float
//char

int main() {
    int a = 10, b = 20;
    double c = 10.3, d = 30.2; // precision = in this case it will not work

    cout << add(a,b) << endl;
    cout << add(c,d) << endl;

    // cout << add(a,d) << endl; // not valid -> error: no matching function for
    // call to 'add(int&, double&)'
    cout << add<double>(a,d) << endl; //valid
                                     //10.0 30.2

    return 0;
}
```

Class template

- Class templates allow you to create a class that can work with any data type.
- Similar to function templates, class templates use the **template** keyword followed by the type parameter
- The type parameter can be used to define member variables and member functions within the class.
- When you create an object of a class template, you specify the data type for which the class should be instantiated.

Syntax:

```
template <typename T>
class ClassName {
    T memberVariable;
public:
    // class members
};
```

Example:

```
#include <iostream>
using namespace std;

class A{
    int val;
public:
    A(int val);
    void display();
};

A::A(int val): val(val){}
```

```

void A::display(){
    cout<< val <<endl;
}

template <typename T>
class Demo{
    T val;
public:
    Demo(T val);

    T getVal();
};

template <typename T>
Demo<T>::Demo(T val) : val(val){}    //A::A(int val): val(val){}

template <typename T>
T Demo<T>::getVal(){                //void A::display(){
    return val;                      // cout<< val <<endl;
}                                    //}

template <typename T>
class Number{    // int, float, double
    T val;
public:
    Number(T val): val(val){}

    T getVal(){
        return val;
    }
};

//class IntNumber{};
//class DoubleNumber{};
//class String{};
//class charNumber{};

int main() {
    Demo<double> d1(10.2);

    Number<int> n1(10);

    Number<double> n2(40.5);

    cout << d1.getVal() << endl;

    cout << n1.getVal() << endl;
    cout << n2.getVal() << endl;
    return 0;
}

```

Intro to STL (vector, list, map)

- STL -> Standard Template Library

- It is a powerful set of C++ template classes to provide general-purpose classes and functions with
- templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

Components of STL

1. Algorithms - predefined functions for operations like searching, sorting, counting, manipulating data
2. Containers - data structures to store data
3. Iterators - objects that point to an element in a container and provide a way to access and traverse the elements
4. Functors - function objects that can be used as arguments to algorithms

STL Algorithms

- searching, sorting, counting, manipulating data

```
sort(), reverse(), max_element(), min_element(), accumulate(), count(), find(), binary_search()
```

STL Containers

- data structures to store data

```
vector, list, deque, stack, queue, priority_queue, set, map, unordered_set, unordered_map
```

STL Iterators

- objects that point to an element in a container and provide a way to access and traverse the elements

```
begin(), end(), rbegin(), rend(), advance(), next(), prev()
```

Functors

- function objects that can be used as arguments to algorithms

```
greater(), less(), plus(), minus()
```

We have 3 major components in STL in introductory level with syllabus

1. vector
2. list
3. map

STL Components

vector

- Vector is a dynamic array that can grow and shrink in size as needed.
- It provides random access to elements, meaning you can access any element in constant time using its index.
- Vectors are implemented as contiguous memory blocks, which allows for efficient memory usage and cache performance.

Example:

```

int main() {
    //declare a vector -> create a object of vector
    vector<int> vec;           //<int> -> integer, <double> -> floating,
                               // <char> -> characters, <string> -> string
                               // use(user defined object) <object> - object

    vec.push_back(10); //size -> 1, capacity -> max elements that can be stored
    cout << "Size : " << vec.size() << " Capacity : " << vec.capacity() << endl;
    vec.push_back(20);
    cout << "Size : " << vec.size() << " Capacity : " << vec.capacity() << endl;
    vec.push_back(30);
    cout << "Size : " << vec.size() << " Capacity : " << vec.capacity() << endl;
    vec.push_back(40);
    cout << "Size : " << vec.size() << " Capacity : " << vec.capacity() << endl;
    vec.push_back(50);
    cout << "Size : " << vec.size() << " Capacity : " << vec.capacity() << endl;
    // vec -> 4 values      similar to ArrayList
    cout << endl;

    cout << "Size : " << vec.size() << endl;
    cout << vec[0] << endl;
    cout << vec[1] << endl;
    cout << vec[2] << endl; //allowed

    for(int val : vec){        //for each
        cout << val << " ";
    }

    cout << endl;

    for(int i=0;i<vec.size();i++){
        cout << vec[i] << " ";
    }

    cout << endl;
    cout << "Front: " << vec.front() << endl;
    cout << "Back: " << vec.back() << endl;

    auto it = vec.begin(); // first occurrence with iterator return type
    vec.insert(it + 2, 2); //2 i want to insert at 0
    // insert -> O(n)

    for(int val : vec){        //for each
        cout << val << " ";
    }

    vec.pop_back();           // last element pop

    cout << endl;
}

```

```

    for(int val : vec){          //for each
        cout << val << " ";
    }

    vec.clear();

    for(int val : vec){          //for each
        cout << "anything"<< endl;
        cout << val << " ";
    }

    return 0;
}

```

list

- List is a doubly linked list that allows for efficient insertion and deletion of elements at any position.
- Unlike vectors, lists do not provide random access to elements, meaning you cannot access elements using an index.
- Lists are implemented as a series of nodes, where each node contains a data element and pointers to the previous and next nodes.

Example:

```

int main(){

    // list -> int
    list<int> l1;          //created an object of list

    // list -> same as Doubly linked list

    l1.push_back(10);    // as a last element
    l1.push_front(20);   // as a first element
    l1.push_back(30);
    l1.push_back(30);

    for(int val: l1)
        cout << val << " ";
    cout << endl;

    //Insert in middle
    auto it = l1.begin();
    advance(it, 2);      // iterator paramter, distance from begin

    l1.insert(it, 100);   // 100 -> after 2 distance

    it = l1.begin();
    advance(it, 1);      // distance 1

    l1.insert(it, 90);    // distance 1
}

```

```

    for(int val: l1)
        cout << val << " ";
    cout << endl;

    //Erase
    it = l1.begin();
    advance(it, 2);

    l1.erase(it);    //10 deleted

    for(int val: l1)
        cout << val << " ";
    cout << endl;

    l1.remove(30);

    for(int val: l1)
        cout << val << " ";
    cout << endl;
    cout << l1.size() << endl;

    l1.sort();
    l1.reverse();

    // cout << l1 << endl;
    // cout << l1[1] << endl;    //not valid

    return 0;
}

```

map

- Map is an associative container that stores elements in key-value pairs.
- It allows for fast retrieval of values based on their associated keys.
- Maps are implemented as balanced binary search trees (typically Red-Black Trees), which ensures that operations like insertion, deletion, and lookup can be performed in logarithmic time.
- Maps automatically sort the keys in ascending order.
- Each key in a map is unique, meaning that you cannot have duplicate keys.
- Values associated with keys can be of any data type, including user-defined types.

Example:

```

#include <iostream>
#include <vector>
#include <list>
#include <map>
using namespace std;

int main(){

```

```

    map<int, string> m1;    //created object of map

// keys -> values
// -----
m1[10] = "shil";
m1[10] = "himanshu";      //overrides values[3] = "himashu";
m1[18] = "isha";
m1[11] = "aniket";
m1[1] = "prathamesh";
// sorted keys -> Red-Black Trees

// 1. to check if the key exists -> values[key]

m1.insert({33, "Rahul"}); //modern
m1.insert(make_pair(44, "Gaurav")); //old syntax

m1.erase(10);

m1.empty(); //checks if the map is empty
m1.size();

auto it = m1.find(18); //heap -> dynamic memory
if(it != m1.end()){
    cout << "found : " << it->second<< endl;
}else{
    cout << "not found" << endl;
}

for(auto &m: m1){
    cout << m.first << " : " << m.second << endl;
}

return 0;
}

```

OOP major pillars

Encapsulation

- binding data and functions together
- data hiding
- prevent data from outside world

Abstraction

- hiding the data
- only show essential things to the user
- reduce complexity

Polymorphism

- one object, many forms
- compile time polymorphism -> method overloading, operator overloading
- runtime polymorphism -> method overriding (inheritance + virtual keyword)

Inheritance

- reusability of related things, which share same generalised properties
- parent-child relationship

OOP minor pillars

Association

- Uses-a relationship
- Both classes have their own lifecycle
- Example: Teacher and Marker

Aggregation

- Weak Has-a relationship
- Both classes have their own lifecycle
- Example: Department and Professor

Composition

- Strong Has-a relationship
- Child class cannot exist without parent class
- Example: Car and Engine

Tomorrows topic in detail

Inheritance concept

- reusability of related things, which share same generalised properties
- parent-child relationship

Modes & types

Diamond problem

- resolve it virtual keyword is used -> class, runtime -> method