Class Templates

CS(217) Object Oriented Programming

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
}
```

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);
    ~myArray();
    void setValue(T value, int index); // Type parameter as Argument
    T getValue(int index); // Type parameter as return type
    void printArray();

    bool operator==(const myArray &); //compare size and data of all elements
};
//Add new operator function for comparison of two arrays.
```

Class Templates **Implementation**

```
// Compare data of myArray objects
template < typename T>
bool myArray<T>:: operator==(const myArray & obj){
      if (size != obj.size) return false;
      if (ptr != nullptr && obj.ptr != nullptr ) {
            for (int i = 0; i < size; i++){
                  if (ptr[i] != obj.ptr[i])
                        return false;
            }
      }
      return true;
}
```

Class Templates **Objects**

```
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);

    myArray <int> arr2(arr); //Copy Constructor called on arr2

    cout << arr  << arr2;
    cout << (arr == arr2) <<endl; // Compare called

    arr.setValue(15, 3);
    cout << arr  << arr2;
    cout << (arr == arr2) <<endl; // Compare called
}
```

```
1 9 5 8
1 9 5 8
1
1 9 5 15
1 9 5 8
0
```

Class Templates **Objects**

```
void main(){

    myArray <char> arr(4); // object type int
    arr.setValue('a', 0);  arr.setValue('b', 1); arr.setValue('c', 2);
    arr.setValue('d', 3);

    myArray <char> arr2(arr); //Copy Constructor called on arr2

    cout << arr  << arr2;
    cout << (arr == arr2) <<endl; // Compare called

    arr.setValue('k', 3);
    cout << arr  << arr2;
    cout << (arr == arr2) <<endl; // Compare called
}
```

```
a b c d
a b c d
1
a b c k
a b c d
0
```

Class Templates **Objects**

```
void main(){
    myArray <const char *> arr(3); // object type const char *
    arr.setValue("abc", 0); arr.setValue("xyz", 1); arr.setValue("def", 2);

    myArray <const char *> arr2(3); // object type const char *
    arr2.setValue("abc", 0); arr2.setValue("xyz", 1); arr2.setValue("def", 2);

    cout << arr  << arr2;
    cout << (arr == arr2) <<endl; // Compare addresses instead of data

    arr.setValue ("ghk", 1);
    cout << arr  << arr2;
    cout << (arr == arr2) <<endl; // Compare addresses instead of data
 }
```

```
abc xyz def
abc xyz def
1
abc ghk def
abc xyz def
0
```

Class Templates **Objects**

```
void main(){
      char** ptr = new char* [3];
      for (int i = 0; i < 3; i++)
             ptr[i] = new char[4];
      strcpy(ptr[0], "abc"); strcpy(ptr[1], "def"); strcpy(ptr[2], "ghi");

      myArray <char *> arr(3), arr2(3); // object type char *
      arr.setValue(ptr[0], 0); arr.setValue(ptr[1], 1); arr.setValue(ptr[2], 2);
      arr2.setValue(ptr[0], 0); arr2.setValue(ptr[1], 1); arr2.setValue(ptr[2], 2);

      cout << arr  << arr2;
      cout << (arr == arr2) << endl; // Compare addresses instead of data
      strcpy(ptr[0], "aaa");
      cout << arr  << arr2;
      cout << (arr == arr2) << endl; // Compare addresses instead of data
      // Shallow copy of data in class objects for char *
}
```

```
abc def ghi
abc def ghi
1
aaa def ghi
aaa def ghi
1
```

Class Templates **Specialization**

- Template classes do not work well for all data types.
  - Explicit specialization of template classes is made, when some datatypes require different logic and implementation of class functions.
  - Class template specialization is to design an explicitly specialized class for a particular datatype along with existing template class.

1. Add empty template header before specialized template class

```
template <>
```

2. Add datatype name for specialization after class name <>

```
template <>
class classname < datatypename > {
        // class definition
};
```

Class Templates **Specialization**

1. Add specialized template class to handle shallow copy issue of **char ***
2. Specialized class can have different implementation of all functions.

```
template <> class myArray<char*>{
    int size; // Array size always int
    char* *ptr; // char * as specialized dataType
public:
    myArray() { size=0; ptr=nullptr; }
    myArray(int size);
    ~myArray();
    void setValue(char* value, int index); // Should perform deep copy of char*
    char* getValue(int index);
    bool operator==(const myArray &); //should compare strings instead of addresses

    //Add friend functions for class char* explicitly
    friend ostream& operator<<( ostream& out, myArray<char *> & obj){

        cout << "Special: ";
        if (obj.ptr != nullptr) {
                for (int i = 0; i < obj.size; i++)
                        out << obj.ptr[i] << " "; out << endl;
        }
        return out;
    }
};
```

Class Templates **Specialization**

- Do not add empty template header before any function of class, compile time error.
- Add Complete name of class with data type to resolve scope for member functions.

```
// Constructor
myArray<char *>::myArray(int size) {

  ptr = nullptr;
  this->size = size;
  if (size > 0){
        ptr = new char*[size];
        for (int i = 0; i < size; i++)
        ptr[i] = nullptr;
  }
// Empty Array of pointers
initialized with nullptr
}

// Destructor
myArray<char *>::~myArray() {
        if (ptr != nullptr)
                delete [] ptr;
}
```

```
// Getter
char * myArray<char *>::getValue(int index){
  if (ptr != nullptr) {
        if (index < size && index >=0)
                return ptr[index];
  }
  else
        return nullptr;
}

// Setter performing deep copy of data
void myArray<char *>:: setValue (char* value, int index){
  if (ptr != nullptr) {
        if (index < size && index >=0){
                if (ptr[index] != nullptr)
                        delete[] ptr[index];
                ptr[index] = new char[strlen(value)+1];
                strcpy(ptr[index], value);
        }
}
```

Class Templates **Specialization**

- Do not add empty template header before any function of class, compile time error.
- Add Complete name of class with data type to resolve scope for member functions.

```
// Compare deeply data of all char *
bool myArray<char *>::operator==(const myArray & obj){

        if (size != obj.size)
                return false;

        if (ptr != nullptr && obj.ptr != nullptr ) {
                for (int i = 0; i < size; i++){
                        if (strcmp(ptr[i], obj.ptr[i]) != 0)
                                return false;
                }
        }
        return true;
}
```

Class Templates **Specialization**

```
void main(){
      char** ptr = new char* [3];
      for (int i = 0; i < 3; i++)
             ptr[i] = new char[4];
      strcpy(ptr[0], "abc"); strcpy(ptr[1], "def"); strcpy(ptr[2], "ghi");

      myArray <char *> arr(3), arr2(3); // object type char *
      arr.setValue(ptr[0], 0); arr.setValue(ptr[1], 1); arr.setValue(ptr[2], 2);
      arr2.setValue(ptr[0], 0); arr2.setValue(ptr[1], 1); arr2.setValue(ptr[2], 2);

      cout << arr  << arr2;
      cout << (arr == arr2) << endl; // Compare data
      strcpy(ptr[0], "aaa");
      cout << arr  << arr2;
      arr2.setValue(ptr[0], 0);
      cout << arr << arr2;
      cout << (arr == arr2) << endl; // Compare data
}
```

```
Special: abc def ghi
Special: abc def ghi
1
Special: abc def ghi
Special: abc def ghi
Special: abc def ghi
Special: aaa def ghi
0
```