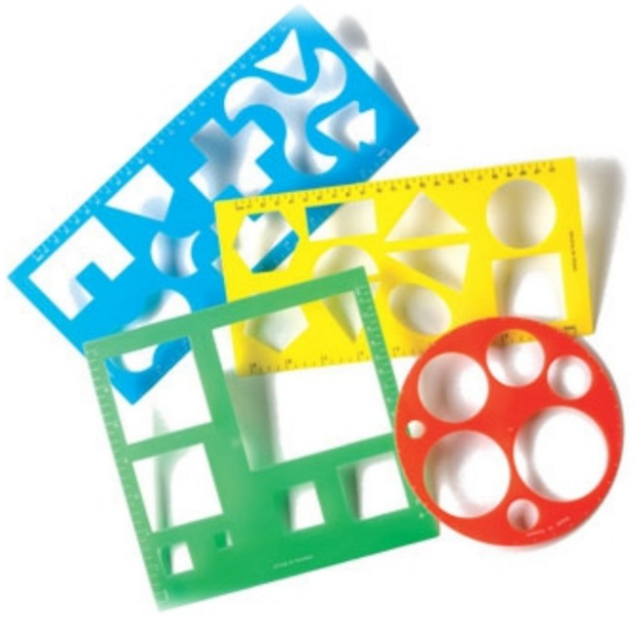
Class Templates

CS(217) Object Oriented Programming

Abeeda Akram

What is a Template?

- A **template** is a model or mold that can be used as a guide to create similar things.
- For example template is like a stencil ruler by using that we can draw same shape with different colors.
  - Once the stencil is created, it can be used many times for drawing shapes.



Templates in C++

- The template is one of C++'s most sophisticated and high-powered features that is used for generic programming.
  - It is a mechanism for automatic code generation, and allows for substantial improvements in programming efficiency.
- Using templates, we can create.
  1. Generic functions
  2. Generic classes
  - In a generic function or class, the type of data upon which the function or class operates is specified as a parameter.
  - We can use one function or class with several different types of data without explicitly recode specific versions for each data type.

Class Templates

- **Class templates** are special classes that serve as a framework or mold for creating other similar classes
  - without explicitly recoding specific versions for each data type.
- A class template is defined
  - To keep all the algorithms and generic logic used by that class at one place.
  - The actual type of data is specified as a parameter, when objects of that class are 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 can be different.

```
class myArray{
    int size; // Array size
    int *ptr; // Pointer for dynamic 1-D Array
};
```

Class Templates **Template header**

- First write keyword **`template`** followed by List of template type parameters in angle brackets (**<** and **>**)
- Each parameter is preceded by keyword **`class`** or **`typename`**

```
template < typename Type >
template < typename Type1, typename Type2>
```

- The labels **`Type, Type1, Type2`** are called a *template type parameters.*
- Type parameter is simply a placeholder or label
  - that is replaced by an actual datatype, when specific **object of that class** is created.
- 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 fucntions.

Class Templates **Definition**

1. Add template header before class definition.
2. Define class functions with generic code, use type parameter in place of actual datatype.

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
```

Class Templates **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 myArray now become template functions.
```

Class Templates **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.

```cpp
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;
}
```

```cpp
// 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;
    }
}
```

Class Templates **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> // 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(int index) {
  if (ptr != nullptr) {
      if (index < size && index >=0)
          return ptr[index];
  }
  else
      return NULL;
}
```

Class Templates **Objects**

- 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.

Class Templates **Objects**

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();
}
```

Class Templates **Objects as Type parameters**

Compiler will generate copy of myArray class template for **Point** objects.

```
void main()

    myArray <Point> arr(4); // object type Point

    arr.setValue(Point(2, 3), 0); // Call Parametrized constructor on Point
    arr.setValue(Point(),   1);  // Call Default constructor on Point
    arr.setValue(Point(5, 5), 2); // Call Parametrized constructor on Point
    arr.setValue(Point(9, 3), 3); // Call Parametrized constructor on Point

    arr.printArray();
    // Point class must implement cout operator as used in function
    printArray
 }
```

Class Templates **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
}
```

Class Templates **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;
}
```

Class Templates **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.

```
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
}
```

Class Templates **Non-Type Parameters with Default value**

```
template < typename T = int, int size = 5 >
class myArray{
   T arr[size]; // Non-Type Parameter as size of array
   //can only create static arrays, not dynamic ones.
public:
};

void main(){
   myArray <> arr; // object type int, size is 5 by default
   myArray <float, 15> arr3; // object type float array size 15
}
```

Class Templates **Non-member friend functions**

- 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;
   }
};
```

Class Templates **Static 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 function 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
   }
```

Class Templates **Composition**

- 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.
}
```

Class Templates **Composition**

- 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.
}
```

Class Templates **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
}
```

Class Templates **Inheritance**

- We can inherit from a template class in another Normal class
  - With specific specialized datatypes only.

```
//Inherited as specialized base class
class derived_MyArray :public myArray<float>{ };

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