Advanced Programming

Lab 15, Friend classes, Nested classes and RTTI

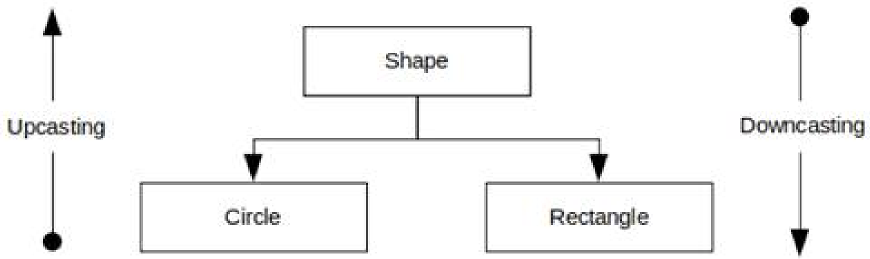
廖琪梅，王大兴，于仕琪，王薇

Friend classes, Nested classes and RTTI

- Friend class
  - declaration and privilege
  - Friendship ***is not symmetric***
  - Friendship ***is not transitive***

- Nested class
  - declaration in **public** section vs **private** vs **protect**

- RTTI(Run-Time Type Identification)
  - Upcasting vs Downcasting
  - implicit casting vs **explicit** casting
  - explicit casting
    - staic vs **dynamic**(suggested)
  - typeid function



Friend Classes

Entire classes or member functions of other classes may be declared to be friends of another class.

To declare all member functions of **ClassTwo** as friend of **ClassOne**,

place a declaration of the form **friend class ClassTwo;** in the definition of **ClassOne**.

That means all member functions of **ClassTwo** have the right to access the private and protected class members of **ClassOne**.

The friend declaration(s) can appear anywhere in a class and are not affected by access specifiers public or private or protected.

Let's consider an example: we have a **Circle** class in which it has a subobject (center point) of **Point** class--class containment(composition).

Can we access the center's **private member(x and y)** in the **Point** class?

```cpp
class Circle
{
private:
    Point center;
    double radius;

public:
    Circle():center(0,0),radius(1.0) { }
    Circle(Point &p, double r):center(p),radius(r) { }

    Circle& move(Point& p)
    {
        center.x = p.getX();
        center.y = p.getY();

        return *this;
    }

    void show() const
    {
        center.show();
        cout << "radius:" << radius << endl;
    }
};
```

class containment(or class composition)

In **move** function we want to set the center to the new point p.
But you **CANNOT** access the **center's private members x and y**.

```
demo1.cpp: In member function 'Circle& Circle::move(Point&)':
demo1.cpp:34:16: error: 'double Point::x' is private within this context
   34 |         center.x = p.getX();
      |                ^
```

You can access the public members of the center.

```cpp
class Point
{

private:
    double x;
    double y;

public:
    Point(double xx = 0, double yy = 0)
    {
        x = xx;
        y = yy;
    }

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

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

    void show() const
    {
        cout << x << "," << y << endl;
    }
};
```

This time you can declare the **Circle** class as a **friend class** of the **Point** class.

```
class Circle;
```

This declaration is necessary which is called **forward declaration**.

```
class Point
{
    friend class Circle;

private:
    double x;
    double y;

public:
    Point(double xx = 0, double yy = 0)
    {
        x = xx;
        y = yy;
    }

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

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

    void show() const
    {
        cout << x << "," << y << endl;
    }
};
```

Declare the **Circle** class as a friend of the **Point** class. That means in **Circle** class, its member functions can access the private members of the **Point** class.

```
class Circle
{
private:
    Point center;
    double radius;

public:
    Circle():center(0,0),radius(1.0) { }
    Circle(Point &p, double r):center(p),radius(r) { }

    Circle& move(Point& p)
    {
        center.x = p.x;
        center.y = p.y;

        return *this;
    }

    void show() const
    {
        center.show();
        cout << "radius:" << radius << endl;
    }
};
```

Member function of the **Circle** class can access the private member of the **Point** class.

Declare the **Circle** class as a friend of the **Point** class. That means in **Circle** class, its member functions(e.g. move) can access the private members of the **Point** class.

```cpp
class Point
{
    friend class Circle;

private:
    double x;
    double y;

public:
    Point(double xx = 0, double yy = 0)
    {
        x = xx;
        y = yy;
    }

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

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

    void show() const
    {
        cout << x << "," << y << endl;
    }
};
```

```cpp
class Circle
{
private:
    Point center;
    double radius;

public:
    Circle():center(0,0),radius(1.0) { }
    Circle(Point &p, double r):center(p),radius(r) { }

    Circle& move(Point& p)
    {
        center.x = p.x;
        center.y = p.y;

        return *this;
    }

    void show()  const
    {
        center.show();
        cout << "radius:" << radius << endl;
    }
};
```

```cpp
int main()
{
    Point p1(1,1),p2(4,5);
    Circle c1;
    Circle c2(p1, 12);

    cout << "Before move:" << endl;
    c1.show();
    c2.show();

    cout << "After move:" << endl;
    c1.move(p1);
    c2.move(p2);
    c1.show();
    c2.show();

    return 0;
}
```

```
Before move:
0, 0
radius: 1
1, 1
radius: 12
After move:
1, 1
radius: 1
4, 5
radius: 12
```

Notes:

- Friendship ***is not symmetric***– if class A is a friend of class B, you CANNOT infer that class B is a friend of class A.

- Friendship ***is not transitive*** ---if class A is a friend of class B and class B is a friend of class C, you CANNOT infer that class A is a friend of class C.

When to use friend class?

If one class(or object) is not another class(or object) and vice versa, so the ***is-a relationship*** of public inheritance doesn't apply. Nor it is either a component of the other, so the ***has-a relationship*** of containment or of private or protected inheritance doesn't apply. This suggests making the one class **a friend** to the other class.

Nested Class

A **nested class** is a class which is declared in another **enclosing class**.

- Nested class can be defined in **private** as well as in **public** and in **protect** section of the enclosing class.
- A nested class is a member and as such has the same access rights as any other member.
- The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.

Nested class defined in public section of the enclosing class

```
#include<iostream>
using namespace std;
class Outer{
public:
    class  Inner{
    public:
        void Fun();
    };

    Inner obj_;

    void Fun(){
        cout<<"Outer::Fun..."<<endl;
        obj_.Fun();
    }
};

void Outer::Inner::Fun(){
    cout<<"Inner::Fun..."<<endl;
}

int main(){
    Outer o;
    o.Fun();
    Outer::Inner i;
    i.Fun();

    return0;
}
```

Declare a nested class **Inner** in the **public** section of **Outer** class

Define the function of the nested class **Inner**, using a **class qualifier**

Define an object of the **Inner** class using a **class qualifier**

```
Outer::Fun...
Inner::Fun...
Inner::Fun...
```

If a nested class(in the example **Inner** class) is declared in a ***public section*** of an enclosing class(in the example **Outer** class)

- **it is available to the enclosing class, to classes derived from the enclosing class**, and, because it's public, **to the outside world**.
- However, because the nested class has class scope, it has to be used with a **class qualifier**(in the example **Outer::**) in the outside world.

Nested class defined in private section of the enclosing class

```cpp
#include<iostream>
using namespace std;
class Outer{
private:
    class  Inner{
    public:
        void Fun(){
            cout<<"Inner::Fun..."<<endl;
        }
    };
    Inner obj_;
public:
    void Fun(){
        cout<<"Outer::Fun..."<<endl;
        obj_.Fun();
    }
};

class OuterDeprive: Outer{
public:
    Inner objDp;
};

int main(){
    Outer o;
    o.Fun();
    Outer::Inner i;
    i.Fun();
    return0;
}
```

Declare a nested class **Inner** in the **private** section of **Outer** class

If a nested class is declared in a ***private section*** of an enclosing class, it is **known only to that enclosing class**. e.g. Inner class nested in a ***private section*** of Outer declaration.

- Outer members can use inner objects.
- Other parts of a program don't even know that the Inner class exists.
- While **derive a class from Outer, inner would be invisible to that derived class(OuterDeprive)**, because a derived class can't directly access the private parts of a base class.

```
nested_demo2.cpp:22:5: error: 'class Outer::Inner' is private within this context
   22 |     Inner objDp;
      |     ^~~~~
nested_demo2.cpp:6:12: note: declared private here
    6 |     class  Inner{
      |            ^~~~~
nested_demo2.cpp: In function 'int main()':
nested_demo2.cpp:28:12: error: 'class Outer::Inner' is private within this context
   28 |     Outer::Inner i;
      |            ^~~~~
nested_demo2.cpp:6:12: note: declared private here
    6 |     class  Inner{
      |            ^~~~~
```

Nested class defined in protected section of the enclosing class

```cpp
#include<iostream>
using namespace std;
class Outer{
protected:
    class  Inner{
    public:
        void Fun(){
            cout<<"Inner::Fun..."<<endl;
        }
    };
    Inner obj_;
public:
    void Fun(){
        cout<<"Outer::Fun..."<<endl;
        obj_.Fun();
    }
};

class OuterDeprive: Outer{
public:
    Inner objDp;
};

int main(){
    Outer o;
    o.Fun();
    Outer::Inner i;
    i.Fun();
    return0;
}
```

Declare a nested class **Inner** in the **protected** section of **Outer** class

If the nested class is declared in a ***protected section*** of a enclosing class, e.g. **Inner** class nested in a ***protected section*** of **Outer** declaration.

- it is **visible to that class(Outer) but invisible to the outside world**.
- **a derived class(OuterDeprive) would know about the nested class** and could directly create objects of that type.

```
nested_demo3.cpp: In function 'int main()':
nested_demo3.cpp:28:12: error: 'class Outer::Inner' is protected within this context
   28 |     Outer::Inner i;
      |            ^~~~~
nested_demo3.cpp:6:12: note: declared protected here
    6 |     class  Inner{
      |            ^~~~~
```

RTTI(Run-Time Type Identification)

Type conversion A type cast is basically a conversion from one type to another. C++ supports four types of casting: **Static** Cast, **Dynamic** Cast, **Const** Cast, **Reinterpret** Cast.

```
static_cast<type> (expression)
```

```
int firstNumber, secondNumber;
double result = ((double)firstNumber)/secondNumber;
double result = static_cast<double> (firstNumber)/secondNumber;
```

C style cast

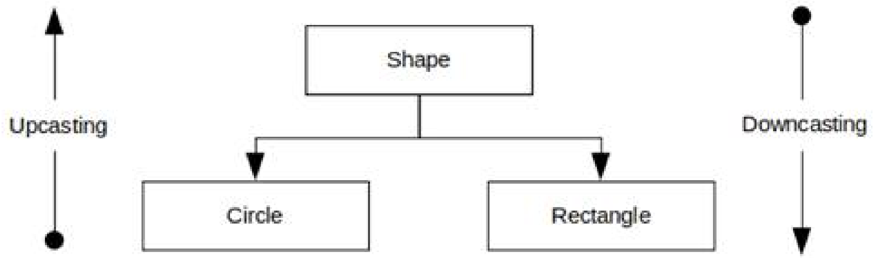
C++ style cast

static_cast has basically the same power and meaning as the general-purpose C-style cast. It also has the same kind of restrictions. For example, you **CANNOT** cast a **struct** into an **int** or a **double** into a pointer. Furthermore, **static_cast** can't remove constness from an expression.

RTTI (RunTime Type Identification).

- It is a mechanism to find the type of an object dynamically(**in runtime**) from an available **pointer** or **reference** to the base type.
- It provides an **explicit** way to identify the runtime type separately from what is possible with the **virtual function mechanism**.

The casting of an object is mainly required when dealing with the inheritance hierarchy of classes.



- **Up casting** is the process where we treat a pointer or a reference of a derived class object as a base class pointer. It is automatically accomplished by assigning a derived class pointer or a reference to its base class pointer.

```
Shape *shape_ptr = nullptr;
Rectangle rec(10, 20);
shape_ptr = &rec;     //upcasting, need not explicitly cast
```

- **Down casting** is converting a base class pointer or reference to a derived class. It requires explicit cast.

```
Shape *shape_ptr = nullptr;
Rectangle rec(10, 20);
shape_ptr = &rec;
Rectangle *rec_ptr = nullptr;
rec_ptr = (Rectangle *) shape_ptr;
rec_ptr = static_cast<Rectangle *> (shape_ptr);
```

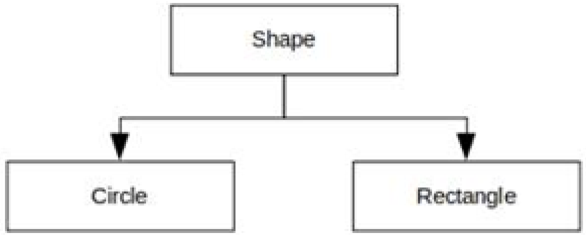
downcasting, need expicitly cast

Using C-style cast or C++ static_cast, neither conversion is safety

dynamic_cast is used to perform safe casts down or across an inheritance hierarchy. That is, you use **dynamic_cast** to **cast** pointers or references to **base** class objects **into** pointers or references to **derived** or **sibling base class objects**. **dynamic_cast < type-name > (expression)**

In all cases, the type of ***expression*** must be either **a class type that is publicly derived from the type-name type**, a public base class of the type-name which points to the type-name, or **the same as the type-name**.

- **If expression has one of these types, then the cast will succeed. Otherwise, the cast fails.**
- If a ***dynamic_cast*** to a pointer type fails, the result is 0. If a ***dynamic_cast*** to a reference type fails, the operator throws an exception of type **bad_cast**.




```
Shape *shape_ptr = nullptr;
Rectangle rec(10, 20);
shape_ptr = &rec;
Rectangle *rec_ptr = nullptr;
rec_ptr = dynamic_cast<Rectangle *> (shape_ptr);
```

```
Rectangle rec(10, 20);
Shape &shape_sr = rec;
try{
    Rectangle &rectangle_rr = dynamic_cast<Rectangle &> (shape_sr);
}catch(std::bad_cast &bc){
    cerr << bc.what() << endl;
}
```

When failing to cast a reference, dynamic_cast throws ***std::bad_cast*** exception defined in the ***typeinfo*** header.

Note: ***dynamic_cast***s CANNOT be applied to types lacking virtual functions, nor can they cast away constness.

NOTE: There MUST be at least one virtual function in the class B, otherwise it fails to compile.

```
class B { ... };
class D : public B { ... };
void f()
{
   B* pb = new D;
   B* pb2 = new B;
   D* pd = dynamic_cast<D*>(pb);      // ok: pb points to D
   ...
   D* pd2 = dynamic_cast<D*>(pb2);   // fail,pb2 points to B not D
                                     //  pd2 is NULL
   ...
}
```

It is suggested to check whether the downcast is successful by "if" statement.

```
if (Derived *dp = dynamic_cast<Derived*>(bp))   //bp is a base class pointer
{
}
```

If **bp** points to a Derived object, then the cast will initialize dp to point to the Derived object to which bp points. In this case, it is safe for the code inside the if to use Derived operations. Otherwise, the result of the cast is 0.

```cpp
#include <iostream>

#ifndef _CAST_H_
#define _CAST_H_

class Base
{
public:
    Base(){}
    virtual ~Base(){}

    void show()
    {std::cout << "Base funtion" << std::endl;}
};

class Inherit :public Base
{
public:
    Inherit(){}
    ~Inherit(){}

    void show()
    { std::cout << "Inherit funtion" << std::endl;}
};

#endif
```

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

int main()
{
    Base* pBase = new Inherit();

    pBase->show();

    delete pBase;

    return 0;
}
```

Invoke the show() of base class, though pBase points to the derived object.

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

int main()
{
    Base* pBase = new Inherit();

    if(Inherit* pInherit = dynamic_cast<Inherit*>(pBase))
    {
        pInherit->show();
    }

    delete pBase;

    return 0;
}
```

Invoke the show() of derived class, because pBase is converted to the derived pointer.

typeid operator

typeid operator can tell you what type is the object.

typeid(expression)

The operand can be any expression or type name.

- The ***typeid*** operator returns a reference to a **type_info** object, where type_info is a class defined in the typeinfo header file.
- The type_info class overloads the **==** and **!=** operators so that you **can use these operators to compare types**.
- **If** the **expression's type is a class and contains at least one virtual function**, the ***typeid*** operator returns the **dynamic type of the expression**; **otherwise**, it provides **static type** information.

Suppose there is at least one virtual function in the class B.

```
class Base { ... };
class Derived : public Base { ... };
Derived *dp = new Derived;
Base *bp = dp;

// compare the type of two objects at run time
if (typeid(*bp) == typeid(*dp))
 {  …                              }
```

the operands of the typeid are objects, so use *dp not dp

```
 // test whether the run-time type is a specific type
 if (typeid(*bp) == typeid(Derived))
{ …                                                    }
```

type_info class includes a ***name()*** member that returns an string that is typically the name of the class.

```
#include <iostream>
#include <typeinfo>

using namespace std;

class Base_A {
public:
    virtual ~Base_A(){}
};

class Derived_A: public Base_A{
public:
    Derived_A(){}
};

class Base_B{

};

class Derived_B: public Base_B{
public:
    Derived_B(){}
};
```

has no virtual function

```
int main()
{
    const Derived_A ref_da;
    const Base_A *pa = &ref_da;
    cout<<"typeid(pa) is: " << typeid (pa).name()<< ", "
        <<"typeid(*pa) is: " << typeid (*pa).name()<<endl;

    cout<<"typeid(*pa) == typeid(ref_da)? "
        << (typeid (*pa)==typeid (ref_da) ? "true":"false")<<endl;
    cout <<"typeid(Devrived_A) == typeid(const Derived_A)? "
        <<(typeid(Derived_A) == typeid(const Derived_A)? "true":"false")<< endl;

    const Derived_B ref_db;
    const Base_B *pb = &ref_db;

    cout <<"typeid(pb) is: " << typeid (pb).name() << ", "
         << "typeid(*pd) is: " << typeid (*pb).name()<<endl;

    cout << "typeid(*pb) == typeid(ref_db)? "
         <<(typeid (*pb)==typeid (ref_db) ? "true":"false")<<endl;
    cout << "typeid(Derived_B) == typeid(ref_db)? "
         <<(typeid(Derived_B) == typeid(const Derived_B) ? "true":"false")<< endl;

    return 0;
}
```

```
typeid(pa) is: PK6Base_A, typeid(*pa) is: 9Derived_A
typeid(*pa) == typeid(ref_da)? true
typeid(Devrived_A) == typeid(const Derived_A)? true
typeid(pb) is: PK6Base_B, typeid(*pd) is: 6Base_B
typeid(*pb) == typeid(ref_db)? false
typeid(Derived_B) == typeid(ref_db)? true
```

P:pointer, **K**: const,

6: numbers of the characters of the type

Exercise 1

There are two classes named **Car** and **Driver**, suppose the car can drive automatically, and driver also can drive the car. The declarations of car and driver are as follows:

```
class Car {
private:
    enum {Off, On};  //Off- non automatically drive, On-automatically drive
    enum {Minvel, Maxvel = 200}; //range of velocity from 0  to 200
    int mode;     //mode of car, Off or On
    int velocity;

public:
    friend class Driver;
    Car(int m = On, int v = 50):mode(m),velocity(v){ }
    bool velup(int v);       //increase  velocity by v
    bool veldown(int v);    //decrease velocity by v
    bool ison() const;     //Check whether the mode is on
    int getvel() const;    //get the velocity
    void showinfo() const; // show the mode and velocity of car
};
```

```
class Driver {
public:

    bool velup(Car& car,int v);      //increase velocity by v
    bool veldown(Car& car, int v);    //decrease velocity by v
    void setmode(Car& car);       //If the mode is On, set it to Off,otherwise set it to Off
    bool ison(Car& car) const;  //Check whether the mode is on
};
```

Implement all the member functions of the two classes and make Driver as Car's friend class so that it can access the members of Car. Write a program to test the two classes.

Output sample:

```
The infomation of car:mode is On,velocity is 50
Increase velocity by car,its mode is On,velocity is 170
Set the mode of car by driver:
The mode of car is:On.
The infomation of car:mode is Off,velocity is 170
Decrease velocity by driver:mode is Off,velocity is 70
Increase velocity by driver: The velocity is 220. It is out of Maxvel.
```

Exercise 2

There are two declarations of two classes, **Base** and Derived. You are required to implement the three kind of functions:

(1) **Equality operator ==** is declared as a friend function of the Base class. Two objects are equal if they have the same type and same value for a given set of their data members. If two objects have different types, throw a message "The two objects have different types, they can not be compared." as an exception.

(2) Virtual **equal member functions** in Base and Derived class which check whether the data members have the same values in its own objects respectively.

(3) **void process(const Base&, const Base&) function** is a normal function who checks if the two objects are equal and handles the exception.

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

class Base
{
protected:
    int bvalue;
public:
    Base(int i ) : bvalue(i) {}

    virtual bool equal(const Base& b) const;

    friend bool operator == (const Base&, const Base&);
};
```

```
class Derived : public Base
{
private:
    int dvalue;
public:
    Derived(int a, int b):Base(a), dvalue(b){}

    virtual bool equal(const Base& b) const override;

};
```

```
int main()
{
    Base b1(2);
    Base b2(2);

    Derived d1(1,2);
    Derived d2(2,2);

    process(b1,b2);
    process(d1,d2);
    process(b1,d1);

    return 0;
}
```

Run the main function to check your defined functions.

```
Two Base type objects are equal.
Two Derived type objects are not equal because they have different values.
The two objects have different types, they can not be compared.
```