C/C++ Program Design

LAB 14

CONTENTS

- Class Objects as members
- Review Function Templates
- Learn Class Templates

2 Knowledge Points

2.1 Class Containment(Composition)

2.2 Type of Inheritance

2.3 Function Templates

2.4 Class Templates

2.1 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 Student
{
private:
    string name;                 // use a string object for name
    valarray<double> scores;     // use a valarray<double> object for scores
    ...
};
```

Initializing Contained Objects

For inherited objects, constructors use the class name in the member initializer list to invoke a specific base-class constructor.

invoke the base-class constructor

```
hasDMA::hasDMA(const hasDMA & hs) : baseDMA(hs) {...}
```

initialize the new data in subclass

For member objects, constructors use the member name.

Using explicit turns off implicit conversions.

```
Student() : name("Null Student"), scores() {}
explicit Student(const std::string & s)
    : name(s), scores() {}
explicit Student(int n) : name("Nully"), scores(n) {}
Student(const std::string & s, int n)
    : name(s), scores(n) {}
Student(const std::string & s, const ArrayDb & a)
    : name(s), scores(a) {}
Student(const char * str, const double * pd, int n)
    : name(str), scores(pd, n) {}
```

```
typedef std::valarray<double> ArrayDb;
```

If you omit the initialization list, C++ uses the default constructors defined for the member objects' classes.

Using an Interface for a Contained Object

The interface for a contained object isn't public, but it can be used within the class methods.

define a function of the Student class

```
double Student::Average() const
{
    if (scores.size() > 0)
        return scores.sum()/scores.size();
    else
        return 0;
}
```

use the object methods

define a friend function

```
// use string version of operator<<()
ostream & operator<<(ostream & os, const Student & stu)
{
    os << "Scores for " << stu.name << ":\n";
    ...
}
```

use the string version of the << operator

Example:

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

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

    Point(Point & p);

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

};

Point::Point(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(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(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 whose has **Point** objects

constructor and copy constructor

Initializes object first

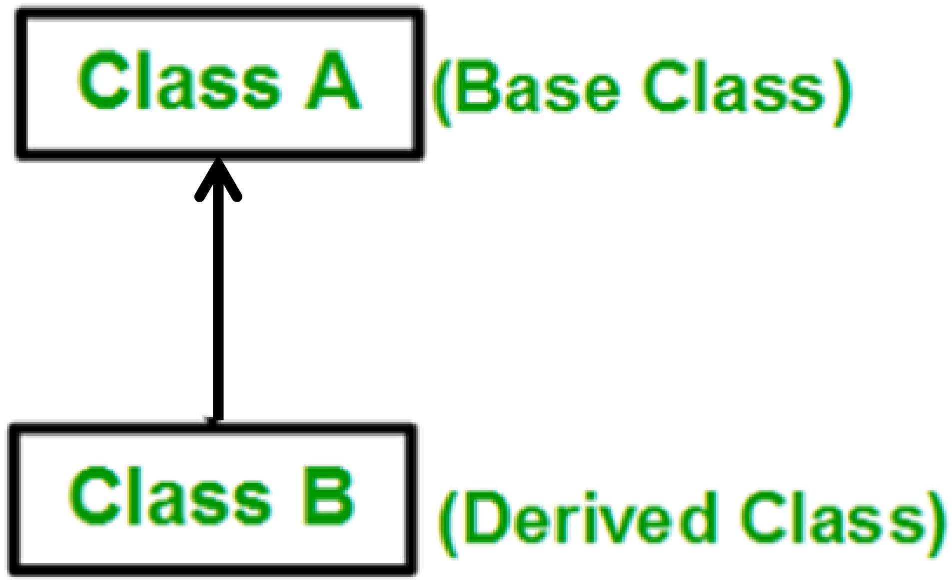
Initializes object first

```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;

}
```

```
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
```

2.2 Type of Inheritance

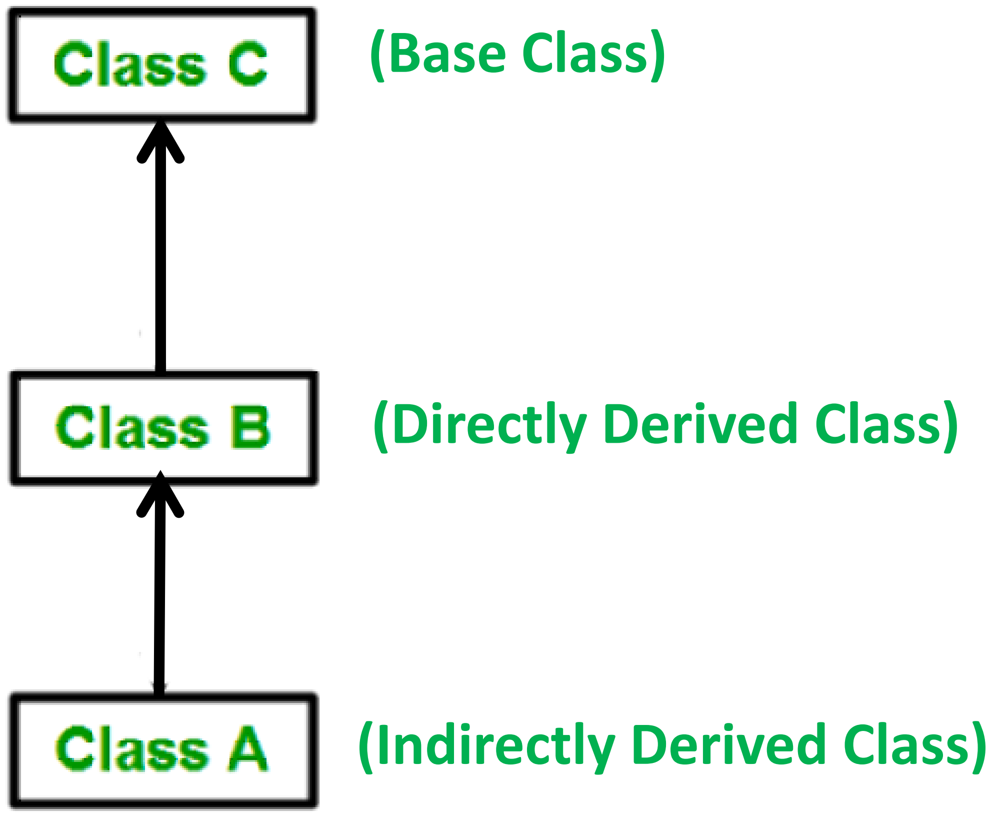
1. Single Inheritance



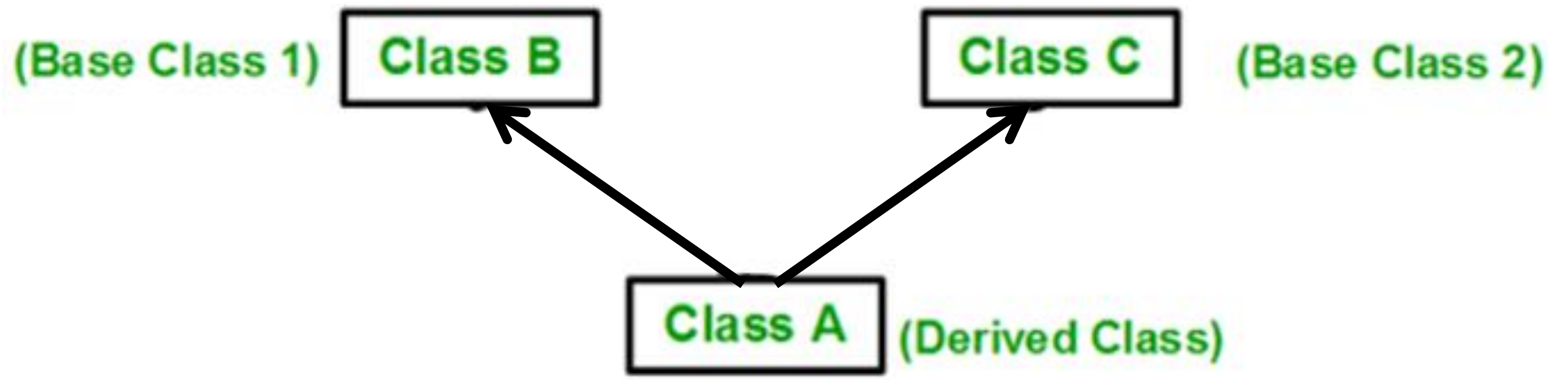
Syntax:

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

2. Multilevel Inheritance



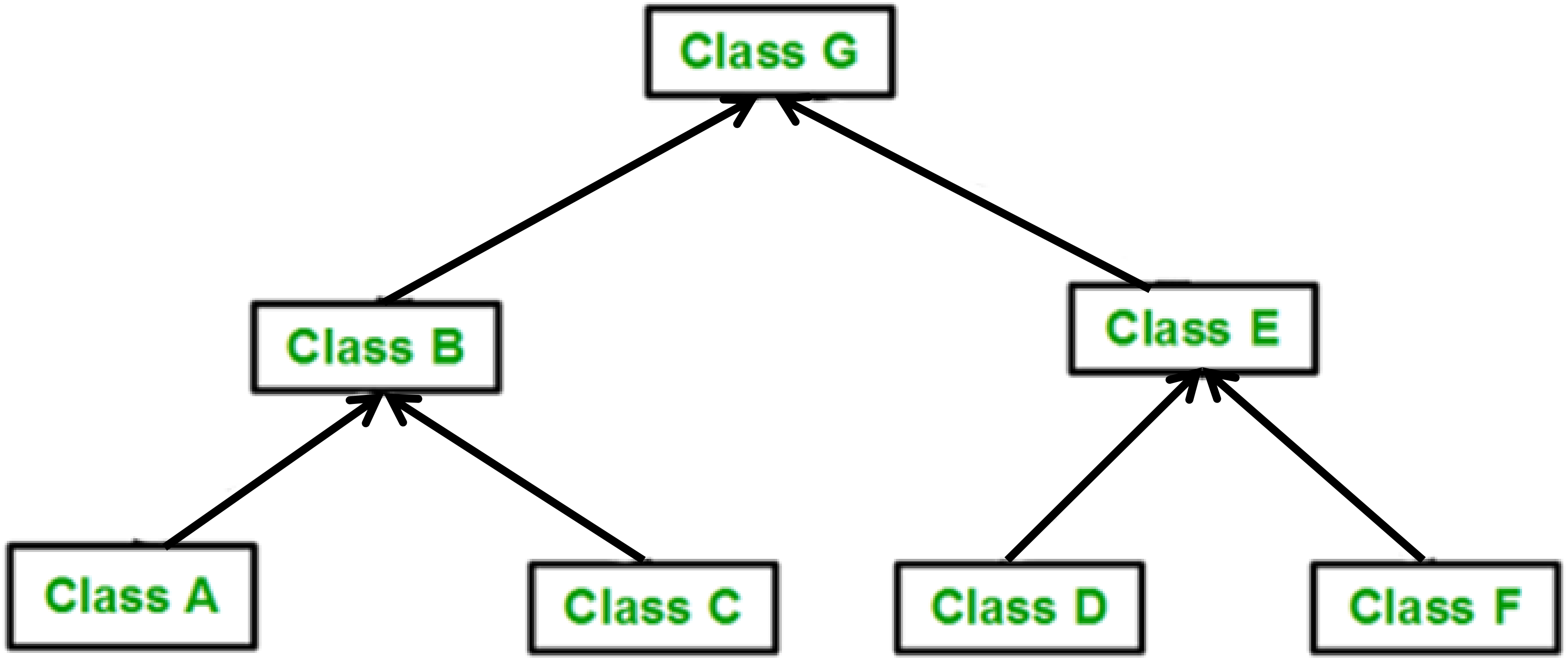
3. Multiple Inheritance(MI)



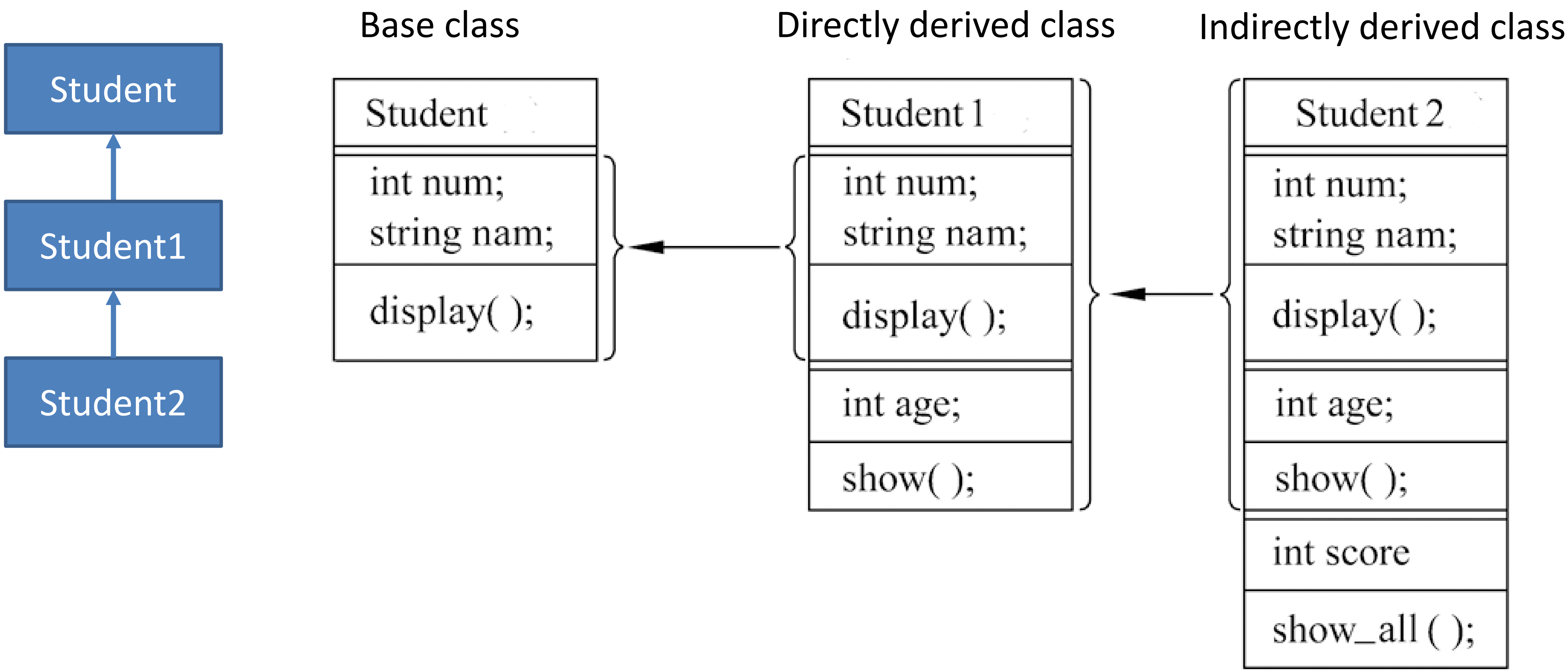
Syntax:

```
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, 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, 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 class **Student1**

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

class  Student2 :public  Student1
{
private:
    int score;

public:
    Student2(int n, 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;
}
```

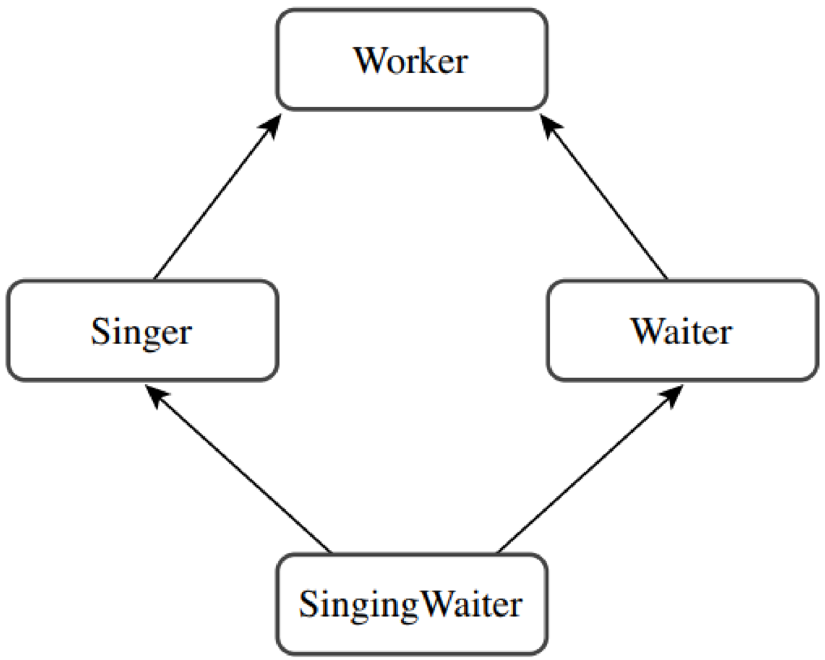
```
num:10010
name:Li
age: 17
score:89
```

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 SingingWaiter : public Waiter, public Singer {...};
```

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



SingingWaiter has two copies of Worker objects. Because both Singer and Waiter inherit a Worker component, SingingWaiter winds up with two Worker 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 Singer : virtual public Worker {...};
class Waiter : public virtual Worker {...};
```

virtual and public can appear in either order

```
class SingingWaiter: public Singer, public  Waiter {...};
```

A SingingWaiter object will contain a single copy of a Worker 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, string nam, int a, int s) : Student1(n, nam, a)
{
    score = s;
}
```

immediate base class

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

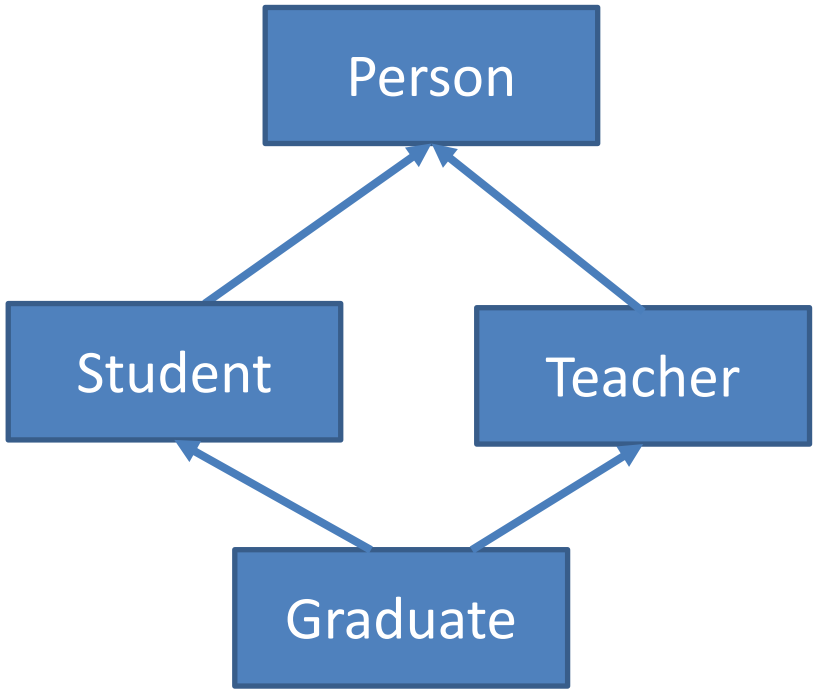
```
SingingWaiter(const Worker & wk, int p = 0, int v = Singer::other)
                 : Worker(wk), Waiter(wk,p), Singer(wk,v) {}
```

invoke the appropriate base constructor explicitly

```
SingingWaiter(const Worker & wk, int p = 0, int v = Singer::other)
                 : Waiter(wk,p), Singer(wk,v) {}  // flawed
```

No invoking the base constructor explicitly means using the default Worker constructor

Multiple inheritance example:




```cpp
// Declare base class Person
#pragma once
#include <iostream>
#include <string>
using namespace std;

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

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

};
```

data in class **Student**

```cpp
//Declare multiple inheritance class Graduate
#pragma once
#include "Teacher.h"
#include "Student.h"

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

public:
    Graduate(string nam, char g, int a, 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
```

2.3 Template

A template is a mechanism in C++ that lets you write a function or a class that uses a generic data type. A placeholder is used instead of a real type and a substitution is done by the compiler whenever a new version of the function or class is needed by your program.

In this lab we will take the simple Matrix ADT and use it for 2D arrays of integers, floats, and strings. Without templates if you needed all three in one program you would need to write three versions. With templates you would need only the template and a few simple calls.

2.3.1 Function Templates

Consider the simple function **print**:

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

void print(int a, int b)
{
    int c = a + b;
    cout << "You gave me " << a << " and " << b << ".\n";
    cout << "Together they make " << c << "." << endl;
}

int main()
{
    print( a: 1, b: 2);
    print( a: 2.6,  b: 3.7);
    print( a: 'A',  b: '1');

    return 0;
}
```

```
You gave me 1 and 2.
Together they make 3.
```

```
You gave me 2 and 3.
Together they make 5.
```

```
You gave me 65 and 49.
Together they make 114.
```

The print function accepts only integer, so the floating numbers 2.6 and 3.7 and the characters 'A' and '1' are coerced to int and are not treated as they should be.

Make print generic with **templates**

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

template <typename T>
void print(T a, T b)
{
    T c = a + b;
    cout << "You gave me " << a << " and " << b << ".\n";
    cout << "Together they make " << c << "." << endl;
}
int main()
{
    string sA = "Oh ";
    string sB = "noes!";
    print( a: 1,  b: 2);
    print( a: 2.6,  b: 3.7);
    print( a: 'A',  b: '1');
    print(sA, sB);

    return 0;
}
```

A new version of print is generated for each data type used as a parameter.

```
You gave me 1 and 2.
Together they make 3.
You gave me 2.6 and 3.7.
Together they make 6.3.
You gave me A and 1.
Together they make r.
You gave me Oh  and noes!.
Together they make Oh noes!.
```

A few notes on the syntax:

- The **template <typename T>** bit can be on the same line as the function type declaration, but it is usually on the line above.
- The value **T** stands for the type the template will be **instantiated with**. **T** can be any valid token, but watch out for **namespace** clashes.
- **T**'s scope is limited to just one function, in the case **print**.
- Notice that we don't need to do anything special to make **print** work for new types.
- If a placeholder appears more than once in a function's parameter list, the types you use in their place in the function call must match.

Note: prototypes

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

```
template <typename T>
void print(T, T);
```

2.3.2 Class Templates

1. Class Definition

```
#ifndef CLASSTEMPLATE_MATRIX_H
#define CLASSTEMPLATE_MATRIX_H

#define MAXROWS 5
#define MAXCOLS 5

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

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

#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,…)

```
template<class T>
Matrix<T>::Matrix()
{
    rows = MAXROWS;
    cols = MAXCOLS;
}

template<class T>
void Matrix<T>::setMatrix(T array[][MAXCOLS])
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
            matrix[i][j] = array[i][j];
    }
}

template<class T>
void Matrix<T>::printMatrix()
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
            std::cout << matrix[i][j] << "  ";

        std::cout << std::endl;
    }
}

template<class T>
void Matrix<T>::addMatrix(T otherArray[][MAXCOLS])
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
            matrix[i][j] += otherArray[i][j];
    }
}
```

2. 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[5][5];

    Matrix<int> m;

    for (int i = 0; i < 5; i++)
        for (int j = 0; j < 5; j++)
            a[i][j] = i+j;
    m.setMatrix(a);
    m.printMatrix();

    return 0;
}
```

```
0  1  2  3  4
1  2  3  4  5
2  3  4  5  6
3  4  5  6  7
4  5  6  7  8
```

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 don'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.**

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.