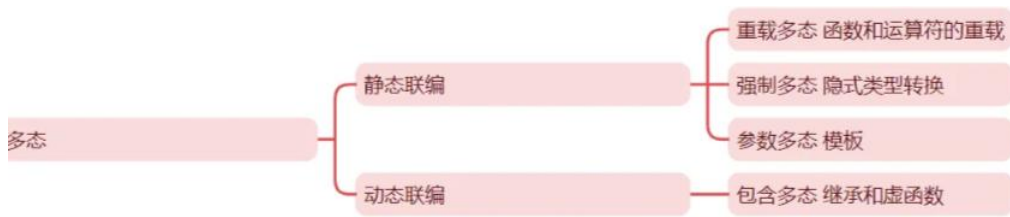


虚函数的“运行时多态”是靠“编译期生成的虚表 + 运行时的对象类型”共同实现的



运行多态必要条件：

- 必须通过基类的指针或者引用调用虚函数。
- 被调用的函数必须是虚函数，且派生类必须对基类的虚函数进行重写。

只有类的非静态成员函数前可以加 virtual，普通函数前不能加 virtual

内联函数不能是虚函数

构造函数、静态函数、拷贝构造函数不能是虚函数

若派生类中有一个和基类完全相同的虚函数(返回值类型相同、函数名相同以及参数列表完全相同)，此时我们称该派生类的虚函数重写了基类的虚函数。

虚函数的协变指的是基类与派生类虚函数的返回值类型不同——基类虚函数返回基类对象的指针或者引用，派生类虚函数返回派生类对象的指针或者引用

如果基类的析构函数为虚函数，此时派生类析构函数只要定义，无论是否加 virtual 关键字，都与基类的析构函数构成重写

重写 重载 重定义/同名隐藏

在虚函数的后面写上=0，则这个函数为纯虚函数

抽象类不能实例化出对象

只有重写纯虚函数，派生类才能实例化出对象

继承 实现继承 接口继承（虚函数）

有虚函数的类族的析构函数 一定是 虚析构函数

虚函数的形参默认值写在基类中

引用一旦初始化后，就无法重新赋值

virtual 只用于类中成员函数的声明，不能用在函数实现时。

派生类通过基类的成员函数调用虚函数时，将访问到派生类中的版本。

C++通过虚函数表来实现虚函数的动态绑定。

在有虚函数的类对象中，C++除了为它保存每个成员变量外，还保存了一个指向本类虚函数表的指针（vfptr）

纯虚函数在基类中声明后，不能在基类中定义函数体

dynamic_cast 成功条件：基类必须是**多态的**、只有当基类指针或引用**实际指向了一个派生类对象**时才会成功

typeid 在程序运行时判定一个表达式的**真实数据类型**

运算符重载实质是一个函数，是函数重载的特殊情况

针对 C++中原有运算符进行的，不可能通过重载创造出新的运算符

不得为重载的运算符函数设置默认值，调用时也就不得省略实参

除了 new 和 delete 这两个较为特殊运算符以外，任何运算符如果作为成员函数重载时，不得重载为静态函数

不能作为静态函数：构造与析构、友元函数、虚函数，其他的运算符重载

只能作为**成员函数重载**：**=、[]、()、->**

只能作为**非成员函数重载**：**输入输出流运算符**

不可重载的运算符 5 个（. * :: ? : sizeof） 点星域三大

重载赋值运算符的类，通常还需要定义什么函数？拷贝构造函数。

参数表

- 要改参数——传“引用 &”
- 只看参数，不改——传“const &”/传值
- 很小的简单数据（比如 int）——直接传值就好

需求	运算符应该写成……	理由
需要支持交换律（如 a + b 和 b + a）	非成员函数（友元函数）	因为左右两边都可能不是类对象，成员函数无法处理左侧非类对象
左边必须是类对象（比如 []、=）	成员函数	编译器只允许成员函数重载这些运算符

后置++/-- 比前置 形参表多一个 int

Test tmp(*this); 【虽然=也行，更推荐这种啦】

类型转换函数 只作为成员函数 定义不需要返回值

使用函数模板时，编译器一般不会进行类型转换操作

对于非模板函数和同名的函数模板，如果其他条件都相同，在调用时会**优先调用非模板函数**

对于类模板 成员函数若是放在类外定义，需要加模板参数列表

类模板不支持分离编译，即声明在 xxx.h 文件中，而定义却在 xxx.cpp 文件中

非类型模板参数只能使用常量，只允许使用**整型**家族，浮点数、类对象以及字符串是不允许作为非类型模板参数的。

非类型的模板参数在**编译期就需要确认结果**

定义模板时，不允许 template 语句与函数模板定义之间有任何其他语句

函数模板可重载

普通函数---模板/模板的重载---如果没有特别符合的，看特化

```
//如果是指针
// 情况1: 定义指针 (不会实例化)
Box<double>* ptr; // 只是声明一个指针, 不生成 Box<double> 的代码
// ptr 只是一个占位符, 指向内存地址, 此时不占用 Box 对象的空间

// 情况2: 指针指向对象 (触发实例化)
ptr = new Box<double>(3.14); // 此时才生成 Box<double> 的代码, 并创建对象
```

注意指针的类模板 类模板与友元函数

(1) 普通友元

```
template<class T>
class A
{
    friend class C1;
    friend void func1();
    //...
};
```

(2) 模板友元

作为友元的模板生成的所有实例都是类模板的友元

特点: 类模板的实例 一对多 友元

```
template<class type>
class A
{
    template<class T> friend class C2;

    template<class T> friend void func2(T u);
    //...
};
```

(3) 模板特例友元

特点: 类模板的实例 一对一 友元

- 指定为具体类型

```
C++  
template <class T> class c3; //声明类模板  
template <class T> void func3(T u); //声明函数模板  
  
template<class T>  
class A  
{  
    friend class c3<int>; //类模板的特例  
    friend void fun3<int>(int u); //函数模板特例  
    //...  
};
```

- 指定为类模板的类型参数

```
template <class T> class c3; //声明类模板  
template <class T> void func3(T u); //声明函数模板  
  
template<class T>  
class A  
{  
    friend class c3<T>; //类模板的特例  
    friend void fun3<T>(T u);  
    //...  
};
```

模板主要是 记住这么多的形式都是长什么样子的。。见

模板 不指名他究竟是什么类型

对比

函数模板

```
template<typename T>  
T Add(T a, T b)  
{  
    return a + b;  
}
```

类模板

```
template<class T>  
class AAA  
{  
private:  
    T a;  
    T b;  
    ...  
};
```

函数模板

隐式实例化 (不能隐式类型转换)

显式实例化

Add<int>(1, 2.5) --- 都变成int

或者你自己强制类型转换

Add<int>(1,(int)2.5)

类模板

特殊 如果成员函数定义在类外 要加上模板那个title

非参数类型

```
template<class T, int N>
```

当然更推荐 template<class T, size_t N>

```
main: AAA<int, 10>
```

两者都有的——模板的特化

函数模板的特化

可以用非模板函数替换。。

差别:

实例

```
int Add(const int& x, const int& y)
```

```
template<typename T>  
T Add(const T& x, const T& y) + Add<int>
```

特化--为了除了更加特殊的情况 比如加入指针

template<>

```
bool IsEqual<char*>(char* x, char* y)
```

```
bool IsEqual(char* x, char* y)
```

类模板的特化

全特化 所有参数确定化

```
template<class T1, class T2>
```

```
class Dragon
```

特化变成

template<>

```
class Dragon<double, int>
```

```
{
```

```
private:
```

```
    double _D1;
```

```
    int _D2;
```

```
};
```

偏特化

部分特化

```
template<class T2>
```

```
class Dragon<int, T2>
```

指针/引用

```
template<class T1, class T2>
```

```
class Dragon<T1*, T2*>
```

```
template<class T1, class T2>
```

```
class Dragon<T1&, T2&>
```

子类不会继承构造/析构函数、基类的私有成员访问权限、友元关系和静态初始化逻辑

关于不同继承方式继承之后剩下那些不同成员的访问限制符会怎么变。。。

公有继承 所有成员访问方式都不变

保护继承 公有 保护 变保护 私有不变

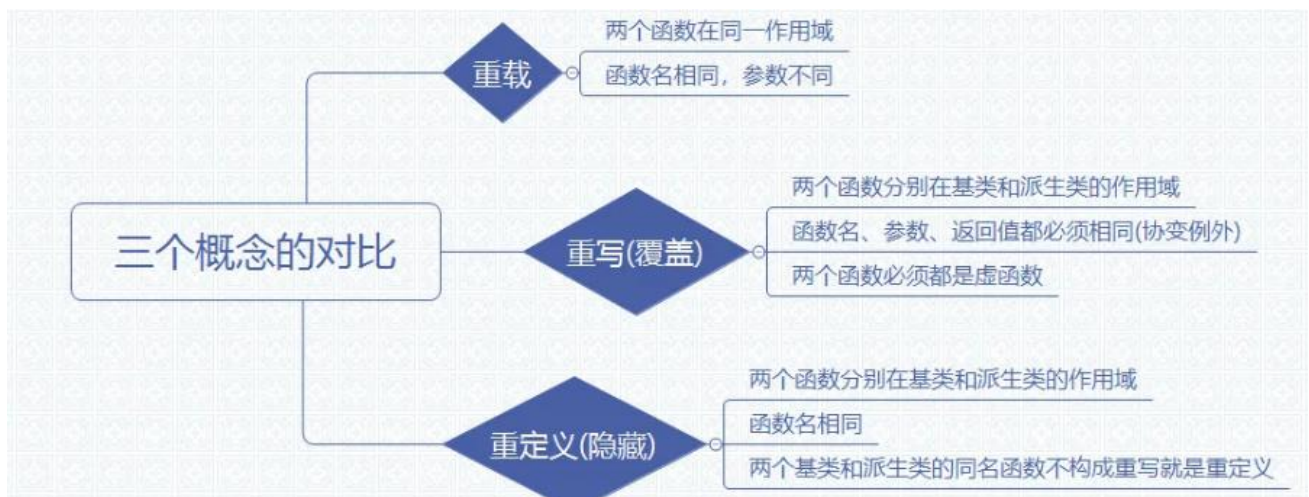
私有继承 全变私有

派生类对象可以赋值给基类的**对象**、基类的**指针**以及基类的引用——切片

基类对象不能赋值给派生类对象，基类的指针可以通过强制类型转换赋值给派生类的指针

若子类和父类中有同名成员，子类成员将屏蔽父类对同名成员的直接访问，这种情况叫**隐藏**，也叫重定义。

就是继承 fun 函数之后又在子类写了 fun 函数喵。



!!!!!!! 运算符重载与继承

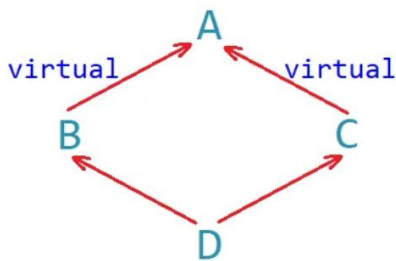
```
Student& operator=(const Student& s)
{
    cout << "Student& operator=(const Student& s)" << endl;
    if (this != &s)
    {
        Person::operator=(s);
        //调用基类的operator=完成基类成员的赋值--切片行为
        _id = s._id; //完成派生类成员的赋值
    }
    return *this;
}
```

子类的构造函数 拷贝构造函数 赋值运算符重载都需要调用父类相关的。

无论派生出多少个子类，都只有一个 **static 成员实例**。

菱形继承---数据冗余和

虚拟继承 虚基类 虚基表 虚基表指针



```
using namespace std;
class Person
{
public:
    string _name; //姓名
};
class Student : virtual public Person //虚拟继承
{
protected:
    int _num; //学号
};
class Teacher : virtual public Person //虚拟继承
{
protected:
    int _id; //职工编号
};
class Assistant : public Student, public Teacher
{
protected:
    string _majorCourse; //主修课程
};
```

B C 继承的时候前面加 virtual 采用虚基表指针和虚基表

A 虚基类 在 D 的时候只储存一份

引用在声明时必须初始化

引用的初始值可以是一个变量或另一个引用，且类型一致。

可以有指针的引用，不能有引用的指针

```
int b=1;
int* p=&b;
(int*)&rp=p; //rp是一个引用，它引用的是指针
(int*)&ra=a; //error
```

不能建立数组的引用，可以建立数组元素的引用

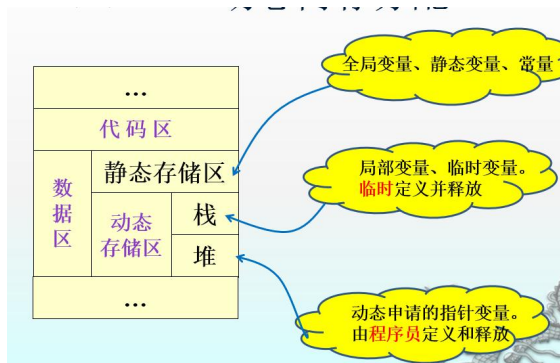
```
int a[10];
int& ra[10]=a; //error
int& ra=a[8]; //ok
```

缺省形参值与函数的调用位置

- ◆ 如果一个函数有原型声明，且原型声明在定义之前，则缺省形参值必须在函数原型声明中给出；
- ◆ 而如果只有函数的定义，或函数定义在前，则缺省形参值需在函数定义中给出。

```
int add(int x = 5, int y = 6);
//原型声明在前
int main()
{
    add();
}
int add(int x, int y)
{ //此处不能再指定缺省值
    return x + y;
}
```

```
int add(int x = 5, int y = 6)
{ //只有定义，没有原型声明
    return x + y;
}
int main()
{
    add();
}
```

内联函数与函数宏区别？

- 内联函数对函数的参数及返回值有明确定义，增强了代码的安全性。而宏定义没有。
- 内联函数的参数和返回值具有明确的类型标识，宏定义没有。

关键字 `inline` 必须与函数定义体放在一起才能使函数成为内联，仅将 `inline` 放在函数声明前面不起任何作用。

函数体相对简单、被频繁调用的函数适合定义为内联函数。

内联函数体内不能有循环语句和 `switch` 语句。

内联函数不能定义为递归函数。

内联函数不能有异常处理机制

`private` 是默认权限

成员数据不能是自身类的对象。此外，数据成员不能指定为自动 (`auto`)、寄存器 (`register`) 和外部 (`extern`) 存储类型

在类 `x` 的 **const 成员函数** 里，`this` 被设置成 `const X *` 类型

静态成员函数没有 `this` 指针

对象数组 `Clock clock[10];`

对象数组指针 `Clock * clock1 = new Clock[10]; + delete[] clock1;`

对象指针数组

`Clock * clock[10];`

`for(int i=0;i<10;i++)`

`clock[i] = new Clock;...`

`for(int i=0;i<10;i++)`

`delete clock[i];`

对象指针 用 `->` 访问对象成员

拷贝构造函数 是用来 **创建新对象** 的时候赋值，

赋值运算符重载 是用来 **给已存在的对象重新赋值** 的。

运算符重载实质是一个函数

多态 运行多态 编译多态

函数重载 模板

继承+虚函数

虚函数

构成条件：父类指针/引用调用虚函数+重写虚函数

重写 协变 析构函数的重写

Override final

纯虚函数 抽象类

实现继承 接口继承

虚表（虚函数表） 虚表指针

运行时类型信息 RTTI

typeid typeid(...).name == !=比较

运行时的数据 有无虚函数

类型转换

dynamic_cast 多态 动态 向上 向下（基类）

const_cast const 非 const

static_cast 关联类型转换 不安全

reinterpret_cast 任何 不安全

运算符重载 实质

实现方式：类的成员函数 友元函数

差别 this 指针有无 参数数目

分类 单目运算符(前置++ 后置++ 前置—后置--) 双目运算符(加减 赋值 复合**赋值** 比较运算符 下标运算符)【类之间喵】 特殊（输入输出、类型转换函数）

不可重载的运算符 5 个 (.,*::?: sizeof) 点星域三大

只能作为成员函数重载: =、[]、()、->

只能作为非成员函数重载: 输入输出流运算符

模板 不是函数 类型参数化

函数模板 类模板

函数: 隐式实例化, 显式实例化, (强制类型转换), 模板特化

类: 实例化, 类外定义、模板特化——全特化 所有参数确定/偏特化 部分参数确定 或者 指针/引用类型 、 非参数类型

构造函数由系统自动调用, 不能在程序中显式调用构造函数。

数组的部分初始化

`Point a2[3] = {Point(1, 2), Point(2, 3), Point(3, 4)};`
调用有参构造函数 3次

`Point a3[3] = {Point(1, 2), Point(2, 3)};`
调用有参构造函数 2次 调用无参构造函数1次

在用默认构造函数创建对象时, 如果创建的是全局对象或静态对象, 则对象所有数据成员初始化为 0;
如果创建的是局部非静态对象, 即不进行对象数据成员的初始化。

一旦定义了任何形式的构造函数, 系统就不再产生默认构造函数。

但为了系统工作, 有时候要写重载的无参构造函数 `point() { x = 0; y = 0; }`

缺省参数的构造函数与无参构造函数的冲突问题 !!!!!!!!!!!!!!!你二选一吧。

```
class X
{public:
    X() {}
    X(int i=0) { x = i; }
private:
    int x;
};
int main()
{ X one(12);
  X two;
  return 0;
}
```

解决方法1:
去掉无参构造函数

解决方法2:
去掉有参构造函数的默认值

调用:
X:X() ?
还是
X:X(int i=0) ?

通常可以利用 缺省参数将 所有类型的构造函数覆盖, 保证不出错。。大概

拷贝构造函数 形参表写 const。

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

默认构造函数 浅拷贝 遇见指针成员会出错 指针悬挂 需要自己构造深拷贝

构造函数初始化列表中的成员初始化次序与它们在类中的声明次序相同，与初始列表中的次序无关。

构造函数初始化列表先于构造函数体中的语句执行。

常量成员，引用成员，类对象成员，派生类构造函数对基类构造函数的调用必须采用初始化列表进行初始化。

不能重载：每个类仅有一个析构函数

类的前向引用声明


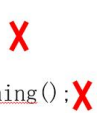
类应该先声明，后使用

如果需要在某个类的声明之前，引用该类，则应进行前向引用声明：

```
class B;
```

不能声明该类的对象，也不能在内联成员函数中使用该类的对象

◆ 应该记住：当你使用前向引用声明类B，而没有定义类B时，你只能使用被声明的符号，而不能涉及类的任何细节。

可以声明 B的形参、引用、指针	不可以 声明B的对象、调用B的行为
<pre>class B; //前向引用声明 class A { void f(B b); B& rb; B* pb; }; class B { };</pre> 	<pre>class B; //前向引用声明 class A { B m_b; void f(B b); { b.DoSomething(); } }; class B { };</pre> 

派生类不能继承基类以下内容：基类的构造函数和析构函数、基类的友元函数、静态成员数据和静态成员函数

类型兼容规则只针对于公有继承，在私有继承和保护继承中，不能将派生类对象赋值给基类对象

```

class Derived: public Base2,public Base1,public Base3
{public:
    Derived(int a,int b,int c,int d):Base1(a),_member2(d),_member1(c),Base2(b)
    { }
private:    //派生类的内嵌成员对象
    Base1 _member1;
    Base2 _member2;
    Base3 _member3;
};

```

```

constructing Base2 2
constructing Base1 1
constructing Base3 *
constructing Base1 3
constructing Base2 4
constructing Base3 *

```

因为基类 1 2 的构造函数有参，而 3 没参，所以需要显式定义构造

派生类构造顺序：先基类（按照继承顺序）再内嵌（按声明顺序）再自己（按自己顺序）

跟初始化列表顺序无关

派生类的生成过程

1. 吸收基类成员
2. 改造基类成员
 - 改变基类成员在派生类中的访问属性
 - 重写基类函数 → 同名隐藏
 - 重载基类函数
3. 添加新的成员

◆ 虚基类的成员是由最远派生类的构造函数通过调用虚基类的构造函数进行初始化的。

◆ 在整个继承结构中，直接或间接继承虚基类的所有派生类，都必须在构造函数的成员初始化表中给出对虚基类的构造函数的调用。如果未列出，则表示调用该虚基类的默认构造函数。

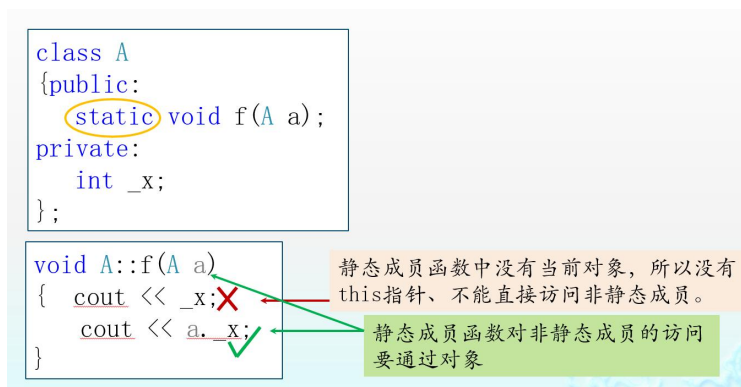
```

class Derived: public Base1, public Base2
{public:
    int _var;
    Derived(int var) : Base0(var), Base1(var), Base2(var)
    { }
    void fun()
    { cout << "Member of Derived" << endl; }
};

int main()
{ Derived d(1);
  d._var=2;
  d.fun();
  return 0;
}

```

在最远派生类的构造函数中，只有最远派生类的构造函数调用虚基类的构造函数，该派生类的其他基类对虚基类构造函数的调用被忽略



不使用对象时，静态成员函数只能访问静态成员数据或静态成员函数

不能定义成 `const` 函数，没有 `this` 指针

对 `public` 的静态成员的访问：可以通过类 `Point::showCount()`；也可以通过任何一个对象 `a.showCount()`；

静态成员数据不是由构造函数创建的，是由变量定义语句创建的：

友元关系是单向的不传递的不继承的

常数据成员以初始化列表的形式初始化

常成员函数：不更新对象的成员数据

!!!!!!! 常成员函数只能调用常成员函数

`const` 函数传递的是 `const this` 指针 —— `const` 可以用于函数重载

多文件组织

.h

```
#ifndef HEAD_H
```

```
#define HEAD_H
```

```
.....
```

```
#endif
```

.cpp 文件负责 `#include` 就行

条件编译指令 #elif

```
#if 常量表达式1
    程序正文1 //当“常量表达式1”非零时编译
#elif 常量表达式2
    程序正文2 //当“常量表达式2”非零时编译
#else
    程序正文3 //其他情况下编译
#endif
```

#ifdef 标识符

程序段1

#else

程序段2

#endif

如果“标识符”经#defined定义过，且未经undef删除，则编译程序段1，否则编译程序段2。

#ifndef 标识符

程序段1

#else

程序段2

#endif

如果“标识符”未被定义过，则编译程序段1，否则编译程序段2。