

# 第六天课程笔记

## 一、回顾上周知识点

### 1.1 c扩展c++

`::` 作用域（全局变量）  
`namespace`（全局性声明），`using` 声明，`using` 编译，命名空间名`::`成员（变量、函数）  
强化了全局变量（定义和声明不分开）  
强化了`struct`结构体，定义结构体变量时，不需要 `struct`关键字  
强化了三目运算（表达返回变量）  
引用：`&`运算符，类似于指针，为变量的别名  
函数的重载，参数默认值，占位参数  
`const`优化（常量值的初始化定义`const`变量时，不会分配空间，只会在符号表中创建）  
类型强转（c++中 `void *` 赋值给目标类型变量时，必须强转）  
新增`"bool"`布尔类型，`true`真（非0），`false`假（0）  
新增 `inline` 内联函数，同有参宏，每一次使用时展开一次，不会调入栈中。  
`extern "C"` 应用：兼容c定义的函数，避免c++对函数重命名（因函数重载的特性引起来）

### 1.2 类与对象

类：c++的数据类型，从具有共同特性和行为的多个对象抽象出来的。在类中，可以将属性和方法封装到类中，并使用访问权限修饰符（`private`，`protected`，`public`）控制成员的访问。

对象：通过类，创建的或实例化的。

如A类的对象创建的方式：

```
A a;  
A a = A();  
A a = A(12);  
A a = 12;  
A *a = new A();  
delete a;  
  
A *a = new A[10];  
delete[] a;
```

构造函数与析构函数：

默认存在三个函数：无参构造函数、拷贝构造函数、析构函数

构造函数支持重载，默认编译器为所有类增加无参构造函数，但是显式地声明或定义有参数的构造函数时，默认不会增加无参构造，如需要时，手动添加无参构造函数。

构造函数与析构函数的名称：类名，无返回值

类成员函数：普通成员函数、静态成员函数、构造函数、析构函数、运算符重载函数等，可以在类中声明，类外定义。

类外实现函数时，格式：

返回值类型 类名`::`成员函数名(参数列表){ }

## 1.3 static静态成员和const成员

静态成员变量不占用对象空间的大小，静态成员变量可以在类外部直接通过类名访问：类名::静态成员。  
静态成员函数内，不能访问类中非静态成员，只能访问类内的静态成员。

```
static int a;  
static void xxx() {}
```

const成员：成员函数、成员对象

const修饰成员函数的方式：

```
void xxx() const { }
```

const成员函数内不能修改成员变量的值

const成员对象，不能修改成员变量值，可以调用const成员函数。

## 1.4 友元

友元：在类的外部，可以访问类内部所有成员。

存在的形式有三种：友元全局函数、友元成员函数、友元类

友元全局函数：cout, cin的重载 <<, >>

友元成员函数：++, --, +, -, =, [], -, \*, (), +=, -=, >, <, >=, <=, ==, &=等

## 1.5 运算符重载

语法：

```
operator运算符(参数列表){  
  
}
```

运算符的特性或功能决定重载函数的参数列表。

## 1.6 继承与派生

父类：基类

子类：派生类

继承的方式：public [推荐], protected, private，将父类的成员以继承方式将指定的访问权限修改为同级及以上。

c++支持类的多继承：一个子类具有多个父类

菱形继承：一个子类的多个父类又同属一个父类，在子类中调用父类的成员时，存在二义性。决定此问题子的多个父类（同属一个父类时，使用virtual继承 - 虚继承 vbptr指针+偏移量）。

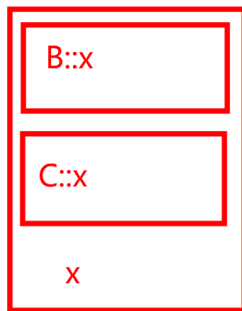
如：多继承（同属一个超级父类）产生的问题

```

class A{
public:
    int x;
};
class B : public A{};
class C : public A{};
class D : public B, public C {}

int main(){
    D d1;
    d1.x = 100; // x从B类来的，还是从C类的?
    return 0;
}

```



栈区 d1

```

class A: int x;           代码区

class B : public A {}

class C: public A {}

class D : public B, public C {}

```

访问d1的x的，x从B和C都继承过来的，冲突了。

解决此问题：

```

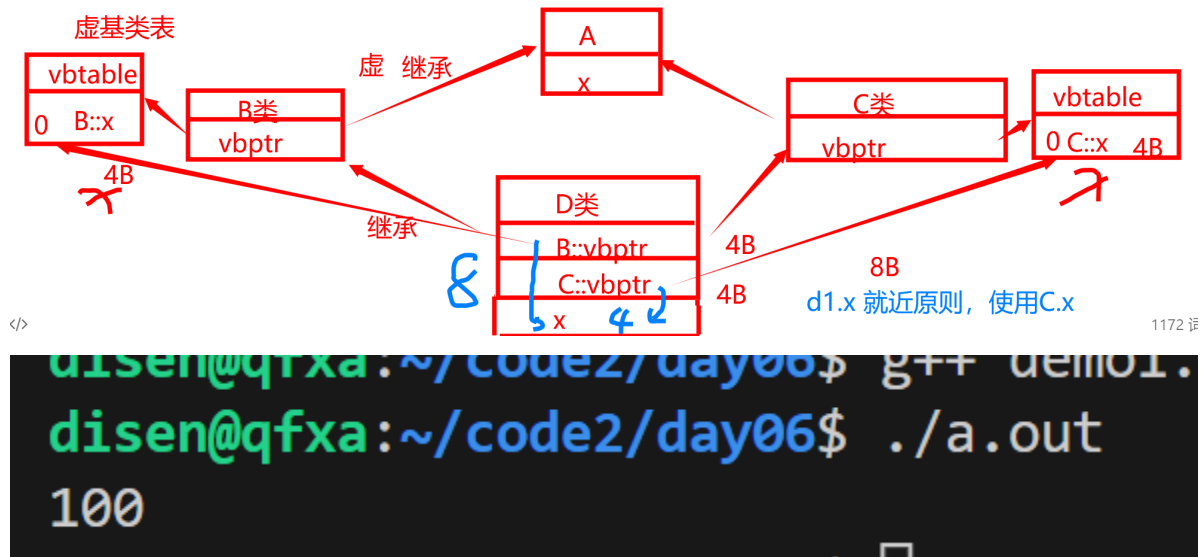
#include <iostream>

using namespace std;
class A
{
public:
    int x;
};
class B : virtual public A
{
public:
    B()
    {
        x = 50;
    }
};
class C : virtual public A
{
public:
    C()
    {
        x = 100;
    }
};
class D : public B, public C
{
}

```

```
};

int main(int argc, char const *argv[])
{
    D d1;
    cout << d1.x << endl;
    return 0;
}
```



## 二、多态【重要】

### 2.1 多态基本概念

多态性(polymorphism)提供接口与具体实现之间的另一层隔离, 从而将"what"和"how"分离开来。多态性改善了代码的可读性和组织性, 同时也使创建的程序具有可扩展性, 项目不仅在最初创建时期可以扩展, 而且当项目在有新的功能时也能扩展。

c++支持静态多态(编译时多态)和动态多态(运行时)

运算符重载和函数重载就是编译时多态  
派生类和虚函数实现的是运行时多态

静态多态和动态多态的区别:

函数地址是早绑定(静态联编)还是晚绑定(动态联编)  
在编译时确定函数的入口, 属于早绑定  
在编译时不确定函数的入口, 只有在运行时才确定的, 属于晚绑定。

### 2.2 初次体验多态示例

```
#include <iostream>

using namespace std;
class AbsCalculator
{ // 定义计算器
protected:
```

```

    int a, b;

public:
    void setA(int a) { this->a = a; }
    virtual void setB(int b) { this->b = b; }
    // 扩展的功能，让子类去实现
    virtual int getResult() = 0; // 纯虚函数
};

class AddCa : public AbsCaculator
{
    // 必须实现纯虚函数
    int getResult()
    {
        return a + b;
    }
};

class SubCa : public AbsCaculator
{
    // 必须实现纯虚函数
    int getResult()
    {
        return a - b;
    }
};

// 多态的体现：
// 1) 定义函数时，形参的类型为父类对象的指针，
// 2) 调用函数时，实参的类型为子类对象的指针
int getResult(AbsCaculator *c, int a, int b)
{
    c->setA(a);
    c->setB(b);
    return c->getResult();
}

int main(int argc, char const *argv[])
{
    // AbsCaculator c1; // error 抽象类型不能定义对象
    SubCa *c1 = new SubCa();
    cout << getResult(c1, 10, 20) << endl;
    delete c1;
    return 0;
}

```

```

disen@qfxa:~/code2/day06$ ./a.out
-10

```

#### 【小结】

多态的体现，父类对象的指针指向是子类对象的空间。  
多态的前提条件：继承+重写（重新实现父类的虚函数或纯虚函数）。

## 2.3 向上类型转换

对象可以作为自己的类或者作为它的基类的对象来使用。还能通过基类的地址来操作它。取一个对象的地址(指针或引用)，并将其作为基类的地址来处理，这种称为向上类型转换。

即父类引用或指针可以指向子类对象，通过父类指针或引用来操作子类对象。

问题：如果定义是父类的引用或指针，指向是子类对象时，通过父类引用或指针调用非虚函数时，优先调用父类的函数；解决此问题，可以在父类中声明此函数为虚函数。

如：存在问题的，父类的引用调用的是父类的函数

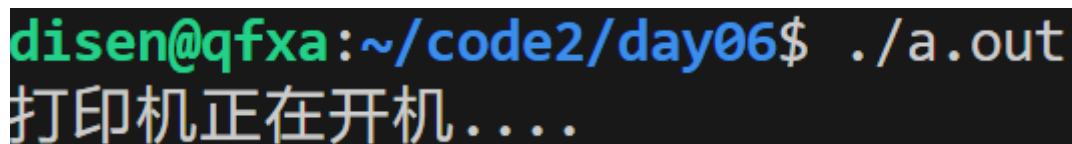
```
#include <iostream>

using namespace std;
class Printer
{
public:
    void open()
    {
        cout << "打印机正在开机...." << endl;
    }
};

class DellPrinter : public Printer
{
    void open()
    {
        cout << "Dell打印机 ^_^...." << endl;
    }
};

class HuaweiPrinter : public Printer
{
    void open()
    {
        cout << "Huawei打印机 *)*...." << endl;
    }
};

int main(int argc, char const *argv[])
{
    DellPrinter dellPrinter;
    // 多态：通过父类的引用或指针调用子类对象的成员
    // 子类对象自动向上转型为父类的类型
    Printer &printer = dellPrinter;
    printer.open();
    return 0;
}
```



```
disen@qfxa:~/code2/day06$ ./a.out
打印机正在开机....
```

如2：将父类的函数改为虚函数

```

#include <iostream>

using namespace std;
class Printer
{
public:
    virtual void open()
    {
        cout << "打印机正在开机...." << endl;
    }
};

class DellPrinter : public Printer
{
    void open()
    {
        cout << "Dell打印机 ^_^...." << endl;
    }
};

class HuaweiPrinter : public Printer
{
    void open()
    {
        cout << "Huawei打印机 *)*...." << endl;
    }
};

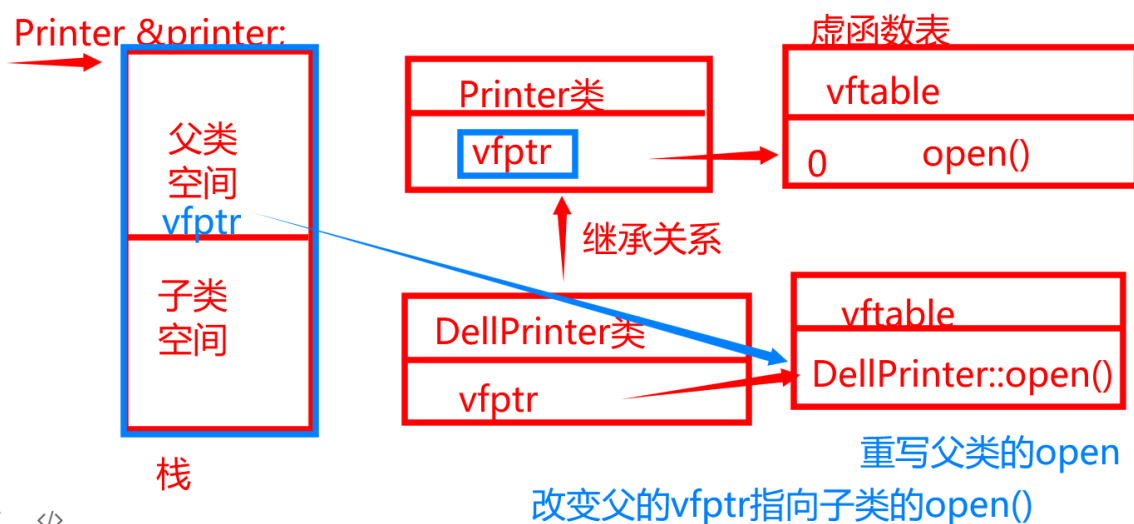
int main(int argc, char const *argv[])
{
    DellPrinter dellPrinter;
    // 多态: 通过父类的引用或指针调用子类对象的成员
    Printer &printer = dellPrinter;
    printer.open();
    return 0;
}

```

```

disen@qfxa:~/code2/day06$ ./a.out
Dell打印机 ^_^....

```



如果父类的函数是一个虚函数时，则会产生一个vfptr(虚函数指针)指向一个虚函数表（表中存放的是虚函数入口地址和偏移量-（多继承））。子类重写了父类的虚函数时，则会更新父类的vfptr指向的位置为子类的重写函数，所以在父类引用调用函数时（父类引用指向是子类对象），则调用vfptr实际上指向函数位置（子类的重写函数）。

## 2.4 多态的条件和应用方式

继承+子类重写父类的虚函数（返回值、函数名、参数列表必须相同，virtual关键字可写可不写，建议写上）。

【注意】子类扩展新的功能和属性，不能作为多态的方式应用。

多态的应用方式：

- 1) 局部使用，父类的引用或指针指向子类对象
- 2) 设计函数时，父类的引用或指针作为函数（全局函数、某类的成员函数）的参数，调用函数时，传入的子类的对象。

如：宠物店里买宠物的场景

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

class Animal{
protected:
    string name;
    float price;
public:
    Animal(const string &name, float price){
        this->name = name;
        this->price = price;
    }
    virtual void buy(); // 类内声明的虚函数
};

// 类外实现类中的虚函数，不需要 virtual关键字
void Animal::buy(){
    cout << "小宠物 " << name << " 卖了 " << price << endl;
}

class Dog:public Animal{
private:
    int age;
public:
    Dog(string name, float price, int age): Animal(name, price),age(age){}
    virtual void buy(){ // 重写父类的虚函数
        cout << age << " 岁小狗 " << name << " 卖了 " << price << endl;
    }
    void eat(){ // 扩展的新功能
        cout << name << "喝酒" << endl;
    }
};

class Cat:public Animal{
public:
```



```

Cat(string name, float price): Animal(name, price){}
virtual void buy(){ // 重写父类的虚函数
    cout << "小猫 " << name << " 卖了 " << price << endl;
}
void eat(){ // 扩展的新功能
    cout << name << "吃鱼" << endl;
}
};

class AnimalHome{
public:
    // 多态的应用之一
    void buy(Animal *animal){
        cout << "-----动物之家正在出售小宠物-----" << endl;
        animal->buy();
        cout << "-----出售小宠物成功-----" << endl;

        delete animal;
    }
};

int main(){
    AnimalHome home;
    // 多态的应用
    home.buy(new Dog("小黄", 900, 2));
    home.buy(new Cat("小白白", 500));
    return 0;
}

```

```

disen@qfxxa:~/code2/day06$ ./a.out
-----动物之家正在出售小宠物-----
2 岁小狗 小黄 卖了 900
-----出售小宠物成功-----
-----动物之家正在出售小宠物-----
小猫 小白白 卖了 500
-----出售小宠物成功-----

```

如：继以上的类，重新写main函数：

```

int main(){
    Animal *a1 = new Dog("小黄", 900, 3); // 子类到父类的类型，自动转换
    // 可调用的函数： a1.buy();
    // 如何调用 Dog类的eat()函数的呢？
    // ((Dog *)a1)->eat(); // 从父类到子类的类型，必须强制转换
    Dog *dog1 = (Dog *) a1;
    dog1->eat();
    return 0;
}

```

```
disen@qfxa:~/code2/day06$ ./a.out
3 岁小狗 小黄 卖了 900
小黄喝酒
```

## 2.5 纯虚函数和多继承

纯虚函数可以解决多继承的二义性的问题。

如果一个类存在一个纯虚函数，则这个类为抽象类，抽象类不创建实例（对象）。

如果一个类中的所有函数都是纯虚函数，又称之为“接口”。（c++没有interface接口关键字，但可以实现接口的特性。接口的定义：只有功能（接口）的声明，没有属性）。

纯虚函数的格式：

```
virtual 返回值类型 函数名(参数列表)=0;
```

多重继承接口不会带来二义性和复杂性问题。接口类只是一个功能声明，并不是功能实现，子类需要根据功能说明定义功能实现。

**【注意】除了析构函数外，其他声明都可以是纯虚函数。**

如：多层继承（ITv-> AbsAndroidTv -> Abs5GTV -> XiaoMITv）

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

// 接口类
class ITv
{
public:
    virtual void power() = 0;
    virtual void volumeUp() = 0;
};

// 抽象类， 从父类中继承一个纯虚函数 volumeUp()，并没有实现
class AbsAndroidTv : public ITv
{
public:
    virtual void power()
    {
        cout << "Android Tv 正在打开" << endl;
    }
    // virtual void volumeUp() = 0;
    virtual void installApp()
    {
        cout << "Android Tv 正在安装程序" << endl;
    }
};

class Abs5GTV : public AbsAndroidTv
{
public:
    virtual void power()
    {
```

```

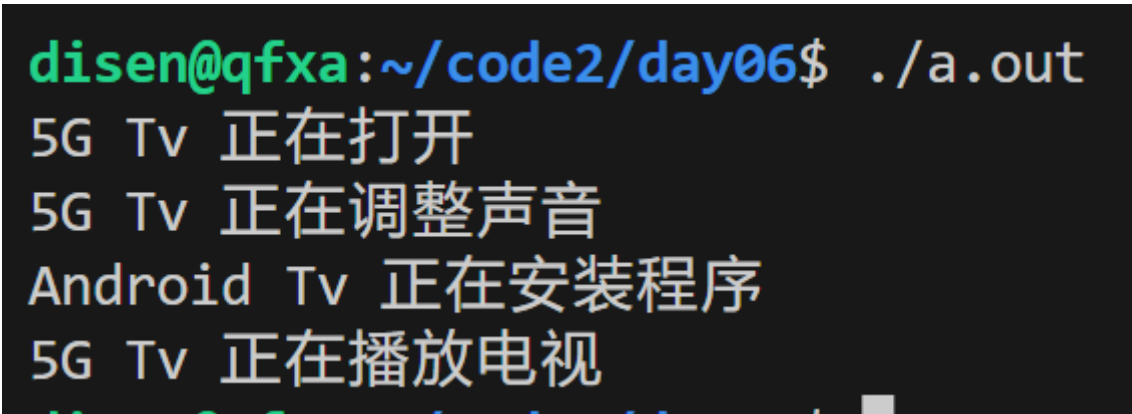
        cout << "5G Tv 正在打开" << endl;
    }
    virtual void volumeUp()
    {
        cout << "5G Tv 正在调整声音" << endl;
    }
    virtual void playVideo()=0;
};

class XiaoMITv : public Abs5GTV
{
private:
    int width, height;
    int mVolume;

public:
    XiaoMITv() {}
    XiaoMITv(int width, int height) : width(width), height(height), mVolume(10)
    {}
    virtual void volumeUp() // 重写父类的函数
    {
        if (mVolume < 100)
            mVolume++;
        cout << "Tv当前声量为 " << mVolume << endl;
    }
    virtual void playVideo(){ // 实现父类的纯虚函数
        cout << "xiaoMITv 正在播放动漫..." << endl;
    }
};

int main()
{
    XiaoMITv tv(2900, 2100);
    tv.power();
    tv.volumeUp();
    tv.installApp();
    tv.playVideo();
    return 0;
}

```



```

disen@qfxa:~/code2/day06$ ./a.out
5G Tv 正在打开
5G Tv 正在调整声音
Android Tv 正在安装程序
5G Tv 正在播放电视

```

如：多继承时，超级父类 不能出现纯虚函数，子类的父类需要从超级父类虚继承。

```
#include <iostream>
```

```
using namespace std;

// 接口类
class ITv
{
public:
    virtual void power()
    {
        cout << "Tv 正在打开" << endl;
    };
    virtual void volumeUp()
    {
        cout << "Tv 正在调整声音" << endl;
    };
};

class AbsAndroidTv : virtual public ITv
{
public:
    virtual void installApp()
    {
        cout << "Android Tv 正在安装程序" << endl;
    };
};

class Abs5GTV : virtual public ITv
{
public:
    virtual void playVideo()
    {
        cout << "5G Tv 正在播放大片..." << endl;
    };
};

class XiaoMITv : public AbsAndroidTv, public Abs5GTV
{
private:
    int width, height;
    int mVolume;

public:
    XiaoMITv(int width, int height) : width(width), height(height), mVolume(10)
    {}
};

int main()
{
    XiaoMITv tv(2900, 2100);
    tv.power();
    tv.volumeUp();
    tv.installApp();
    tv.playVideo();
    return 0;
}
```

```
disen@qfxa:~/code2/day06$ ./a.out
Tv 正在打开
Tv 正在调整声音
Android Tv 正在安装程序
5G Tv 正在播放电视
```

如3：继承多个接口类

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

// 接口类
class IA
{
public:
    virtual void a() = 0;
};

// 接口类
class IB : public IA
{
public:
    virtual void b() = 0;
};

// 接口类（特殊的抽象类）
class IC : public IA
{
public:
    virtual void c() = 0;
};

class ID : public IB, public IC // 继承多个接口类
{
public:
    void a()
    {
        cout << " --a- " << endl;
    }

    void b()
    {
        cout << " --b- " << endl;
    }

    void c()
    {
        cout << "--c---" << endl;
    }
};

int main()
{
    ID d;
    d.a();
}
```

```

    d.b();
    d.c();
    return 0;
}

```

如4：优化Tv的接口类，一个类可以实现（C++继承）多个接口类

```

#include <iostream>
using namespace std;

// 接口类
class ITv
{
public:
    virtual void power() = 0;
    virtual void volumeUp() = 0;
};

class IAndroidTv : public ITv
{
public:
    virtual void installApp() = 0;
};

class I5GTV : public ITv
{
public:
    virtual void playVideo() = 0;
};

class XiaoMITv : public IAndroidTv, public I5GTV
{
private:
    int width, height;
    int mVolume;

public:
    XiaoMITv() {}
    XiaoMITv(int width, int height) : width(width), height(height), mVolume(10)
    {}

    virtual void power()
    {
        cout << "xiaoMITv 正在打开电视..." << endl;
    }

    virtual void volumeUp() // 重写父类的函数
    {
        if (mVolume < 100)
            mVolume++;
        cout << "Tv当前声量为 " << mVolume << endl;
    }

    virtual void installApp()
    {
        cout << "XiaoMI Tv正在安装应用..." << endl;
    }

    virtual void playVideo()
    { // 实现父类的纯虚函数

```

```

        cout << "xiaoMITv 正在播放动漫..." << endl;
    }
};

int main()
{
    xiaoMITv tv(2900, 2100);
    tv.power();
    tv.volumeUp();
    tv.installApp();
    tv.playVideo();
    return 0;
}

```

```

disen@qfxa:~/code2/day06$ ./a.out
xiaoMITv 正在打开电视...
Tv当前声量为 11
XiaoMI Tv正在安装应用...
xiaoMITv 正在播放动漫...

```

## 2.6 虚析构函数

虚析构函数作用: 虚析构函数是为了解决基类的指针指向派生类对象, 并用基类的指针删除派生类对象。

如: 删除A类指针时, 执行是A类的析构函数

```

#include <iostream>

using namespace std;

class A
{
public:
    A()
    {
        cout << "A()" << endl;
    }
    ~A()
    {
        cout << "~A()" << endl;
    }
};

class B : public A
{
public:
    B()
    {
        cout << "B()" << endl;
    }
}

```

```

    ~B()
    {
        cout << "~B()" << endl;
    }
};

int main(int argc, char const *argv[])
{
    // 多态的应用
    A *a1 = new B();
    delete a1;
    return 0;
}

```

运行结果:

```

A()
B()
~A()

```

如2: 解决上面的问题, 将A类的析构设置为虚析构 (前面加 virtual关键字)

```

#include <iostream>

using namespace std;

class A
{
public:
    A()
    {
        cout << "A()" << endl;
    }
    virtual ~A()
    {
        cout << "~A()" << endl;
    }
};

class B : public A
{
public:
    B()
    {
        cout << "B()" << endl;
    }
    ~B()
    {
        cout << "~B()" << endl;
    }
};

int main(int argc, char const *argv[])
{
    // 多态的应用
    A *a1 = new B();
}

```



```

delete a1;
return 0;
}

```

```

disen@qfxa:~/code2/day06$ ./a.out
A()
B()
~B()
~A()

```

如3： 纯虚析构函数

在类的外部必须实现纯虚析构函数。当前类也是抽象类。

```

#include <iostream>

using namespace std;

class A
{
public:
    A()
    {
        cout << "A()" << endl;
    }
    virtual ~A() = 0; // 将当前类转化为抽象类
};

// 纯虚析构函数必须声明函数体
A::~~A()
{
    cout << "~A()" << endl;
}

class B : public A
{
public:
    B()
    {
        cout << "B()" << endl;
    }
    ~B()
    {
        cout << "~B()" << endl;
    }
};

int main(int argc, char const *argv[])
{
    // A a2; 不能实例化
    // 多态的应用
}

```

```

    A *a1 = new B();
    delete a1;
    return 0;
}

```

## 2.7 重写&重载 【面试】

override 重写, overload 重载

### 重载:

同一作用域的同名函数, 参数个数、参数顺序、参数类型等不同, 返回值无关的;

另外const也可以引起重载, 如 a(int x) 和 a(const int x) 是两个函数, 即为函数的重载。

### 重定义 (隐藏): 不能体现多态的

有继承+子类 (派生类) 重新定义父类 (基类) 的 (静态与非静态) 同名成员 (非 virtual 函数) 。

### 重写 (覆盖): 可以体现多态的

有继承+子类 (派生类) 重写父类 (基类) 的 virtual 函数, 函数返回值, 函数名字, 函数参数, 必须和基类中的虚函数一致。

如:

```

#include <iostream>

using namespace std;

class A
{
public:
    void f1() { cout << "f1" << endl; }
    void f1(int) { cout << "f1(int)" << endl; }
    void f1(int, int) { cout << "f1(int,int)" << endl; }
    void f4() { cout << "f4()" << endl; }
    virtual void f5() { cout << "virtual f5() " << endl; }
};

class B : public A
{
public:
    void f1(double) // 函数的重定义, 父类的f1的所有函数将隐藏了
    {
        cout << "f1(double)" << endl;
    }
    // void f5(int) // 重定义
    void f5() // 重写
    {
        cout << "B f5()" << endl;
    }
};

int main(int argc, char const *argv[])
{
    B b1;
    // b1.f1();
}

```

```
b1.f1(10);  
b1.f1(10.25);  
b1.f5();  
return 0;  
}
```

```
f1(double)  
f1(double)  
B f5()
```

## 三、C++模板

### 3.1 模板的概论

C++提供了函数模板（function template），函数模板是将函数的参数类型不具体化，在函数使用时，再给定具体的参数的数据类型。

如：实现两数值的相加函数，考虑的问题，两个整数，还两个小数，还是一个整数一个小数呢？

如果考虑的情况较多时，通过函数的重载来完成，但C++支持模板方式，可以先定义功能，再具体使用，减少代码量。

**c++提供两种模板机制：函数模板和类模板**

无论哪一种机制，都属于 参数类型的模板，又称之为参数模板， 在Java语言中，称之为泛型。

### 3.2 函数模板

函数模板可以自动推导参数的类型，当然也可以显式指定类型。

语法：

```
template<class或typename T>  
函数的返回值 函数名(T &n1, T &n2) {  
    // 函数体  
}
```

【推荐】在定义函数模板时，使用typename，表示T是某一种 类型的名称。

如：

函数重载的方式

```
void swap(int &a, int &b){}  
void swap(char &a, char &b){}  
void swap(float &a, float &b){}
```

函数模板的方式：

```
template<class T>
void swapF(T &a, T &b){ // 要求a,b的两个数据保持一致
    a = a^b;
    b = a^b;
    a = a^b;
}
```

```
#include <iostream>

using namespace std;

template <typename T1, typename T2> //解决int,char两个变量的交换的问题
void swapF(T1 &a, T2 &b)
{
    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
}

int main(int argc, char const *argv[])
{
    int a = 10, b = 20;
    cout << "a=" << a << ",b=" << b << endl;
    swapF(a, b); // 自动推演T模板的类型
    cout << "a=" << a << ",b=" << b << endl;

    char c = 30;
    cout << "a=" << a << ",c=" << (int)c << endl;
    swapF<int, char>(a, c); // 显示指定T1的类型为int,T2为char
    cout << "a=" << a << ",c=" << (int)c << endl;
    return 0;
}
```

```
disen@qfxxa:~/code2/day06$ g++ demo11.cpp
disen@qfxxa:~/code2/day06$ ./a.out
a=10,b=20
a=20,b=10
a=20,c=30
a=30,c=20
disen@qfxxa:~/code2/day06$
```

如：课堂练习，使用函数模板实现对 char 和 int 类型数组进行排序和打印

```
template<typename T>
void sort(T arr[], int size){
    for(int i=0;i<size-1;i++){
        int min=i;
        for(int j=i+1; j < size; j++){
            if(arr[min] > arr[j])
                min = j;
        }
        if(min != i){
            arr[i] = arr[i] ^ arr[min];
            arr[min] = arr[i] ^ arr[min];
            arr[i] = arr[i] ^ arr[min];
        }
    }
}
```

```

        arr[min] = arr[i] ^ arr[min];
        arr[i] = arr[i] ^ arr[min];
    }
}

template<typename T>
void printArr(T arr[], int size){
    for(int i=0; i< size; i++){
        cout << arr[i] << " ";
    }
    cout << endl;
}

```

完整的示例:

```

#include <iostream>

using namespace std;

template <typename T>
void sort(T arr[], int size)
{
    for (int i = 0; i < size - 1; i++)
    {
        int min = i;
        for (int j = i + 1; j < size; j++)
        {
            if (arr[min] > arr[j])
                min = j;
        }
        if (min != i)
        {
            arr[i] = arr[i] ^ arr[min];
            arr[min] = arr[i] ^ arr[min];
            arr[i] = arr[i] ^ arr[min];
        }
    }
}

template <typename T>
void printArr(T arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    int arr[] = {1, 4, 2, 0, 3, 9, 8};
    sort(arr, 7);
    printArr(arr, 7);

    char arr2[] = {'a', 'A', 'F', 'b', 'd', 'c', 'B'};
    sort(arr2, 7);
}

```

```

    printArr(arr2, 7);
    return 0;
}

```

```

disen@qfxa:~/code2/day06$ ./a.out
0 1 2 3 4 8 9
A B F a b c d

```

### 3.3 函数模板和普通函数区别

函数模板不允许自动类型转化  
普通函数能够自动进行类型转化

### 3.4 函数模板和普通函数在一起调用规则

C++编译器优先考虑普通函数  
可以通过空模板实参列表的语法限定编译器只能通过模板匹配

```
printArr<>()
```

函数模板可以像普通函数那样可以被重载

```

swapF(T &a, T &b);
swapF(T1 &a, T2 &b);

```

如果函数模板可以产生一个更好的匹配，那么选择模板

```

#include <iostream>

using namespace std;

template <typename T>
void sort(T arr[], int size)
{
    for (int i = 0; i < size - 1; i++)
    {
        int min = i;
        for (int j = i + 1; j < size; j++)
        {
            if (arr[min] > arr[j])
                min = j;
        }
        if (min != i)
        {
            arr[i] = arr[i] ^ arr[min];
            arr[min] = arr[i] ^ arr[min];
            arr[i] = arr[i] ^ arr[min];
        }
    }
}

void sort(int arr[], int size)
{
    cout << "----普通函数----" << endl;
    for (int i = 0; i < size - 1; i++)
    {

```

```

        int min = i;
        for (int j = i + 1; j < size; j++)
        {
            if (arr[min] > arr[j])
                min = j;
        }
        if (min != i)
        {
            arr[i] = arr[i] ^ arr[min];
            arr[min] = arr[i] ^ arr[min];
            arr[i] = arr[i] ^ arr[min];
        }
    }
}

template <typename T>
void printArr(T arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    int arr[] = {1, 4, 2, 0, 3, 9, 8};
    sort<>(arr, 7); // 空模板
    printArr(arr, 7);

    char arr2[] = {'a', 'A', 'F', 'b', 'd', 'c', 'B'};
    sort(arr2, 7);
    printArr(arr2, 7);
    return 0;
}

```