Template Class

Humayun Kabir
Professor, CS, Vancouver Island University, BC, Canada

Template Class

- If multiple classes have **logically similar** member variables and member functions and their only difference is in their data types, it is possible to write a single template class definition instead of multiple class definitions.
- Given the argument types provided during object instantiations, C++ automatically generates separate class template specializations to handle each type of object appropriately.

Template Class

- All template class definitions begin with the template keyword followed by a template parameter list enclosed in angle brackets (< and >).
- Every parameter in the template parameter list is preceded by keyword typename or keyword class.
- The type parameters are placeholders for fundamental types or user-defined types.
- Used to specify the types of the object’s data types.

Template Class

```
class DynamicArrayInt {
    private:
        int _capacity;
        int _size;
        int* _elements;
    public:
        DynamicArrayInt(int capacity);
        ~DynamicArrayInt();
        int get_capacity();
        int get_size();
        void add(int element);
        bool contains(int element);
        bool remove(int element);
        int* asArray();
};
```

Template Class

```
class DynamicArrayDuble {
    private:
        int _capacity;
        int _size;
        double* _elements;
    public:
        DynamicArrayDouble(int capacity);
        ~DynamicArrayDouble();
        int get_capacity();
        int get_size();
        void add(double element);
        bool contains(double element);
        bool remove(double element);
        double* asArray();
};
```

Template Class

```
template<typename T>
class DynamicArray {
    private:
        int _capacity;
        int _size;
        T* _elements;
    public:
        DynamicArray(int capacity);
        ~DynamicArray();
        int get_capacity();
        int get_size();
        void add(T element);
        bool contains(T element);
        bool remove(T element);
        T* asArray();
};
```

Template Class

```
template<typename T>
DynamicArray<T>::DynamicArray(int capacity):_capacity(capacity),
_size(0), _elements(new T[_capacity]) {}

template<typename T>
DynamicArray<T>::~DynamicArray(){ delete [] _elements;}

template<typename T>
int DynamicArray<T>::get_capacity(){return _capacity;}

template<typename T>
int DynamicArray<T>::get_size() {return _size;}
```

Template Class

```
template <typename T>
void DynamicArray<T>::add(T element) {
        if(_size<_capacity) {
            _elements[_size++] = element;
            }
        else {
            throw string("DArray is full!");
        }
}
```

Template Class

```
template <typename T>
bool DynamicArray<T>::contains(T element) {
    for(int i=0; i<_size; i++) {
        if(_elements[i] == element) {
            return true;
        }
    }
    return false;
}
```

Template Class

```
template <typename T>
bool DynamicArray<T>::remove(T element) {
    for(int i=0; i<_size; i++) {
        if(_elements[i] == element) {
            for(int j=i; j<_size-1; j++) {
             _elements[j] = _elements[j+1];
            }
         _size--;
         return true;
        }
    }
    return false;
}
```

Template Class

```
template <typename T>
T* DynamicArray<T>::asArray() {
      T* copy = new T[_size];
      for(int i=0; i<_size; i++) {
        copy[i]=_elements[i];
      }
      return copy;
}
```

Template Class

```
int main() {
        int capacity = 10;
        DynamicArray<int> intDArrayT(capacity);
        DynamicArray<double> doubleDArrayT(capacity);
        for(int i=0; i<capacity; i++) {
            intDArrayT.add(i+5);
            doubleDArrayT.add(i+5.5);
          }
         int* intArray = intDArray.asArray();
         double* doubleArray = doubleDArray.asArray();
         return 0;
}
```