Class templates

A class template is defined

- To keep all the algorithms and generic logic used
- The actual type of data is specified as a created.
- For example we have created a class myArray
- All common functions related to array can be defined in the generic template class.
- But data type of actual objects of myArray

```
class myArray{
        int size;
        int *ptr;
};
```

Type parameters can be used as

1. Data members of class.
2. Arguments of class functions.
3. Local variable in class functions.
4. Return type of class functions.

Template class definition for myArray.

```
template < typename T>
class myArray{
    int size; // Array size always int
    T *ptr; // Pointer for dynamic 1-D Array
public:
    myArray() { size=0; ptr=nullptr; }
    myArray(int size);
};
    //No code should be written between template header and class definition
```

```
template < typename T>
class myArray{
    int size; // Array size always int
    T *ptr; // Type parameter as dataType
public:
    myArray() { size=0; ptr=nullptr; }
    myArray(int size);
    void setValue(T value, int index); // Type parameter as Argument
    T getValue(int index); // Type parameter as return type
    void printArray();
    ~myArray();
};
//All function of class now become template functions.
```

Functions Implementation

1. Add template header before every function of class to define it outside.
2. Add template type **<type>** with class name to resolve scope of member function.

```
template < typename T> // Constructor
myArray<T>::myArray(int size) {
  if (size > 0)
        ptr = new T[size];
  this->size = size;
}
template < typename T> // Destructor
myArray<T>::~myArray() {
        if (ptr != nullptr)
                delete []ptr;
}
```

```
// Print data of Array
template < typename T>
void myArray<T>::printArray() {
        if (ptr != nullptr) {
                for (int i = 0; i < size; i++)
                        cout << ptr[i] << " ";
                cout << endl;
        }
}
```

Functions Implementation

```
template < typename T> // Setter
void myArray<T>::setValue
(T value, int index) {
  if (ptr != nullptr) {
        if (index < size && index >=0)
              ptr[index] = value;
  }
}
```

```
template < typename T> // Getter
T myArray<T>::getValue(T value, int index) {
  if (ptr != nullptr) {
        if (index < size && index >=0)
              return ptr[index];
  }
  else
        return NULL;
}
```

Functions Implementation

- Class templates are called *parameterized types.*
- Provide name of data type as <datatype> when object is created.
- At compile time, when compiler finds an object creation of specific type,
  - It generates the complete copy of template class by replacing the type parameters with the provided datatypes of object.
  - This is called *implicit specialization* or *class template instance*.
- If class object is not created, then no copy of template class is created by compiler.
- The class definition and implementation should be in same file.
  - Compiler need access to all functions for replacing type parameter.
  - Runtime function linking is not possible with template classes.

Compiler will generate three copies of myArray class template for **int**, **char**, and **const char ***.

```
void main()
    myArray <int> arr(4); // object type int
    arr.setValue(1, 0);  arr.setValue(9, 1); arr.setValue(5, 2); arr.setValue(8, 3);
    arr.printArray();

    myArray <char> arr2(3); // object type char
    arr2.setValue('a', 0); arr2.setValue('b', 1); arr2.setValue('c', 2);
    arr2.printArray();

    myArray <const char *> arr3(3); // object type const char *
    arr3.setValue("abc", 0); arr3.setValue("xyz", 1); arr3.setValue("def", 2);
    arr3.printArray();
}
```

Default value of Parameters

- A template class can has, default arguments associated with a template type parameter.
- Here, the type **int** will be used if no other type is specified, when an object is created.

```
template < typename T = int>
class myArray{
    int size; // Array size always int
    T *ptr; // Type parameter as dataType
public:
    myArray() { size=0; ptr=nullptr; }
    myArray(int size);
};

void main(){
    myArray <> arr(5); // object type int by default
    myArray <char> arr2(5); // object type char
    myArray <float> arr3(4); // object type float
}
```

Non-Type Parameters

- A template class can have non-type parameters along type parameters
- Their scope is global in class accessible in all functions.
- Non-type parameters can be integers, pointers, or references only.
- Non-type parameters are considered as constants, since their values cannot be changed.

```
template < typename T, int size>
class myArray{
    T arr[size]; // Non-Type Parameter as size of array
    //can only used to create static arrays, not dynamic ones.
public:
    void printArray();
};
// Template header is now changed for all functions too
template < typename T, int size>
void myArray <T,int>:: printArray(){// class name is also changed according to
template header
    for (int i = 0; i < size; i++)
                    cout << arr[i] << " ";
    cout << endl;
}
```

```
void main(){
    myArray <int, 10> arr;
    // object type int with static array of size 10
    myArray <float, 15> arr3;
    // object type float array size 15
}
```

Non-Type Parameters

- Add definition of non-member friend functions in class definition.
- For each instance of class an instance of friend function is created.

```
template < typename T = int>
class myArray{
    int size; // Array size always int
    T *ptr; // Type parameter as dataType
public:
// Generic function for All classes
    friend ostream& operator<<( ostream& out, myArray<T> & obj){
            if (obj.ptr != nullptr) {
                    for (int i = 0; i < obj.size; i++)
                            out << obj.ptr[i] << " ";
                    out << endl;
            }
            return out;
    }
};
```

Static data members

- In Non-template class **`static`** data members are shared between all objects
- In template class **`static`** data members are not shared between all different class instances
- Class-template specialization (Implicit by compiler, or Explicit by Programmer)
  - Each specialized instance of class owns copy of **`static`** member functions and **`static`** data members,
    - That is shared among all objects, that belong to specialized instance of class

```
void main()
    myArray <int> a(4); // object type int
    myArray <int> b(5); // object type int
    // Both objects a and b share single static data member, as they belong to same class type.

    myArray <char> c(3); // object type char
    myArray <char> d(3); // object type char
    // Both objects c and d share single static data member, as they belong to same class type.

// All objects not share single static member due to difference in type of specialized classes
    }
```

Compose

- We can compose a template class object in another template class
  - With specific specialized datatype,
  - Or as general template object, type is decided, when whole class object is created.

```
//Composed in template class
template < typename U >
class Compose{
    U abc;

//General template type object, type is decided by type of Compose object
    myArray<U> l1;

// char specialized object
    myArray<char> l2;
};
void main()
    Compose <int> c;
    // Specialized object type int, l1 type is also int and l2 type is char.
}
```

Compose

- We can compose template class object in another Normal class
  - With specific specialized datatypes only.

```
//Composed in template class
class Compose2{

// float specialized object
    myArray<float> l1;

// char specialized object
    myArray<char> l2;
};
void main()
    Compose2 c;
    // Normal object with composed types, float for l1 and char for l2.
}
```

Inheritance

- We can inherit from a template class in another template class
  - With specific specialized datatype of base class
  - General template class, base class type is decided according to derived class object type.

```
//Inherited as general base class
template < typename U >
class derived_MyArray :public myArray<U>{ };

//Inherited as specialized char base class
template < typename U >
class derived_MyArray2 :public myArray<char>{ };

void main()
    derived_MyArray <int> d1; // Derive object type int with base type int
    derived_MyArray2 <int> d2; // Derive object type int, but base type is char
}
    //Inherited as specialized base class
    class derived_MyArray :public myArray<float>{ };

    void main()
        derived_MyArray d1; // Normal derived object with base object type float
    }
```

Compose

- We can inherit from a template class in another template class
  - With specific specialized datatype of base class
  - General template class, base class type is decided according to derived class object type.

```
//Inherited as general base class
template < typename U >
class derived_MyArray :public myArray<U>{ };

//Inherited as specialized char base class
template < typename U >
class derived_MyArray2 :public myArray<char>{ };

void main()
    derived_MyArray <int> d1; // Derive object type int with base type int
    derived_MyArray2 <int> d2; // Derive object type int, but base type is char
}
```