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.

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

data in **Line** class has **Point** objects

Example:

```cpp
#pragma once
// Declare Point class
class Point {
private:
    double x, y;

public:
    Point(double newX, double newY)
    {
        x = newX;
        y = newY;
    }

    Point(const Point& p);

    double getX() const { return x; }
    double getY() const { return y; }

};

Point::Point(const Point & p)
{
    x = p.x;
    y = p.y;
}
```

data in **Point** class

constructor

copy constructor

```cpp
#pragma once
// Declare Line class, include Point object

#include <iostream>
#include <cmath>
#include "Point.h"

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

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

};

Line::Line(Point xp1, Point xp2) :p1(xp1), p2(xp2)
{
    double x = p1.getX() - p2.getX();
    double y = p1.getY() - p2.getX();

    distance = sqrt(x * x + y * y);
}

Line::Line(const Line& q) :p1(q.p1), p2(q.p2)
{
    std::cout << "calling the copy constructor of Line" << std::endl;
    distance = q.distance;
}
```

data in **Line** class has **Point** objects

constructor and copy constructor

Initialize object first by initialization list

Initialize object first by initialization list

```cpp
#include <iostream>
#include "Point.h"
#include "Line.h"

using namespace std;

void func1(Point p)
{
    cout << "fun1:" << p.getX() << ", " << p.getY() << endl;
}

Point func2()
{
    Point a(1, 2);
    return a;
}

int main()
{
    // Point
    Point a(8, 9);
    Point b = a;
    cout << "test point b: x = " << b.getX() << ", y = " << b.getY() << endl;
    func1(b);
    b = func2();
    cout << "test point b: x = " << b.getX() << ", y = " << b.getY() << endl;

    cout << "-------------------------------------------" << endl;
    Point m(3, 4), n(5, 6);
    Line line1(m, n);
    cout << "line1:" << line1.getDistance() << endl;

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

    return 0;

}
```

Invoke Point's constructor

Invoke Point's copy constructor

Invoke Line's constructor

Invoke Line's copy constructor

```
test point b: x = 8, y = 9
fun1:8, 9
test point b: x = 1, y = 2
-------------------------------------------
line1:2.23607
calling the copy constructor of Line
line2:2.23607
```

```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"; }
};

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"; }
};
```

Define an object of Engine as Car's attribute

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

```cpp
#include "car.h"

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

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

    return 0;
}
```

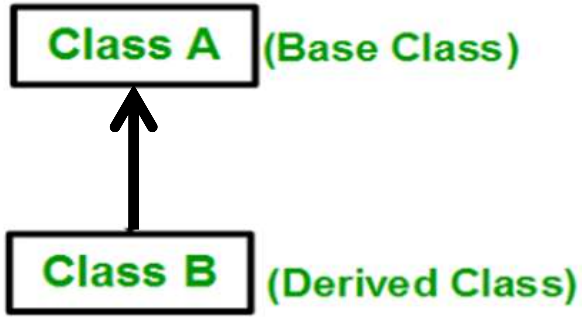
Call the Car's default constructor
First, constructs the object in Car class then, constructs the Car object

```
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.

Type of Inheritance

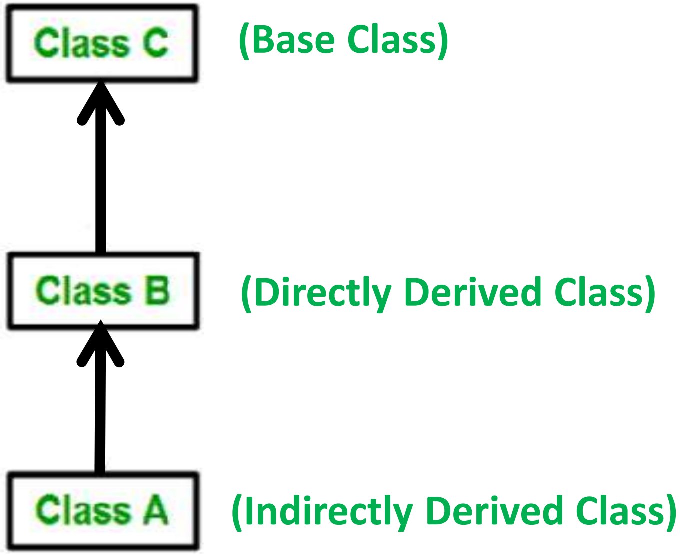
1. Single Inheritance



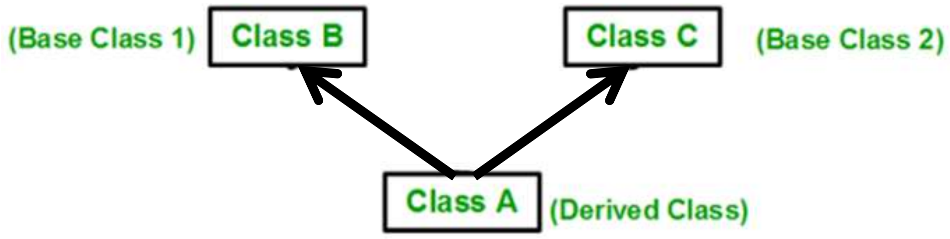
Syntax:

```
class derived_name : access_mode base_class
{
  //body of subclass
};
```

2. Multilevel Inheritance



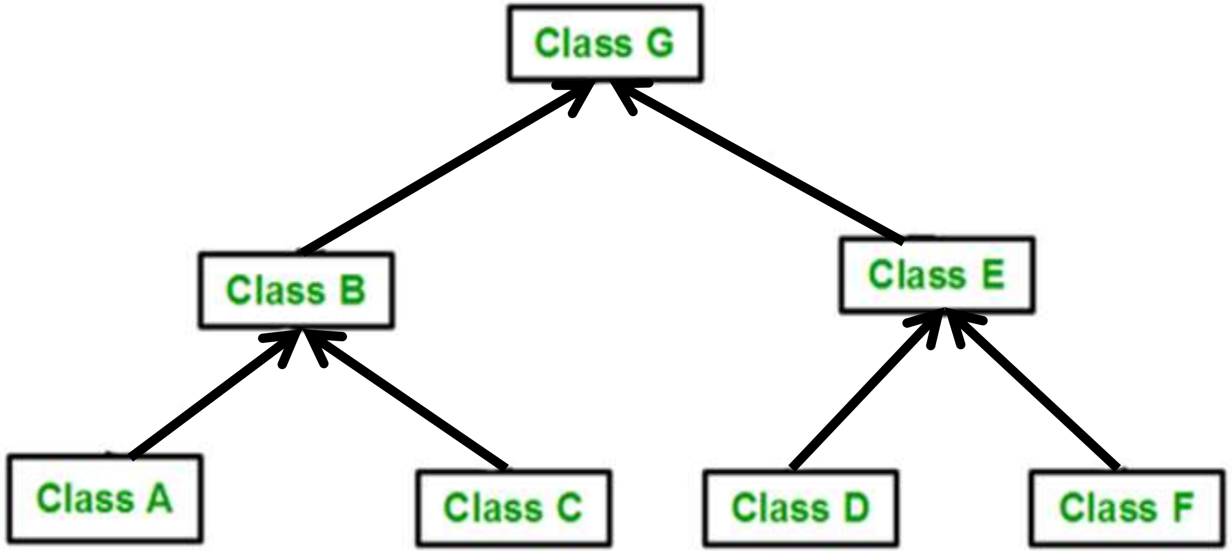
3. Multiple Inheritance(MI)



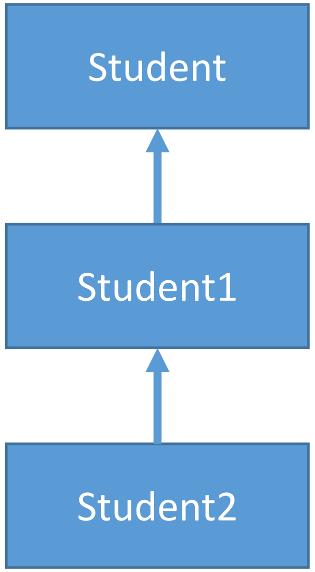
Syntax:

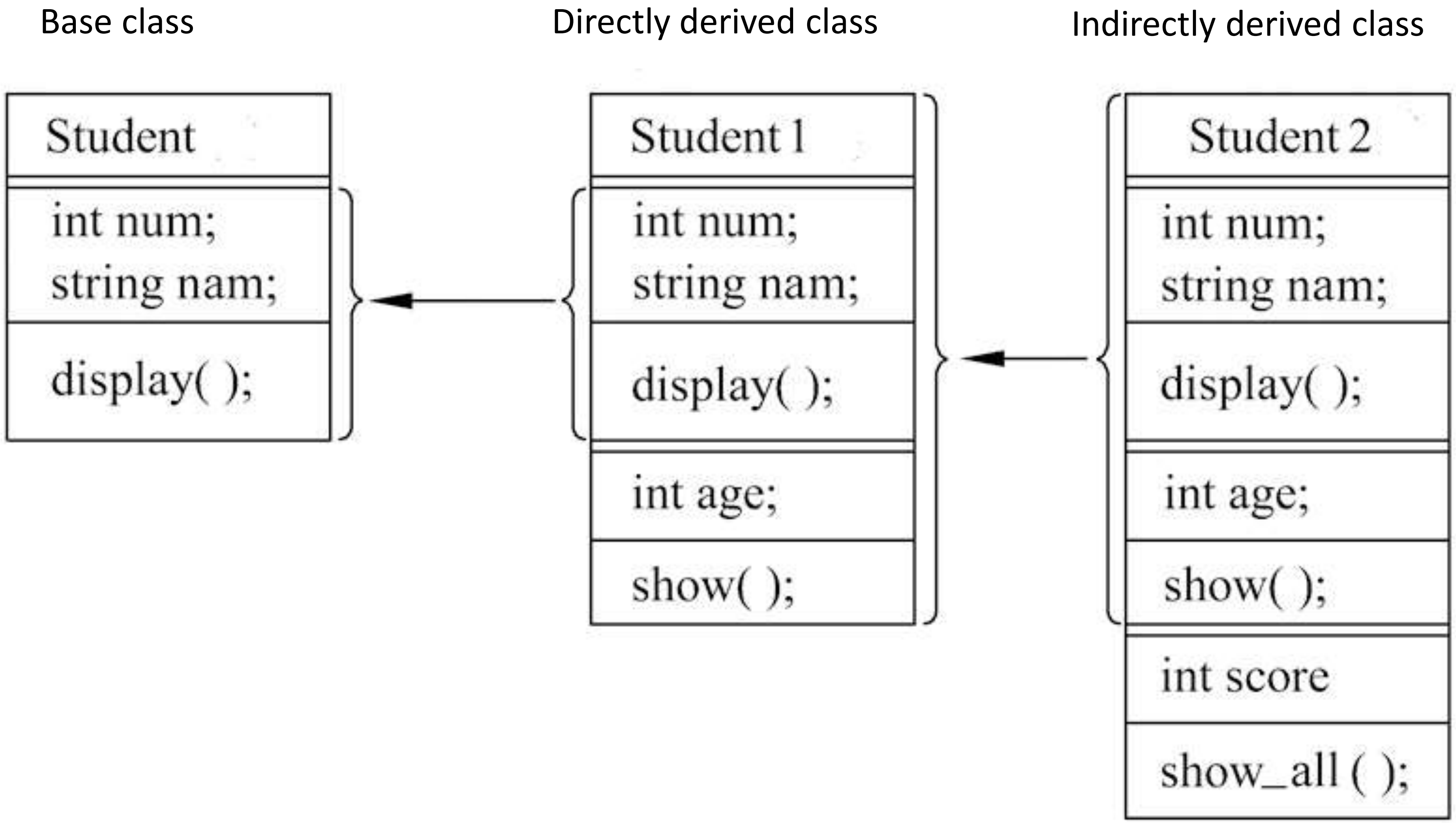
```
class derived_name : access_mode base_class1, access_mode base_class2,
....
{
  //body of subclass
};
```

4. Hierarchical Inheritance



Multilevel inheritance example:





```cpp
// Declare base class Student
#pragma once
#include <iostream>
#include<string>

using namespace std;
class  Student
{
protected:
    int num;
    string name;

public:
    Student(int n, const string& nam):num(n),name(nam){}

    void display()
    {
        cout << "num:" << num << endl;
        cout << "name:" << name << endl;
    }
};
```

data in class **Student**

constructor of class **Student**

```cpp
// Declare the directly derived class Student1
#pragma once
#include "Student.h"

class  Student1 : public  Student
{
private:
    int age;

public:

    Student1(int n, const string& nam, int a) : Student(n, nam)
    {
        age = a;
    }
    void show()
    {
        display(); // call the base class funciton
        cout << "age: " << age << endl;
    }
};
```

data in class **Student1**

constructor of **Student1**

```cpp
// Declare the indirectly derived class Student2
#pragma once
#include "Student1.h"

class  Student2 :public  Student1
{
private:
    int score;
public:
    Student2(int n, const string& nam, int a, int s) : Student1(n, nam, a)
    {
        score = s;
    }
    void show_all()
    {
        show();    //call the directly derived class

        cout << "score:" << score << endl;
    }
};
```

data in class **Student2**

constructor of **Student2**

```cpp
#include "Student2.h"

int main()
{
    Student2 stud(10010, "Li", 17, 89);
    stud.show_all();    //show all the data of class Student2
    return 0;
}
```

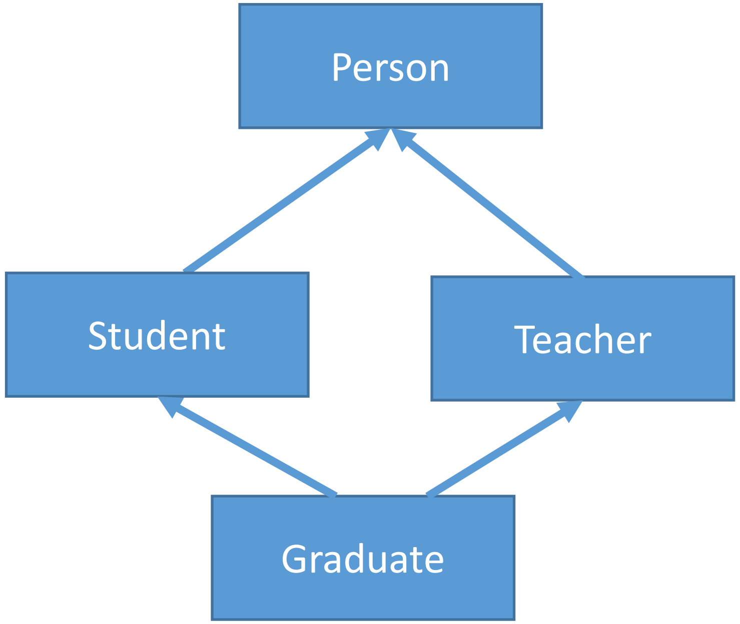
```
num:10010
name:Li
age: 17
score:89
```

Multiple Inheritance(MI)

MI describes a class that has more than one immediate base class. As with single inheritance, public MI should express an **is-a** relationship.

```
class Graduate : public Teacher, public Student { };
```

you must qualify each base class with the keyword **public**



Graduate has two copies of **Person** objects. Because both **Student** and **Teacher** inherit the **Person** component, **Graduate** winds up with two **Person** components.

Virtual Base Classes

Virtual base classes allow an object derived from multiple bases that themselves share a common base to **inherit just one object** of that shared base class.

```
class Student : virtual public Person {  };

class Teacher : virtual public Person {   };
```

virtual and public can appear in either order

```
class Graduate : public Teacher, public Student {  };
```

A Graduate object will contain a single copy of the Person object.

Constructor

With nonvirtual base classes, the only constructors that can appear in an initialization list are constructors for the immediate base classes. But these constructors can, in turn, pass information on to their bases.

```
Student2(int n, const string& nam, int a, int s) : Student1(n, nam, a)
{
    score = s;
}
```

Just invoke immediate base class constructor, need not invoke the upper base class constructor

This automatic passing of information doesn't work if a class is a virtual base class.

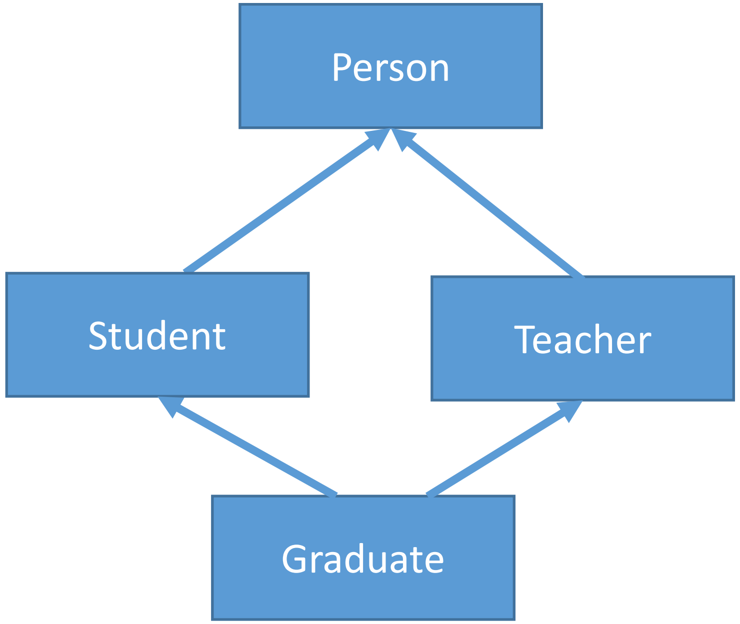
```
Graduate(const string& nam, char g, int a, const string& t, float sco, float w)
    :Person(nam, g, a), Teacher(nam, g, a, t), Student(nam, g, a, sco), wage(w) { }
```

Invoke the Person class(the top-level base class) constructor explicitly. If you don't invoke the constructor, the compiler will invoke its default constructor.

invoke the Teacher and Student class (immediate base classes)constructors

initialize its own data

Multiple inheritance example:




```
class Person
{
protected:
    string  name;
    char   gender;
    int    age;
public:

    Person(const string& nam, char g, int a)
       :name(nam),gender(g),age(a){}

};
```

data in class **Person**

constructor of **Person**

```cpp
//Declare the directly derived class Teacher form Person
#pragma once
#include <cstring>
#include "Person.h"

class Teacher : virtual public Person
{
protected:
    string  title;

public:
    Teacher(const string& nam, char s, int a, const string& t) :Person(nam, s, a), title(t) {}

};
```

data in class **Teacher**

```cpp
//Declare the directly derived class Student form Person
#pragma once
#include "Person.h"

class Student : virtual public Person
{
protected:
    float   score;

public:

    Student(const string& nam, char s, int a, float sco) :Person(nam, s, a), score(sco) {}

};
```

data in class **Student**

```cpp
#include "Student.h"

class Graduate : public Teacher, public Student
{
private:
    float wage;

public:
    Graduate(const string& nam, char g, int a, const string& t, float sco, float w)
      :Person(nam, g, a), Teacher(nam, g, a, t), Student(nam, g, a, sco), wage(w) { }

    void show()
    {
        cout << "name:" << name << endl;
        cout << "age:" << age << endl;
        cout << "gender:" << gender << endl;
        cout << "score:" << score << endl;
        cout << "title:" << title << endl;
        cout << "wages:" << wage << endl;
    }
};
```

data in class **Graduate**

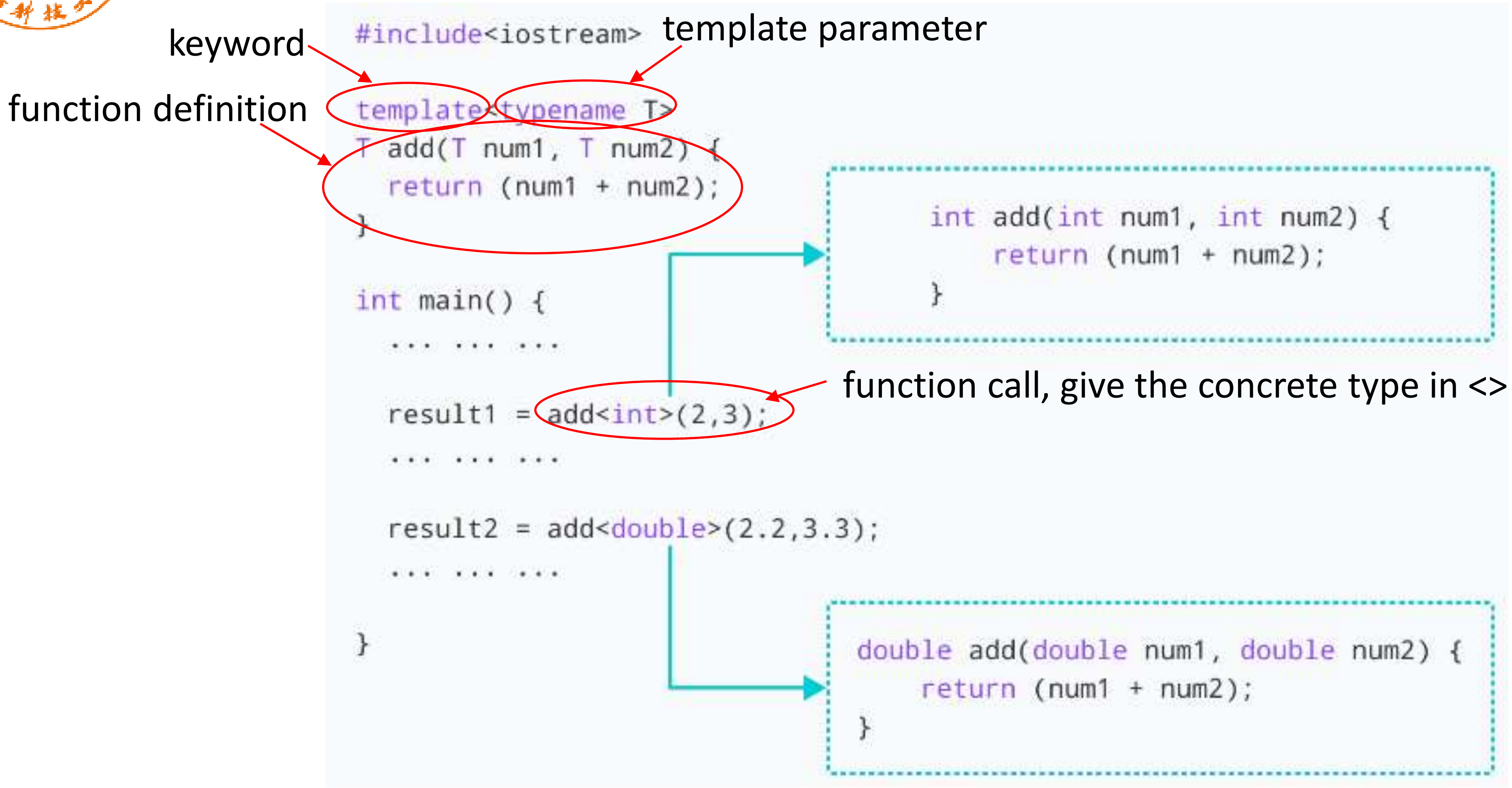
```cpp
#include "Graduate.h"

int main()
{
    Graduate grad1("Wang-li", 'f', 24, "assistant", 89.5, 1234.5);
    grad1.show();

    return 0;
}
```

```
name:Wang-li
age:24
gender:f
score:89.5
title:assistant
wages:1234.5
```

Template

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

Function Templates



To create a prototype for a template function remember to include the template specifier like this:

template <typename T>
T add(T, T);

Class Templates

Similar to function templates, we can use class templates to create a single class to work with different 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 argument

```
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. Class 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;

Taken together Matrix becomes **the name of a new class**.

```
#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] << " ";
        }
    }
};
```

nontype template argument

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

Nontype template arguments can be strings, constant expression and built-in types.

multiple parameters

```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 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";
    }
};

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;
}
```

primary class

class specialization

```
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:

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

primary class

```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;
    }
};
```

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) {}
```

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) {}
```

class partial specialization

template partial specialization for pointer types

Bringing it All Together

Normally when you write a C++ class you break it into two parts: **a header file** with the interface, and a **.cpp file** with the implementation. With templates this doesn’t work so well because the compiler needs to see the definition of the member functions to create new instance of the template class. Some compilers are smart enough to figure out what to do, but some aren’t. These are usually the most efficient way to use templates. **We recommend that template classes be declared and implemented in .h files to ensure proper linking. Or you can include .cpp file in your main program instead of including .h file.**

A Word of Warning

Templates are powerful, but they are not magical. They do not give data types features that they did not have before. When you design or use a template you should be aware of what operations the data types you will use need to support.

Exercise:

Define a 2D array named **Matrix** using template type, implement the member functions of Matrix and the function **useMatrixTemplate**, make the program run as the sample.

```
#define MAXROWS 5
#define MAXCOLS 5

template<class T>
class Matrix
{
private:
    T matrix[MAXROWS][MAXCOLS];
    size_t rows;
    size_t cols;

public:
    // constructor Initialize all the values of matrix to zero
    Matrix(size_t row = MAXROWS, size_t col = MAXCOLS);  // Set rows to MAXROWS and cols to MAXCOLS

    //print Function
    void printMatrix();

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

    // No destructor needed
};
```

```cpp
#include <iostream>
#include <string>
#include "matrix.h"

using namespace std;

template<typename T1>
void useMatrixTemplate(Matrix<T1>& M, T1 array1[][MAXCOLS], T1 array2[][MAXCOLS]);

int main()
{
    string str1[MAXROWS][MAXCOLS] = { {"Congra","y","ar"},{"alm","don","La"} };
    string str2[MAXROWS][MAXCOLS] = { {"tulations","ou","e"},{"ost","e the","b!"} };

    int num1[MAXROWS][MAXCOLS] = { {1,2,3},{4,5,6} };
    int num2[MAXROWS][MAXCOLS] = { {6,5,4},{3,2,1} };

    Matrix<string> stringMartix(2,3);
    Matrix<int> intMatrix(2,3);

    cout << "Demonstrating with string matrix:" << endl;
    useMatrixTemplate(stringMartix, str1, str2);

    cout << "\nDemonstrating with int matrix:" << endl;
    useMatrixTemplate(intMatrix, num1, num2);

    cout << "\n" << endl;
    return 0;
}
```

implement the function

```
Demonstrating with string matrix:

Matrix set first array
Congra  y  ar
alm  don  La

Matrix incremented by second array
Congratulations  you  are
almost  done the  Lab!

Demonstrating with int matrix:

Matrix set first array
1  2  3
4  5  6

Matrix incremented by second array
7  7  7
7  7  7
```