

c++第七天课堂笔记

一、回顾知识点

1.1 多态

多态的本质： 父类的引用或指针指向子类的对象

多态的前提条件： 继承+重写父类的虚函数

多态的应用方式：

- 1) 局部使用， 如在一个函数的内部使用
- 2) 函数的参数上体现，
形参是父类的引用或指针， 调用函数时的实参是子类对象

虚函数： 虚函数可以有函数体，子类重写时，可以带virtual关键字

```
virtual 返回值类型 函数名(形参列表){}
```

在类声明虚函数，在类外实现虚函数时，不需要virtual关键字。

1.2 抽象类与纯虚函数

抽象类： 包含一个纯虚函数的类，则为抽象类； 抽象类不能实例化（不能创建对象）

纯虚函数：

```
virtual 返回值类型 函数名(形参列表) = 0;
```

类中除了构造函数之外，其它的函数都可以是纯虚函数。

1.3 虚析构与纯虚析构

虚析构：

析构函数前加 `virtual`，在多态的应用中可以保证子类对象的析构函数正常调用。

纯虚析构：

纯虚析构所在的类，是抽象类， 在类中声明为纯虚析构，必须在类外定义它的函数体。

```
virtual ~类名() = 0;
```

1.4 接口类的多继承

接口类： 类中的所有函数都是纯虚函数（除了构造函数）

多继承： 一个类可以继承多个不同的接口类， 这种多继承不会产生二义性。

1.5 override、overload、redefined

override: 重写

子类重写父类的虚函数

overload: 重载

函数名相同，参数个数、类型、顺序等不同，与函数返回值类型无关

redefined: 重定义

子类重定义父类的非虚函数，重定义会隐藏父类的同名所有的函数。

1.6 函数模板

简单化函数重载的写法，大大地减少代码量。

函数模板，即为参数模板，在设计函数时，将函数的形参类型进行泛化（泛型），在调用函数时，可以指定某一种具体的数据类型。

函数模板的语法：

```
template<typename或class 泛型名>  
返回值类型 函数名(泛型名 形参名, ...) {}
```

在调用带泛型的函数时，简单的数据类型编译可以自动推演出来，也可以手动指定。

函数名<指定泛型的类型>()

一个函数模板中，可以定义多个泛型

```
template<typename 泛型名1, typename 泛型名2, ...>
```

1.7 普通函数与带泛型的函数的区别

普通函数 优先 泛型函数
空泛型函数的调用，则会调用泛型函数

二、深入C++模板

2.1 模板实现机制

函数模板机制结论：编译器并不是把函数模板处理成能够处理的任何类型的函数

函数模板通过具体类型产生不同的函数
编译器会对函数模板进行两次编译，在声明的地方对模板代码本身进行编译，在调用的地方对参数替换后的代码进行编译

2.2 模板的局限性

函数模板是存在的一些局限性的，如果函数模板声明的泛型，在实际使用中，具体化为基础数据类型则完全可以使用，但是具体化是一个类对象时或地址时，则需要特殊的处理。

如: 问题（两个类实例无法比较的）

```
template<typename T>
T &maxVal(T &a, T &b){
    return a>b?a: b;
}

class A{
private:
    int x;
public:
    A(int x): x(x){}
    void show(){
        cout << "x=" << x << endl;
    }
};

int main(){
    A a1(20), a2(50);
    A &b = maxVal(a1, a2);
    b.show();
    return 0;
}
```

2.2.1 具体化优先于常规模板

为maxVal函数模板定义一个具体泛型的函数模板

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

template <typename T>
T &maxVal(T &a, T &b)
{
    return a > b ? a : b;
}

class A
{
public:
    int x;

public:
    A(int x) : x(x) {}
    void show()
    {
        cout << "x=" << x << endl;
    }
};

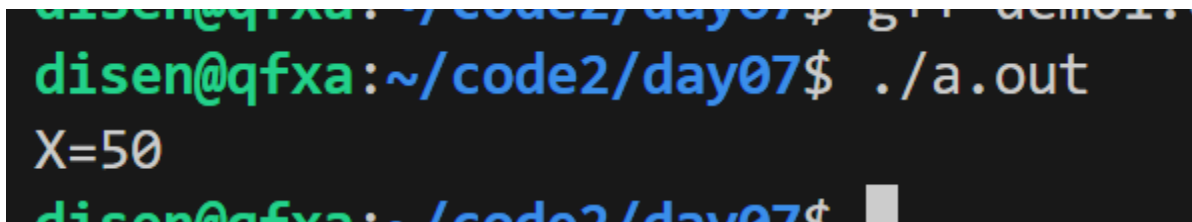
// 具体化函数模板的重载
template <>
```

```

A &maxVal<A>(A &a, A &b)
{
    if (a.x > b.x)
        return a;
    return b;
}

int main()
{
    A a1(20), a2(50);
    // A &b = maxVal<A>(a1, a2);
    A &b = maxVal(a1, a2); // 优先调用具体化的函数模板
    b.show();
    return 0;
}

```



```

disen@qfxa:~/code2/day07$ ./a.out
X=50
disen@qfxa:~/code2/day07$

```

2.2.2 具体化函数模板为友元函数

可以访问类对象的私有成员

```

#include <iostream>
using namespace std;

template <typename T>
T &maxVal(T &a, T &b)
{
    return a > b ? a : b;
}

class A
{
    friend A &maxVal<A>(A &a, A &b);

private:
    int x;

public:
    A(int x) : x(x) {}
    void show()
    {
        cout << "X=" << x << endl;
    }
};

// 具体化函数模板的重载
template <>
A &maxVal<A>(A &a, A &b)
{
    if (a.x > b.x)
        return a;
}

```

```

        return b;
    }

    int main()
    {
        A a1(20), a2(50);
        // A &b = maxVal<A>(a1, a2);
        A &b = maxVal(a1, a2); // 优先调用具体化的函数模板
        b.show();
        return 0;
    }

```

2.2.3 类中定义运算符重载

```

#include <iostream>
using namespace std;

template <typename T>
T &maxVal(T &a, T &b)
{
    return a > b ? a : b;
}

class A
{
private:
    int x;

public:
    A(int x) : x(x) {}
    void show()
    {
        cout << "X=" << x << endl;
    }
    // 大于运算符重载，解决函数模板的局限性（a>b 比较）
    bool operator>(A &other)
    {
        return this->x > other.x;
    }
};

int main()
{
    A a1(20), a2(50);
    A &b = maxVal(a1, a2);
    b.show();
    return 0;
}

```

```

disen@qfxa:~/code2/day07$ ./a.out
X=50

```

2.3 类模板

类模板针是类成员变量的数据类型泛化。

类模板的声明作用于整个类的，即类的内部的任何位置都可以使用。

2.3.1 初次使用

如：定义长方形的图形类

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

template <typename T>
class Rect
{
private:
    T width, height;

public:
    Rect(T w, T h) : width(w), height(h) {}
    void draw()
    {
        cout << "绘制方形 width=" << width << ",height=" << height << endl;
    }
    T length()
    {
        return (width + height) * 2;
    }
};

int main()
{
    Rect<int> r1(20, 30);
    Rect<float> r2(1.5, 2.25);
    r1.draw();
    cout << "r1的周长:" << r1.length() << endl;

    r2.draw();
    cout << "r2的周长:" << r2.length() << endl;

    return 0;
}
```

```
disen@qfxa:~/code2/day07$ ./a.out
绘制方形 width=20,height=30
r1的周长:100
绘制方形 width=1.5,height=2.25
r2的周长:7.5
disen@qfxa:~/code2/day07$
```

2.3.2 类模板作为函数参数

类模板作为函数参数时，必须指定泛型的具体类型。

当然，类模板的泛型也可以是函数模板的泛型。

如1：类模板作为函数参数时，必须指定泛型的具体类型

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

template <typename T>
class Rect
{
private:
    T width, height;

public:
    Rect(T w, T h) : width(w), height(h) {}
    void draw()
    {
        cout << "绘制方形 width=" << width << ",height=" << height << endl;
    }
    T length()
    {
        return (width + height) * 2;
    }
};

// 类模板作为函数的参数， 具体化类模板
void drawShape(Rect<int> &r)
{
    r.draw();
    cout << "r周长: " << r.length() << endl;
}

void drawShape(Rect<float> &r)
{
    r.draw();
    cout << "r周长: " << r.length() << endl;
}

int main()
{
    Rect<int> r1(20, 30);
    Rect<float> r2(1.5, 2.25);

    drawShape(r1);
    drawShape(r2);

    return 0;
}
```

```
disen@qfxa:~/code2/day07$ ./a.out
绘制方形 width=20,height=30
r周长: 100
绘制方形 width=1.5,height=2.25
r周长: 7.5
```

如2: 类模板的泛型也可以是函数模板的泛型

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

template <typename T>
class Rect
{
private:
    T width, height;

public:
    Rect(T w, T h) : width(w), height(h) {}
    void draw()
    {
        cout << "绘制方形 width=" << width << ",height=" << height << endl;
    }
    T length()
    {
        return (width + height) * 2;
    }
};

// 类模板作为函数的参数, 具体化类模板
// 将类模板的泛型再次泛型, 通过函数模板
template <typename T>
void drawShape(Rect<T> &r)
{
    r.draw();
    cout << "r周长: " << r.length() << endl;
}

int main()
{
    Rect<int> r1(20, 30);
    Rect<float> r2(1.5, 2.25);

    drawShape(r1);
    drawShape(r2);

    return 0;
}
```


2.3.3 类模板派生普通类

类模板派生普通类的时候时，必须指定具体的泛型的数据类型，以确保子类对象创建时的具体的父类。

如：设计Shape类，方形Rect类， 计算不同的图形的周长

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

template<typename T>
class Shape{
protected:
    T mLen;
public:
    virtual T length(){
        return mLen;
    }
};

class Rect: public Shape<int>{
private:
    int w,h;
public:
    Rect(int w, int h): w(w),h(h){
        mLen = 2*(w+h);
    }
};

class Triangle: public Shape<float>{
private:
    float a, b,c;
public:
    Triangle(float a, float b, float c){
        this->a = a;
        this->b = b;
        this->c = c;
        mLen = a+b+c;
    }
};

int main(){
    Rect r1(10, 20);
    Triangle t1(5.4, 5.52, 8.15);
    cout << "Rect的周长: " << r1.length() << endl;
    cout << "Triangle的周长: " << t1.length() << endl;

    return 0;
}
```

```
disen@qfxa:~/code2/day07$ ./a.out
Rect的周长: 60
Triangle的周长: 19.07
```

2.3.4 类模板派生类模板

类模板派生子类时，子类也可以模板，将父类中的泛型指定为子类的类模板泛型。

【注意】派生的子类模板的类的内部不能访问父类的泛型成员，并且子类模板创建对象时，必须指定泛型的具体的类型。

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

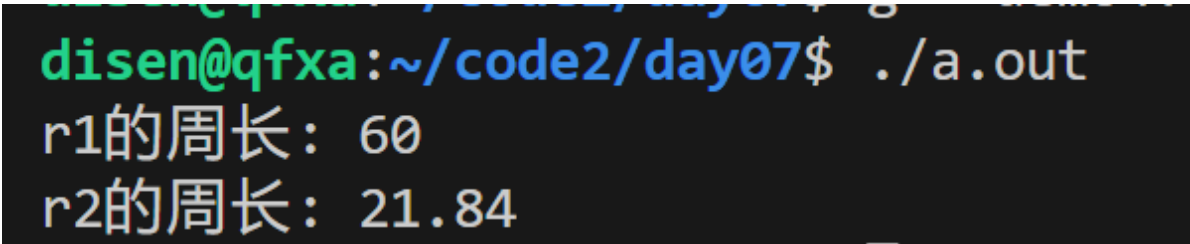
template <typename T>
class Shape
{
public:
    virtual T length() = 0;
};

template <typename T>
class Rect : public Shape<T>
{
private:
    T w, h;

public:
    Rect(T w, T h) : w(w), h(h)
    {
    }
    virtual T length()
    {
        return 2 * (w + h);
    }
};

int main()
{
    Rect<int> r1(10, 20);
    Rect<float> r2(5.4, 5.52);
    cout << "r1的周长: " << r1.length() << endl;
    cout << "r2的周长: " << r2.length() << endl;

    return 0;
}
```



```
disen@qfxa:~/code2/day07$ ./a.out
r1的周长: 60
r2的周长: 21.84
```

2.3.5 类模板类内实现

在类模板内部的成员函数中，可以使用泛型成员变量。

如：

```
#include <iostream>
```

```

#include <cmath>
using namespace std;

template <typename T1, typename T2>
class Point
{
private:
    T1 x;
    T2 y;

public:
    Point(T1 x, T2 y) : x(x), y(y) {}
    // ? T1, T2是哪一个对象的泛型的具体化： 是当前类对象的泛型
    // 此函数要求: other对象的泛型同当前类对象的泛型保持一致, 否则编译器认为一个其它的类
    int distancePow(Point<T1, T2> &other)
    {
        return (x - other.x) * (x - other.x) + (y - other.y) * (y - other.y);
    }
};

int main()
{
    Point<int, int> p1(2, 3);
    Point<int, int> p2(3, 4);

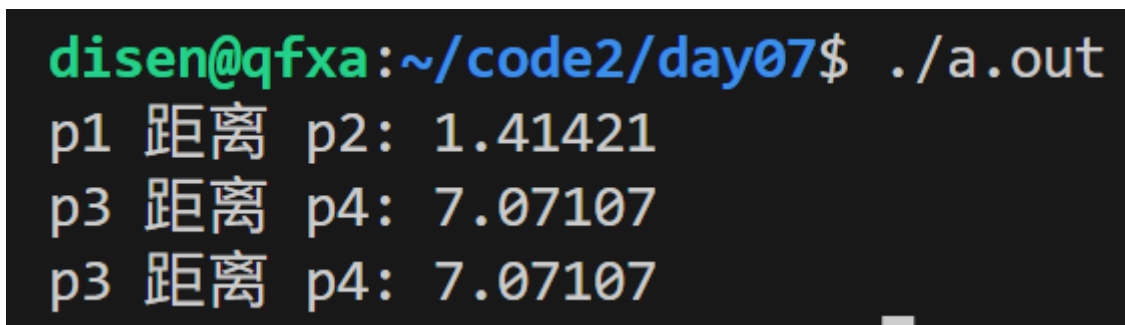
    cout << "p1 距离 p2: " << sqrt(p1.distancePow(p2)) << endl;

    Point<float, float> p3(2.5, 3.5);
    Point<float, float> p4(9.5, 4.5);
    cout << "p3 距离 p4: " << sqrt(p3.distancePow(p4)) << endl;

    Point<int, float> p5(2, 3.5);
    Point<int, float> p6(9, 4.5);
    cout << "p3 距离 p4: " << sqrt(p5.distancePow(p6)) << endl;

    return 0;
}

```



```

disen@qfxa:~/code2/day07$ ./a.out
p1 距离 p2: 1.41421
p3 距离 p4: 7.07107
p3 距离 p4: 7.07107

```

2.3.6 类模板类外实现

类外部实现成员函数时, 指定类模板的泛型转化为函数模板。

如:

```

#include <iostream>
using namespace std;

template<typename T>

```

```

class Point{
private:
    T x, y;
public:
    Point(T x, T y);
    void show();
};

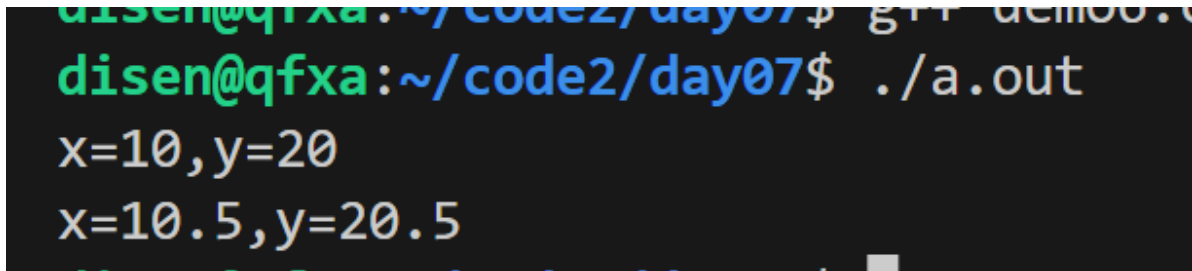
template<typename T>
Point<T>::Point(T x, T y){
    this->x = x;
    this->y = y;
}

template<typename T>
void Point<T>::show(){
    cout << "x=" << x << ",y=" << y << endl;
}

int main(){
    Point<int> p1(10, 20);
    p1.show();

    Point<float> p2(10.5, 20.5);
    p2.show();
    return 0;
}

```



```

disen@qfxa:~/code2/day07$ ./a.out
x=10,y=20
x=10.5,y=20.5

```

2.3.7 类模板头文件与源文件分离的问题

在使用类模板的情况下，声明与实现必须放在一起。

在 Linux 和 vs 编辑器下如果只包含头文件，那么会报错链接错误，需要包含 cpp 文件，但是如果类模板中有友元类，那么编译失败！

如：mypoint.h

```

#ifndef __MYPOINT_H__
#define __MYPOINT_H__
#include <iostream>
using namespace std;

template <typename T>
class Point
{
private:
    T x, y;

```

```

public:
    Point(T x, T y);
    void show();
};
template <typename T>
Point<T>::Point(T x, T y)
{
    this->x = x;
    this->y = y;
}

template <typename T>
void Point<T>::show()
{
    cout << "x=" << x << ",y=" << y << endl;
}

#endif

```

main.cpp

```

#include "mypoint.h"

int main()
{
    Point<int> p1(10, 20);
    p1.show();
    Point<float> p2(10.5, 20.5);
    p2.show();
    return 0;
}

```

```

disen@qfxa:~/code2/day07/demo7$ g++ main.cpp
disen@qfxa:~/code2/day07/demo7$ ./a.out
x=10,y=20
x=10.5,y=20.5

```

【原因】类模板需要二次编译，在出现模板的地方编译一次，在调用模板的地方再次编译。
C++编译规则为独立编译。

2.3.8 模板类碰到友元函数

模板类中的友元函数的写法：1) 内部实现 2) 外部实现 3) 友元函数模板

内部实现的友元函数

```

#include <iostream>
using namespace std;

template <typename T>
class A
{
    // 内部实现友元函数， 接收当前类模板的引用时，可以指定当前的类模板的泛型

```

```

// 友元函数（全局性的），类的外部直接访问，不需要类对象调用
friend void showIn(A<T> &a)
{
    cout << a.item << endl;
}

private:
    T item;

public:
    A(T item)
    {
        this->item = item;
    }
};

int main()
{
    A<int> a(20);
    showIn(a);
    return 0;
}

```

```

disen@qfxa:~/code2/day07$ ./a.out
20

```

外部实现的友元函数

【注意】外部实现的友元函数是函数模板时，必须在类定义之前声明。在类中声明友元全局函数时，必须使用<>空泛型表示此函数是函数模板。

```

#include <iostream>
using namespace std;

template <typename T>
class A;
template <typename T>
void showOut(A<T> &a);

template <typename T>
class A
{
    // 外部实现的友元函数，它有自己的模板
    // <> 空泛型表示外部是 函数模板，模板的泛型同当前类的泛型
    // 要求：必须之前先声明此函数为函数模板
    friend void showOut<>(A<T> &a);

private:
    T item;

public:
    A(T item)
    {
        this->item = item;
    }
}

```

```

    }
};

template <typename T>
void showOut(A<T> &a) // 全局友元函数
{
    cout << "out item is " << a.item << endl;
}

int main()
{
    A<int> a(50);
    showOut(a);
    return 0;
}

```

```

disen@qfxa:~/code2/day07$ ./a.out
out item is 50

```

友元函数模板

格式:

```

template<typename T>
friend 返回值类型 函数名(T &参数名);

```

如:

```

#include <iostream>
using namespace std;

template <typename T>
class A
{
    // 友元函数模板，声明的友元函数名不需要加 <> 空泛型
    template <typename U>
    friend void show(A<U> &a);

private:
    T item;

public:
    A(T item)
    {
        this->item = item;
    }
};

template <typename U>
void show(A<U> &a)
{
    cout << "template item is " << a.item << endl;
}

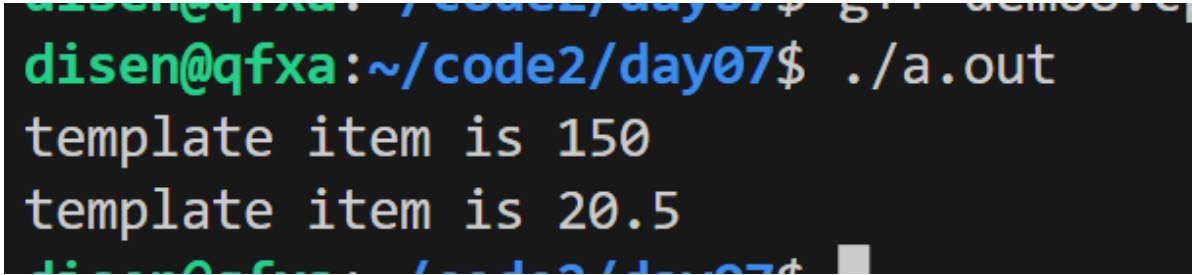
```

```

int main()
{
    A<int> a(150);
    show(a);

    A<float> a2(20.5);
    show(a2);
    return 0;
}

```



```

disen@qfxa:~/code2/day07$ ./a.out
template item is 150
template item is 20.5

```

2.4 类模板的应用

设计一个数组模板类(MyArray),完成对不同类型元素的管理

```

#include <iostream>

using namespace std;

template <typename T>
class MyArray
{
private:
    int index;    // 当前元素的位置
    T *arr;       // 指向数组的指针
    int maxSize;  // 最大容量
public:
    MyArray(int capacity)
    {
        maxSize = capacity;
        arr = new T[maxSize];
        index = 0;
    }
    MyArray(const MyArray<T> &other)
    {
        this->index = other.index;
        this->maxSize = other.maxSize;
        this->arr = new T[this->maxSize];
        for (int i = 0; i <= this->maxSize; i++)
        {
            this->arr[i] = other.arr[i];
        }
    }

    ~MyArray()
    {
        delete this->arr;
    }
}

```



```

    }

    T &get(int i)
    {
        return arr[i];
    }

    T &operator[](int i)
    {
        return arr[i];
    }

    MyArray<T> &push(T item)
    {
        if (index < maxSize)
            arr[index++] = item;

        return *this;
    }

    T pop()
    {
        return arr[--index];
    }
};

int main(int argc, char const *argv[])
{
    MyArray<int> a1 = MyArray<int>(20);
    a1[0] = 100;
    a1[1] = 200;
    MyArray<int> a2 = a1;
    cout << a2[0] << "," << a2[1] << endl;

    a1.push(5).push(9).push(10).push(2);
    for (int i = 0; i < 4; i++)
    {
        cout << a1.pop() << endl;
    }
    return 0;
}

```

100,200

2

10

9

5

三、C++类型转换

类型转换(cast)是将一种数据类型转换成另一种数据类型，一般类型转换由编译器自动完成，除非强制类型转换需要手动完成。

【注意】一般情况下，尽量少的去使用类型转换，除非用来解决非常特殊的问题。

标准 c++ 提供了一个显示的转换的语法，来替代旧的 C 风格的类型转换，新类型的强制转换可以提供更好的控制强制转换过程，允许控制各种不同种类的强制转换。

C++ 风格的强制转换其他的好处是，它们能更清晰的表明它们要干什么。

C++ 新式风格的强制转换方式：

```
目标类型 变量名 = 新式转换函数名<目标类型>(转换变量);
```

新式转换函数名：static_cast, dynamic_cast, const_cast, reinterpret_cast

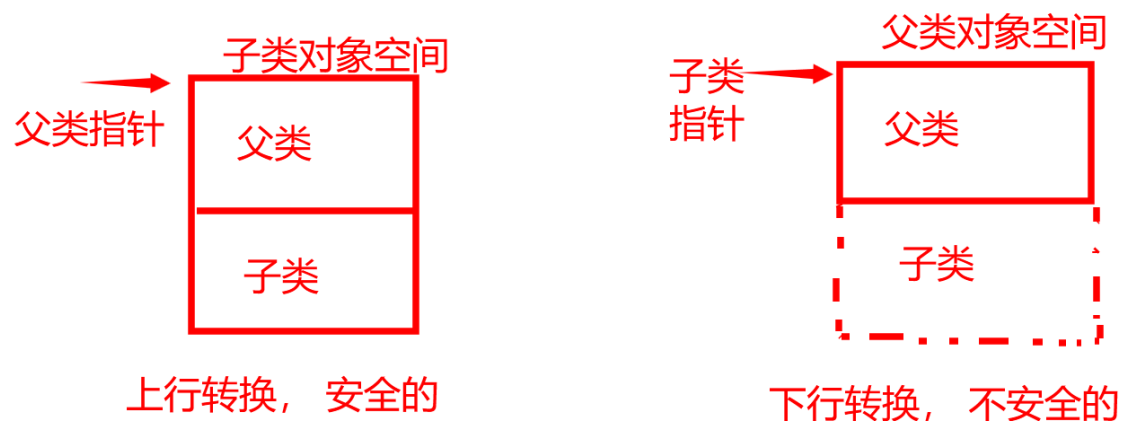
3.1 静态转换

static_cast 静态转换函数模板

上行转换 和下行转换的说明：

上行转换：把派生类的指针或引用转换成基类表示，安全的。【多态的体现】

下行转换：把基类指针或引用转换成派生类表示，不安全的。【调用子类的扩展功能】



3.1.1 基本数据类型转换

如：

```
#include <iostream>
using namespace std;
int test1()
{
    float f = 97.5;
    // char c = f; // c => 97.5
    // 新式转换比较安全的
    char c = static_cast<char>(f); // c => 97
    cout << "c=" << c << endl;
}
int main()
{
    test1();
    return 0;
}
```

```
disen@qfxa:~/code2/day07$ ./a.out
C=a
```

3.1.2 基本数据类型的指针或引用转换

【注意】不同的基本数据类型的指针不能相互静态（或动态）转换。

```
#include <iostream>
using namespace std;
int test2()
{
    float f = 97.5;
    // int *p = &f; // 不准许基本数据类型的指针之间的转换
    // int *p = (int *)&f; // 不安全的
    float *pf = &f;
    int *p = static_cast<int *>(pf); // 不允许将float* => int *
    // int *p = reinterpret_cast<int *>(pf); // 类似于强转
    cout << "*p=" << *p << endl;
}
int main()
{
    test2();
    return 0;
}
```

3.1.3 继承关系之间的指针转换

支持父类与子类的指针的静态上行或下行的转换。

```
#include <iostream>
using namespace std;
class A
{
public:
    int x;
    A(int x) : x(x) {}
};
class B : public A
{
public:
    int y;
    B(int x, int y) : A(x), y(y) {}
};

int main()
{
    A *a = new A(5);
    B *b = new B(10, 20);

    A *c = static_cast<A *>(b); // 静态的上行转换
    // B *b2 = (B *)a; // 强制下行转换，不安全的
    // B *b2 = static_cast<B *>(a); // 静态的下行转换，不安全的
    cout << "a.x=" << a->x << endl;
    cout << "b2.x=" << b2->x << ",b2.y=" << b2->y << endl;
}
```

```

    delete a;
    delete b;
    return 0;
}

```

3.1.4 不相关的两类的引用或指针转换

静态转换无法实现, 不支持

```

#include <iostream>
using namespace std;
class A{};
class B{};

int main()
{
    A *a = new A;
    B *b = new B;

    A *c = static_cast<A *>(b);

    delete a;
    delete b;
    return 0;
}

```

```

disen@qfxa:~/code2/day07$ g++ demo11.cpp
demo11.cpp: In function 'int main()':
demo11.cpp:15:30: error: invalid static_cast from type 'B*' to type 'A*'
    A *c = static_cast<A *>(b);
                        ^

```

3.2 动态转换

动态转换的dynamic_cast<>()函数模板

dynamiccast 主要用于类层次间的上行转换和下行转换;

在类层次间进行上行转换时, dynamiccast 和 staticcast 的效果是一样的;

在进行下行转换时, dynamiccast 具有类型检查的功能, 比 static_cast 更安全

```

#include <iostream>

using namespace std;
class A
{
public:
    int x;
    A(int x) : x(x) {}
    void show()
    {
        cout << "A x=" << x << endl;
    }
};
class B : public A
{
}

```

```

public:
    B(int x) : A(x) {}
    void print()
    {
        cout << "B print x=" << x << endl;
    }
};

class C
{
};

void test1()
{
    int a = 65;
    // 动态转换不支持基本的数据类型之间的转换
    // char c = dynamic_cast<char>(a);
    // char *c = dynamic_cast<char *>(&a);
}

void test2()
{
    A a1 = A(10);
    B b1 = B(20);

    // 上行
    A &a2 = dynamic_cast<A &>(b1);
    a2.show();
    // a2.print(); // 不存在print成员

    // 下行转换：动态不支持
    // B &b2 = dynamic_cast<B &>(a2);
    // b2.print();
}

void test3()
{
    // 不相关类型之间的转换
    A *a = new A(20);
    // C *p = dynamic_cast<C *>(a);
}

int main(int argc, char const *argv[])
{
    test3();
    return 0;
}

```

```

demo12.cpp: In function 'void test3()':
demo12.cpp:55:31: error: cannot dynamic_cast 'a' (of type 'class A*') to type 'class C*' (source type is not polymorphic)
    C *p = dynamic_cast<C *>(a);
                        ^

```

3.3 常量转换

常量转换 `const_cast<>()` 可以将指针或引用变量转化为`const`指针或引用变量，也可以将`const`指针或引用变量转化为指针或引用变量。

注意: 不能直接对非指针和非引用的变量使用 `const_cast`去直接移除它的`const`

如:

```
#include <iostream>

using namespace std;

void change(const int *p, int value)
{
    int *q = const_cast<int *>(p);
    *q = value;
}

int main(int argc, char const *argv[])
{
    int a = 10;
    const int b = 30; // 存储在符号表，在栈中没有空间
    // 1. 将变量转化为 const引用变量 【可以】
    const int &a1 = const_cast<const int &>(a);
    // 2. 将变量地址转化为 const指针变量 【可以】
    const int *a2 = const_cast<const int *>(&a);
    // *a2 = 500; // error

    // 3. 去掉b的const
    // b在取地址时在栈中开辟空间，b1指向空间
    // int b1 = const_cast<int>(b); // 【不支持的】
    int &b1 = const_cast<int &>(b);
    b1 = 100;
    cout << "b=" << b << endl;
    cout << "b1=" << b1 << endl;

    // 4. 去掉a2的const
    int *a3 = const_cast<int *>(a2);
    *a3 = 500;
    cout << "*a2=" << *a2 << endl;

    // 5. 将a3 添加 const
    const int *a4 = const_cast<const int *>(a3);
    // *a4 = 9;
    return 0;
}
```

```
disen@qfxa:~/code2/day07$ ./a.out
b=30
b1=100
*a2=500
```

3.4 重新解释转换

这是最不安全的一种转换机制，最有可能出问题。

主要用于将一种数据类型从一种类型转换为另一种类型。

它可以将一个指针转换成一个整数，也可以将一个整数转换成一个指针。

如：

```
#include <iostream>

using namespace std;

class A
{
public:
    int x;
    void show()
    {
        cout << "A show x=" << x << endl;
    }
};

class B
{
public:
    int x;
    void show()
    {
        cout << "B show x=" << x << endl;
    }
};

int main(int argc, char const *argv[])
{
    A *a = new A();
    a->x = 100;
    B *b = new B();
    b->x = 200;
    // B *b2 = static_cast<B *>(a); // error
    // B *b2 = dynamic_cast<B *>(a); // error
    B *b2 = reinterpret_cast<B *>(a);
    b2->show(); // B show() 还是 A show() : B 的show(), 显示x是A的
    return 0;
}
```

```
disen@qfxa:~/code2/day07$ g++ demo14.cpp
disen@qfxa:~/code2/day07$ ./a.out
B show x=100
disen@qfxa:~/code2/day07$
```

