# C/C++ Program Design

## Lab 13, Composition & Template

廖琪梅，王大兴

# Composition and Template

- Class Objects as members
- Class templates

# Class Containment(Composition)

Using class members that are themselves objects of another class is referred to as ***containment*** or ***composition*** or ***layering***.

**Containment** is typically used to implement ***has-a*** relationship, that is, relationship for which the new class has an object of another class.

```cpp
class Engine
{
private:
    int cylinder;
public:
     Engine(int nc) :cylinder(nc) { cout << "Contructor:Engine(int)\n"; }

    void start()
    {
        cout << getCylinder() <<" cylinder engine started" << endl;
    }

    int getCylinder() { return cylinder; }

    ~Engine() { cout << "Destructor:~Engine()\n"; }
};
```

Define an object of Engine as Car's attribute

Initialize the object by its own constructor via initialization list in Car's constructor

```cpp
 class Car
{
 private:
    Engine eng; // Car has-an Engine
 public:
     Car(int n = 4) : eng(n) { cout << "Constructor:Car(int=)\n"; }

    void start()
    {
       cout << "car with " << eng.getCylinder() << " cylinder engine started" << endl;
       eng.start();
    }

    ~Car() { cout << "Destructor:~Car()\n"; }
};
```

```cpp
#include "car.h"

int main( )
{
    Car car1;
    Car car2(8);

    car1.start();
    car2.start();

    return 0;
}
```

Call the Car's default constructor

First, constructs the object in Car class then, constructs the Car object

```
Contructor:Engine(int)
Constructor:Car(int=)
Contructor:Engine(int)
Constructor:Car(int=)
car with 4 cylinder engine started
4 cylinder engine started
car with 8 cylinder engine started
8 cylinder engine started
Destructor:~Car()
Destructor:~Engine()
Destructor:~Car()
Destructor:~Engine()
```

When an object is destructed, the complier first destructs Car's object, and then destructs the composition object in Car class.

# Template

The C++ programming language offers a powerful feature known as templates, which enable functions and classes to operate on generic types.

This allows a function or class to be designed to work seamlessly with a diverse range of data types, without the need for tedious and repetitive rewrites for each type.

By leveraging templates, C++ code can be made more concise, modular, and easier to maintain, while also boosting its flexibility and versatility.

# Class Templates

Much like function templates, class templates offer the ability to define a single class that can be utilized with a variety of data types.

or <class T>

```
template<typename T>
class class_name
{
    // class definition

};
```

multiple parameters

```
template<typename T1, typename T2, typename T3>
class class_name
{
    // class definition
};
```

nontype template parameter

```
template<typename T, size_t size>
class array
{
    T arr[size];

};
```

multiple and default parameters

```
template<typename T1, typename T2, typename T3 = char>
class class_name
{
    // class definition
};
```

# Class Templates

## 1. Template Definition

```
#ifndef CLASSTEMPLATE_MATRIX_H
#define CLASSTEMPLATE_MATRIX_H

#define MAXSIZE 5

template<class T>
class Matrix
{
private:
    T matrix[MAXSIZE];
    size_t size;

public:
    // constructor Initialize all the values of matrix to zero
    Matrix();  // Set size to MAXSIZE

    //print Function
    void printMatrix();

    // Setter Functions
    void setMatrix(T[]);    //set the array to what is sent
    void addMatrix(T[]);   //add an array to matrix

    // No destructor needed
};

#endif //CLASSTEMPLATE_MATRIX_H
```

data in matrix class

## 2. Member Function Definition

To refer to the class in a generic way you must include the placeholder in the class name like this:

**template <class T>**
**return_type class_name <T>::**
**function_name(parameter_list,…)**

```cpp
template<class T>
Matrix<T>::Matrix():size(MAXSIZE) { }

template<class T>
void Matrix<T>::setMatrix(T array[])
{
    for (size_t i = 0; i < size; i++)
        matrix[i] = array[i];
}

template<class T>
void Matrix<T>::printMatrix()
{
    for (size_t i = 0; i < size; i++)
        std::cout << matrix[i] << "  ";
    std::cout << std::endl;
}

template<class T>
void Matrix<T>::addMatrix(T otherArray[])
{
    for (size_t i = 0; i < size; i++)
        matrix[i] += otherArray[i];
}
```

## 3. Class Instantiation

To make an instance of a class you use this form:

**class_name <type> variablename;**

For example, to create a Matrix with **int** you would type:

**Matrix<int> m;**

Matrix<int> becomes **the name of a new class**.

```cpp
#include <iostream>
#include "Matrix.h"

int main()
{
    int a[MAXSIZE]{ 1,2,3,4,5 };

    Matrix<int> m;

    m.setMatrix(a);
    m.printMatrix();

    return 0;
}
```

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

template<class T, size_t size>
class A
{
private:
    T arr[size];  // automatic array initialization.

public:
    void insert()
    {
        int i = 1;
        for (int j = 0; j < size; j++)
        {
            arr[j] = i;
            i++;
        }
    }

    void display()
    {
        for (int i = 0; i < size; i++)
        {
            std::cout << arr[i] << " ";
        }
    }
};
```

non-type template parameter

```cpp
int main()
{
    A<int, 10> t1;
    t1.insert();
    t1.display();
    return 0;
}
```

Non-type template parameters can be strings, constant expression and built-in types.

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

template<class T1, class T2>
class A
{
private:
    T1 a;
    T2 b;
public:
    A(T1 x, T2 y):a(x),b(y) { }

    void display()
    {
        std::cout << "Values of a and b are : " << a << " ," << b << std::endl;
    }
};

int main()
{
    A<int, float> d(5, 6.5);
    d.display();
    return 0;
}
```

multiple parameters

multiple and default parameters

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

template <class T, class U, class V = char>
class MultipleParameters
{
private:
    T var1;
    U var2;
    V var3;

public:
    MultipleParameters(T v1, U v2, V v3) : var1(v1), var2(v2), var3(v3) {}  // constructor

    void printVar() {
        cout << "var1 = " << var1 << endl;
        cout << "var2 = " << var2 << endl;
        cout << "var3 = " << var3 << endl;
    }
};
int main()
{
    // create object with int, double and char types
    MultipleParameters<int, double> obj1(7, 7.7, 'c');
    cout << "obj1 values: " << endl;
    obj1.printVar();

    // create object with int, double and bool types
    MultipleParameters<double, char, bool> obj2(8.8, 'a', false);
    cout << "\nobj2 values: " << endl;
    obj2.printVar();

    return 0;
}
```

```
obj1 values:
var1 = 7
var2 = 7.7
var3 = c
```

```
obj2 values:
var1 = 8.8
var2 = a
var3 = 0
```

# Template specialization

In some cases, it isn't possible or desirable for a template to define exactly the same code for any type. In such cases you can define a *specialization* of the template for that particular type. When a user instantiates the template with that type, the compiler uses the specialization to generate the class, and for all other types, the compiler chooses the more general template. Specializations in which all parameters are specialized are ***complete specializations***. If only some of the parameters are specialized, it is called a ***partial specialization***.

A template specialization of a class requires a ***primary*** *class* and a type or parameters to specialize. A specialized template class behaves like a new class. There is no inheritance from the primary class. It doesn't share anything with the primary template class, except the name. Any and all methods and members will have to be implemented.

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

template <class Z>
class Test
{
public:
    Test()
    {
        cout << "It is a General template object \n";
    }
};
```

primary class

class specialization

```cpp
template <>
class Test <int>
{
public:
    Test()
    {
        cout << "It is a Specialized template object\n";
    }
};

int main()
{
    Test<int> p;
    Test<char> q;
    Test<float> r;
    return 0;
}
```

```
It is a Specialized template object
It is a General template object
It is a General template object
```

Class templates can be **partially specialized**, and the resulting class is still a template. Partial specialization allows template code to be partially customized for specific types in situations, such as:

(1) A template has multiple types and only some of them need to be specialized. The result is a template parameterized on the remaining types.

(2) A template has only one type, but a specialization is needed for pointer, reference, pointer to member, or function pointer types. The specialization itself is still a template on the type pointed to or referenced.

```cpp
#pragma once
#include <iostream>
using namespace std;

template<class T1, class T2>
class Data
{
private:
    T1 a;
    T2 b;

public:
    Data(T1 m, T2 n) :a(m), b(n)
    {
        cout << "Original class template Data<T1,T2>\n";
    }

    void display()
    {
        cout << "Original class template Data:" << a << "," << b << endl;
    }
};
```

primary class

class partial specialization

```cpp
template<class T1>
class Data<T1, char>
{
private:
    T1 a;
    char b;

public:
    Data(T1 m, char c) :a(m), b(c)
    {
        cout << "Partial specialization Data<T1,char>\n";
    }

    void display()
    {
        cout << "Partial specialization Data:" << a << "," << b << endl;
    }
};
```

```cpp
#include <iostream>
#include "partial.h"

int main()
{
    Data<int, int> d_original(5, 8);
    d_original.display();

    Data<double,char> d_special(3.4, 'A');
    d_special.display();

    return 0;
}
```

```
Original class template Data<T1,T2>
Original class template Data:5,8
Partial specialization Data<T1,char>
Partial specialization Data:3.4,A
```

```cpp
template <class T>
class Bag
{
    T* elem;
    int size;
    int max_size;

public:
    Bag() : elem(0), size(0), max_size(1) {}
    void add(T t)
    {
        T* tmp;
        if (size + 1 >= max_size) {
            max_size *= 2;
            tmp = new T[max_size];
            for (int i = 0; i < size; i++)
                tmp[i] = elem[i];
            tmp[size++] = t;
            delete[] elem;
            elem = tmp;
        }
        else
            elem[size++] = t;
    }

    void print() {
        for (int i = 0; i < size; i++)
            cout << elem[i] << " ";
        cout << endl;
    }
};
```

primary class
Original template class

```cpp
template <class T>
class Bag<T*>
{
    T* elem;
    int size;
    int max_size;

public:
    Bag() : elem(0), size(0), max_size(1) {}
    void add(T* t)
    {
        T* tmp;
        if (t == NULL) {    // Check for NULL
            cout << "Null pointer!" << endl;
            return;
        }

        if (size + 1 >= max_size) {
            max_size *= 2;
            tmp = new T[max_size];
            for (int i = 0; i < size; i++)
                tmp[i] = elem[i];
            tmp[size++] = *t;  // Dereference
            delete[] elem;
            elem = tmp;
        }
        else
            elem[size++] = *t; // Dereference
    }
    void print()
    {
        for (int i = 0; i < size; i++)
            cout << elem[i] << " ";
        cout << endl;
    }
};
```

class partial specialization
template partial specialization
for pointer types

The values that are pointed to are added. If there is no partial specialization, only the pointers themselves are added.

# Bringing it All Together

**Class templates and their member function templates should be declared in .h/.hpp.**

# Template or Inheritance

Templates are powerful, but they are not magical. When you design or use a template you should be aware of what operations the data types you will use need to support.

**Template** should be used to **generate a set of classes** where the object type **does not affect** the function behavior of the class.

**Inheritance** should be used **used on a set of classes** where the object type **does affect** the function behavior of the class.

# Exercises

1. The declarations of Point class and Line class are as follows:

```
class Point {
private:
    double x, y;

 public:
    Point(double newX, double newY) ;

    Point(const Point& p);

    double getX() const;
    double getY() const;

};
```

```
class Line
{
private:
    Point p1, p2;
    double distance;

public:
     Line(Point xp1, Point xp2);
     Line(const Line& q);
    double getDistance() const;
};
```

```
int main()
{
    Point a(8, 9),b(1,2);
    Point c = a;
    cout << "point a: x = " << a.getX() << ", y = " << a.getY() << endl;
    cout << "point b: x = " << b.getX() << ", y = " << b.getY() << endl;
    cout << "point c: x = " << c.getX() << ", y = " << c.getY() << endl;

    cout << "-----------------------------------------" << endl;
    Line line1(a, b);
    cout << "line1's distance:" << line1.getDistance() << endl;

    Line line2(line1);
   cout << "line2's distance:" << line2.getDistance() << endl;

   return 0;
}
```

Implement the member functions of the two classes and then run the program to test the classes.

2. A template class named **Pair** is defined as follows. Please implement the overloading **operator<** which compares the value of the key, if this->key is smaller than that of p.key, return true. Then define a friend function to overload **<< operator** which displays the Pair's data members. At last, run the program. The output sample is as follows:

```
#include <iostream>
#include <string>
using namespace std;
template <class T1,class T2>
class Pair
{
public:
   T1 key;
   T2 value;
   Pair(T1 k,T2 v):key(k),value(v) { };
   bool operator < (const Pair<T1,T2> & p) const;

};
```

```
int main()
{
   Pair<string,int> one("Tom",19);
   Pair<string,int> two("Alice",20);

   if(one < two)
      cout << one;
   else
      cout << two;

   return 0;
}
```

Output:

```
Alice    20
```

3. There is a definition of a template class **Dictionary**. Please write a template partial specialization for Dictionary class whose **Key** is specified to be **int,** and add a member function named sort() which sorts the elements in dictionary in ascending order. At last, run the program. The output sample is as follows:

```cpp
template <class Key, class Value>
class Dictionary {
  Key* keys;
  Value* values;
  int size;
  int max_size;
public:
  Dictionary(int initial_size) :  size(0) {
    max_size = 1;
    while (initial_size >= max_size)
      max_size *= 2;
    keys = new Key[max_size];
    values = new Value[max_size];
  }
  void add(Key key, Value value) {
    Key* tmpKey;
    Value* tmpVal;
    if (size + 1 >= max_size) {
      max_size *= 2;
      tmpKey = new Key [max_size];
      tmpVal = new Value [max_size];
      for (int i = 0; i < size; i++) {
        tmpKey[i] = keys[i];
        tmpVal[i] = values[i];
      }
      tmpKey[size] = key;
      tmpVal[size] = value;
      delete[] keys;
      delete[] values;
      keys = tmpKey;
      values = tmpVal;
    }
    else {
      keys[size] = key;
      values[size] = value;
    }
    size++;
  }

  void print() {
    for (int i = 0; i < size; i++)
      cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
  }

~Dictionary()
  {
     delete[] keys;
     delete[] values;
  }

};
```

```cpp
int main()
{
   Dictionary<const char*, const char*> dict(10);
   dict.print();
   dict.add("apple", "fruit");
   dict.add("banana", "fruit");
   dict.add("dog", "animal");
   dict.print();

   Dictionary<int, const char*> dict_specialized(10);
   dict_specialized.print();
   dict_specialized.add(100, "apple");
   dict_specialized.add(101, "banana");
   dict_specialized.add(103, "dog");
   dict_specialized.add(89, "cat");
   dict_specialized.print();
   dict_specialized.sort();
   cout << endl << "Sorted list:" << endl;
   dict_specialized.print();

  return 0;
}
```

Output:

```
{apple, fruit}
{banana, fruit}
{dog, animal}
{100, apple}
{101, banana}
{103, dog}
{89, cat}

Sorted list:
{89, cat}
{100, apple}
{101, banana}
{103, dog}
```