LAB9
Polymorphism & Template

Department of Electrical Engineering
National Yang Ming Chiao Tung University

Learning Objectives

- Polymorphism
- Virtual functions
- Abstract classes and pure virtual functions
- Function template
- Class template

Polymorphism

- Polymorphism
  - While accessing a member function, the correct version based on the actual calling object is always invoked
  - Namely, the behavior of calling a member function through a pointer/reference may be different polymorphic
- In C++, polymorphism is achieved through
  - Virtual functions
  - Manipulating objects through pointers or references
- A class with virtual functions is called a polymorphic class
- Polymorphism is another cornerstone of OOP

Virtual vs. Non-Virtual Functions (1/2)

- For non-virtual (member) functions
  - Function calls are **STATICALLY bound** (i.e., bound at compile time)

```
class B {
    public: void mf();
};
```

```
class D:public B {
    public: void mf(); //redefine mf();
}
```

```
void f() {
    B b, *pB= &b;
    D d, *pD= &d;
    b.mf();     // statically binding b is of type B call B::mf()
    d.mf();     // statically binding  d is of type D call D::mf()
    pB->mf(); // statically binding  pB is of type B*  call B::mf()
    pD->mf(); // statically binding  pD is of type D*  call D::mf()
    pB= &d;  // ok, D is derived from B
    pB->mf(); // still statically binding pB is of type B* call B::mf()
}             // though pB actually points to d (an object of type D)
```

Virtual vs. Non-Virtual Functions (2/2)

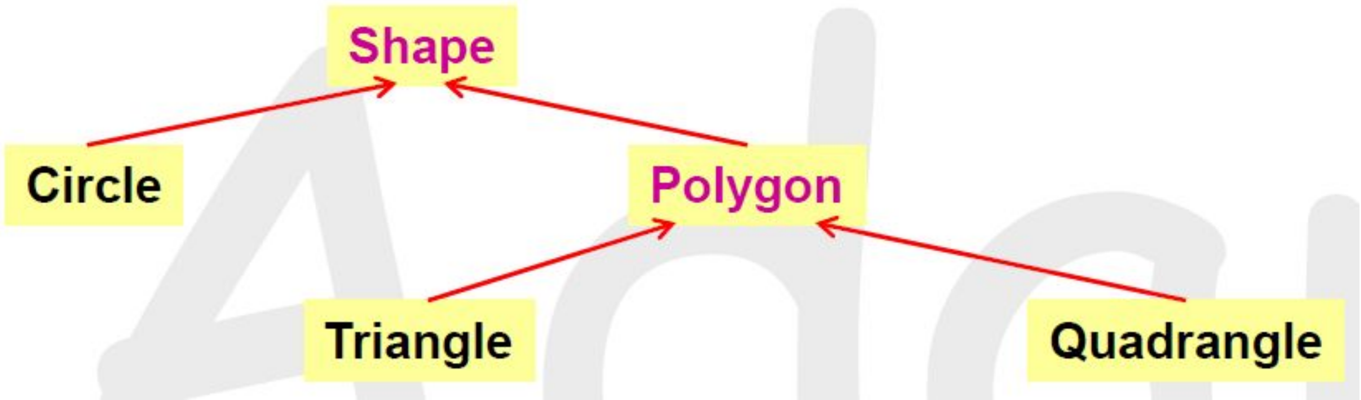
- For virtual (member) functions
  - Must be non-static member functions
  - Function calls are **DYNAMICALLY bound** (i.e., bound at runtime) if they are invoked through pointers or references

```
class B {
    public: virtual void
mf();
};
class D:public B {
    public: void mf(); //override mf();
}
```

```
void f() {
    B b, *pB= &b;
    D d, *pD= &d;
    b.mf();  d.mf();
    pB->mf();  pD->mf();
    pB= &d;  // ok, D is derived from B
    pB->mf();// dynamically binding  pB actually points to d  call D::mf()
}
```

Concrete Class vs. Abstract Class

- Some concepts are concrete and some are abstract



- Abstract classes: Shape and Polygon
  - e.g., no idea how to draw or rotate an arbitrary shape
  - Objects of abstract classes should not exist (they are abstract)
- Concrete classes : Circle, Triangle and Quadrangle
  - Objects of these types can exist
  - They can be drawn, rotated, …

Pure Virtual Functions & Abstract Class


```
class Shape {
    public:
    virtual void rotate(int) = 0;  // pure virtual function
    virtual void draw() = 0;       // pure virtual function
    virtual bool is_closed() = 0; // pure virtual function
    // …                           // only declaration; no definition
};
void f() {
    Shape s; // compilation error! it must be an error, or
    s.draw(); //would be legal ; but draw() is a pure virtual function
};
```

- A class with one or more pure virtual functions is called an abstract class
- No objects of abstract class can be created in C++

Abstract Base Class (ABC)

- Abstract class is always used as a base class (ABC)
  - You cannot create objects of abstract class
  - It only makes sense that some classes derived from it and become concrete by overriding all pure functions

- Abstract class specifies interface requirements

- A class derived from an ABC is still abstract if it doesn't override **ALL** inherited pure virtual functions

The usefulness of function template(1/2)

- C++ allows multiple overloading functions – but need define individually

```
void swap ( int& x ,int& y){          int temp=x;        x=y;  y=temp; }
void swap ( double& x, double& y ){    double temp=x ;x=y,   y=temp; }
void swap (char& x, char& y){      char temp=x;      x=y;  y=temp; }
```

- We want one function to do the things above

```
void swap ( vartype& x ,vartype& y){
        vartype temp=x;
        x=y;
        y=temp;
}
```

The usefulness of function template(2/2)

```
Template <class T >
void swap(T& x , T& y){
        T temp = x ;
        x = y ;
        t  = temp;
}
```

```
void swap(int& x , int& y){
        int temp = x ;
        x = y ;
        t  = temp;
}
```

```
void swap(double& x , double& y){
        double temp = x ;
        x = y ;
        t  = temp;
}
```

```
void swap(char& x , char& y){
        char temp = x ;
        x = y ;
        t  = temp;
}
```

Creating function template (1/2)

```
template < class T >
void swap( T& x , T& y){
        T temp = x ;
        x = y ;
        t  = temp;
}
```

- The key class T does not necessarily mean that T stands for a programmer-created class .□it can be int, char, ….
- Many newer compilers allow you to replace class with typename in the template definition

Creating function template (2/2)

- Function template: functions that use variable types
  - Outline a group of functions that only differ in datatypes of parameters used
- In a function template , at least one argument is generic
- A function template will generate one or more template functions
  - When calling a function template, compiler generates code for different functions as it needs

Overloading function template

- Can overload function templates only when each version takes a different arguments list □allow to distinguish

```
template < class T>
T findMax( T x, T y ){
T max ;
if(y > x)
        max = y ;
else
        max = x ;
return max;
}
```

```
template < class T>
T findMax( T x, T y, T z ){
T max = x ;
if(y > max)
        max = y ;
if(z > max )
        max = z ;
return max;
}
```

More than one type

```
template < class T, class U >
void compare( T v1, U v2){
    if(v1 > v2 )
        cout<< v1 << " >" << v2  << endl;
    else
        cout<< v1 <<" <= " << v2  << endl;
    ....
}
```

- You should be aware of comparison of different type, the datatype created by programmer should have its overloading function

The usefulness of class template(1/2)

- If we want to create a class, which can store some data, but we are not sure what type it is until the program compiles .

```
Template <class T >
class data{
    T m_data ;
    public:
        data(T val):m_data(val) {};
}
```

```
class data{
    int m_data ;
    public:
        data(int val):m_data(val) {};
};
```

```
class data{
    double m_data ;
    public:
        data(double val): m_data(val)   {};
};
```

```
class data{
    char m_data ;
    public:
        data(char val): m_data(val)      {};
};
```

The usefulness of class template(2/2)

- A class template define a family of class :
  - Serve as a class outline to generate many classes
  - Specific classes are generated during compile time
- Class template promote code reusability
  - Reduce program develop time
  - In a class template , at least one argument is genetic

Creating class template

```
template <class T >
class data{
     T m_data ;
     public:
      data(T val):m_data(val)
{};
      void ShowData(){

cout<<m_data<<endl;
      }
}
```

```
int main(){
    data<int> d1(5) ;
    data<char> d2('D') ;

    data d3(5) ;     compile error,
without argument
    data<int> d4 ;      compile error,
no match constructor

    d1.ShowData();      d2.ShowData();
}
```

Template parameters(1/5)

- 3 forms of template parameters
  - type parameter
  - non-type parameter
  - template parameter

Template parameters(2/5)

- Type parameter

```
template <class T, class U, class P>
class c1{
      ………..
};
```

```
int main(){
c1<int,int,double>    x1; //T=int,   U=int,        P=double
c1<char,double,int>x2; //T=char, U=double, P=int

c1<int,int>x3; compile error, wrong number of argument
…….
}
```

Template parameters(3/5)

- Legal non-type parameter
  - integral types: int, char, bool
  - enumeration type
  - reference to object or function
  - pointer to object, function, member
- illegal non-type parameter
  - float, double
  - user-define class type
  - type void

```
template <data d1, double d2, float f>
class c1{
        ..........
};
```

illegal

Template parameters(4/5)

- non-type parameter

```
template <class T = int, int n = 10>
class c1{
        ……….
};
```

good

Note:

without default value of template argument, fill the argument completely when generating a template class .

```
Int main(){
C1< >           x1 ; OK
C1< , >         x2 ;  error, arg1、arg2 invalid
C1< ,30>        x3 ; error, arg1 invalid
C1<double>          x4 ;     OK, T = double, n = 10
C1<double,30>   x5 ;  OK
```

Template parameters(5/5)

- Template parameter

```
template<class A = int , int n = 10 >
class data{
        A m_data;
        public: //…..
                };
```

```
Int main(){

   data<int,10> d1;
   data<int,30> d2;
   data<double>d3;
   oop<data> x1(d1); OK
   oop<data> x2(d2); error, n=10 not match
   oop<data> x3(d3); error, int not match
   ………

}
```

```
template<   template<class A = int, int n = 10 > class V >
class oop{
        V<  > m_data ;
        public:
                oop(V<  > data):m_data(data){};
};
```

Inheritance in templates

```
template <class T>
class basic{
      T x ;
      public:
        basic(T val): x(val){};
        void ShowX(){cout <<
x;}
};
```

```
class derive1: public basic<double>{
      double y ;
      public:
          derive1(double a, double b):
                   basic<double>(a), y(b){}
          void ShowY(){cout<< y;}
};
```

Normal class

```
template <class T>
class derive2: public basic<T>{
      T y ;
      public:
          derive2(T a, T b):basic<T>(a), y(b){}
          void ShowY(){cout<< y;}
};
```

Class template

Inheritance in templates

```
int main(){
        basic<int>       obj1(100);
        derive1          obj2(100.1,100.1) ;
        derive2<char> obj3('A','B');

        obj1.ShowX(); cout<<endl;
        obj2.ShowX(); cout<<"   "; obj2.ShowY();
cout<<endl;
        obj3.ShowX(); cout<<"   "; obj3.ShowY();
cout<<endl;
…
}
```

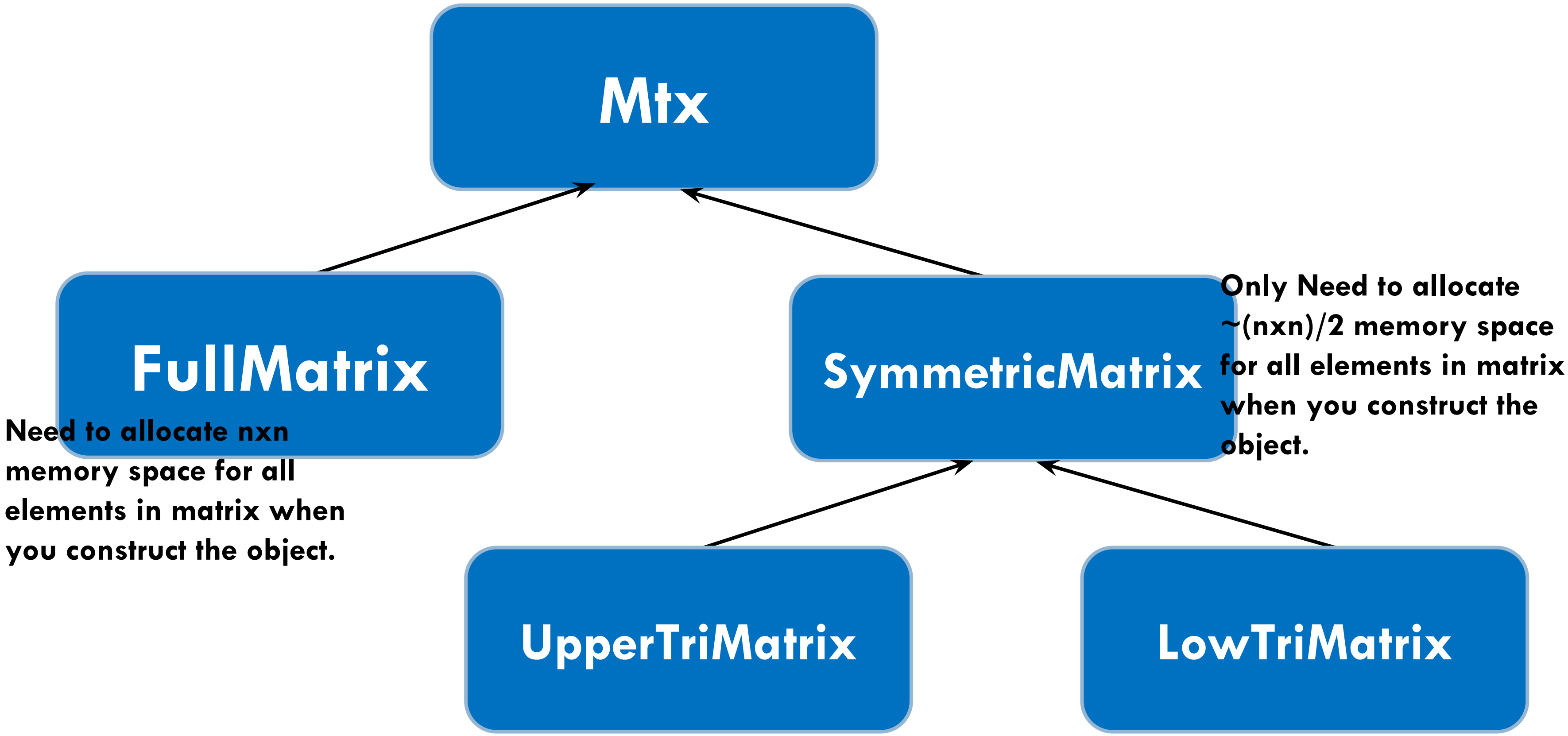
```
100
100.1    100.1
A    B
```

Exercise : Matrix

- Practice the concept of Polymorphism and Template in OOP.
- **Goal : Finish this program and show correct results.**
  (1) `Mtx` : is an abstract class which defines the interface of derived class.
  (2) `FullMatrix` and `SymmetricMatrix` : are two derived classes and inherit from abstract class `Mtx`.
  (3) `UpperTriMatrix` and `LowTriMatrix` : inherit from class `SymmetricMatrix`.
- Notes:
  - main() and simple structure of classes are provided to help you for developing your program.
  - Please mainly focus on finishing the classes and using the concept of template to meet the operations in main() function.
  - Based on structures of reference code, you can add any members if necessary. Also, you should modify code to meet the requirements of using template.

Inheritance Hierarchy

- The inheritance hierarchy :



Main()


```cpp
//-----------------------------------------------------------
int main(){
    FullMatrix<int> A(2);
    A(0,0) = 5; A(0,1) = 4; A(1,0) = 3; A(1,1) = 6;

    SymmetricMatrix<int> B(3);
    B(0,0) = 5; B(1,0) = 3; B(1,1) = 6;
    B(2,0) = 1; B(2,1) = 9; B(2,2) = 2;

    UpperTriMatrix<int> C(3);
    C(0,0) = 1; C(0,1) = 2; C(0,2) = 4;
    C(1,1) = 3; C(1,2) = 5; C(2,2) = 6;

    LowTriMatrix<double> D(3);
    D(0,0) = 9.1; D(1,0) = 8.3; D(2,0) = 7.1;
    D(1,1) = 6.2; D(2,1) = 5.5; D(2,2) = 4.2;

    UpperTriMatrix<double> E(3);
    E(0,0) = 1.3; E(1,0) = 2.2; E(2,0) = 4.7;
    E(1,1) = 3.5; E(2,1) = 5.9; E(2,2) = 6.1;

    LowTriMatrix<double> F(3);
    F(0,0) = 9.7; F(0,1) = 8.1; F(0,2) = 7.1;
    F(1,1) = 6.5; F(1,2) = 5.5; F(2,2) = 4.3;


    Mtx<int> *vecA[3];
    vecA[0] = &A; vecA[1] = &B; vecA[2] = &C;
    for (int i = 0; i < 3; ++i) {
        vecA[i]->showMatrix(); cout << endl;
    }

    Mtx<double> *vecB[3];
    vecB[0] = &D; vecB[1] = &E; vecB[2] = &F;
    for (int i = 0; i < 3; ++i) {
        vecB[i]->showMatrix(); cout << endl;
    }

    return 0;
}
```

Output

- Meet the output requirement and you can pass the Lab.

```
16:06 sychen@vda04 [~/Desktop/OOP_TA] >$ ./lab9_demo.exe
5       4
3       6

5       3       1
3       6       9
1       9       2

1       2       4
0       3       5
0       0       6

9.1     0       0
8.3     6.2     0
7.1     5.5     4.2

1.3     0       0
0       3.5     0
0       0       6.1

9.7     0       0
0       6.5     0
0       0       4.3
```

Submission

- Try your best to develop the classes and templates.
- Try your best to debug your code by yourself
- Ask TA for demo.
- Naming rule : studentID_lab9.cpp