OOP语法

基本语法

auto/decltype推导返回类型

- auto：自动确定变量类型，追踪函数返回值类型
- decltype：对变量或表达式结果的类型进行推导，重用匿名类型
- auto+decltype：推导并追踪函数的返回值类型

```
auto func(char* ptr, int val) -> int;//在原本函数返回值的位置使用auto关键字

struct {} anon_s; //没有名字的结构体，定义了一个变量
decltype(anon_s) as ;  //定义了一个上面匿名的结构体...

auto func(int x, int y) -> decltype(x+y){
    return x + y;
}
```

友元

可对函数/class/struct/union进行友元声明，对其他类型的友元声明会被忽略，但不会报错。

友元不传递，不继承；不能定义新类.

内联函数

与宏定义函数相比，可**调试(类型检查,编译期错误检查)**，可**操纵私有成员**，

内联修饰用在函数定义的时候，而不是函数声明的时候。

定义在类声明中的函数**默认为内联函数。**一般构造函数、析构函数都被定义为内联函数。

new/delete操作符

```
A *pA = new A[3];//size: 4+3*sizeof(A),要多出4个字节来存放数组的大小
```

数据成员

数据成员类型	初始化方式
static	.h中声明，.cpp中初始化，否则将导致**链接错误**
const	**初始化列表/就地**，不得在构造函数体内初始化
const static	同static/就地(仅限int,enum)

自身类对象的引用和指针可以作为类的成员。

```
class A {
public:
    A* b = nullptr;
    A& a = *this;
};
```

类型转换

自动类型转换

```
class Src;
class Dst {
public:
  Dst(const Src& s);//1.在目标类中定义源类对象作参数的构造函数
  //explicit Dst(const Src& s);//禁止隐式自动类型转换
};
class Src {
public:
  operator Dst() const { return Dst();}//2.在源类中定义目标类型转换运算符
  //explicit operator Dst() const; //禁止隐式自动类型转换
};//只能选择1，2其中一种。
```

向上类型转换

继承中的向上类型转换，可由编译器自动完成，**只对public继承有效。**

当派生类的对象(**非指针或引用**)被转换为基类的对象时，派生类新数据丢失，发生**对象切片。**

其它

- dynamic_cast，动态类型转换，运行期间转换。如子类和父类之间的多态类型转换，即**向下类型转换**，被转换的类必须含有**虚函数。**若转换不安全，若被转换的是指针，则将返回空指针；如果被转换的是引用，则将抛出bad_cast异常 。
- static_cast，静态类型转换，编译时转换，可能不安全。也可执行**非多态的转换**，用于代替C中通常的类型转换操作。
- const_cast, 去除类型的const(或volatile)属性。

引用

- 右值：不能取地址的值。常见于**常值、函数返回值、表达式。**
- 左值：通常所说的引用。可以取地址的值。函数返回值可以是引用，但不得返回函数临时变量的引用。定义时进行初始化，不能修改引用指向。

```
int&& c = 1;//右值引用
const int& d = 1;//常量左值引用也能绑定右值

void f(int &x);
void f(int &&x) {//函数的右值形参在函数内部可以作为左值使用。
    f(x); //此时x为左值，右值延续了即将销毁变量的的生命周期。
}
```

右值引用可以转为常左值引用，**常左值引用不能转为右值引用。**

返回值优化(RVO)

将调用一个构造函数，一个拷贝构造函数，一个析构函数的代价降低到调用一个构造函数。

条件：

- return的值类型与函数签名的返回值类型相同；
- return的是一个局部对象。

构造与析构

```
ClassName();//默认构造函数
ClassName(const ClassName& obj);//拷贝构造函数
ClassName(ClassName&&);//移动构造函数
className(...);//其他普通构造函数

A a;//默认构造函数的调用
A a();//注意区别：调用函数运算符"()"，返回一个类型为A的对象

A b(a); //拷贝构造函数的调用:1.用一个类对象定义另一个新的类对象
A c = a;
c = a; //注意区别：调用赋值运算符。
Func(Test a); //2.函数调用时以类的对象为形参
Test Func(void); //3、函数返回类的对象



A d = std::move(a);//移动构造函数的调用
d = std::move(b);//注意区别：调用移动赋值运算符。
```

参数对象的构造和析构：

- 尽量使用对象引用作为参数。防止类内有指针数据成员时，两次释放同一内存。
- 函数体内的静态对象：第一次调用时初始化，离开作用域不析构，第二次及以后不再初始化.

在初始化列表中，如果使用委派构造函数，则不能初始化其他成员.

构造函数(默认、拷贝、移动)和析构函数，被编译器**需要**的时候，会自动合成。

继承和组合

继承基类构造函数

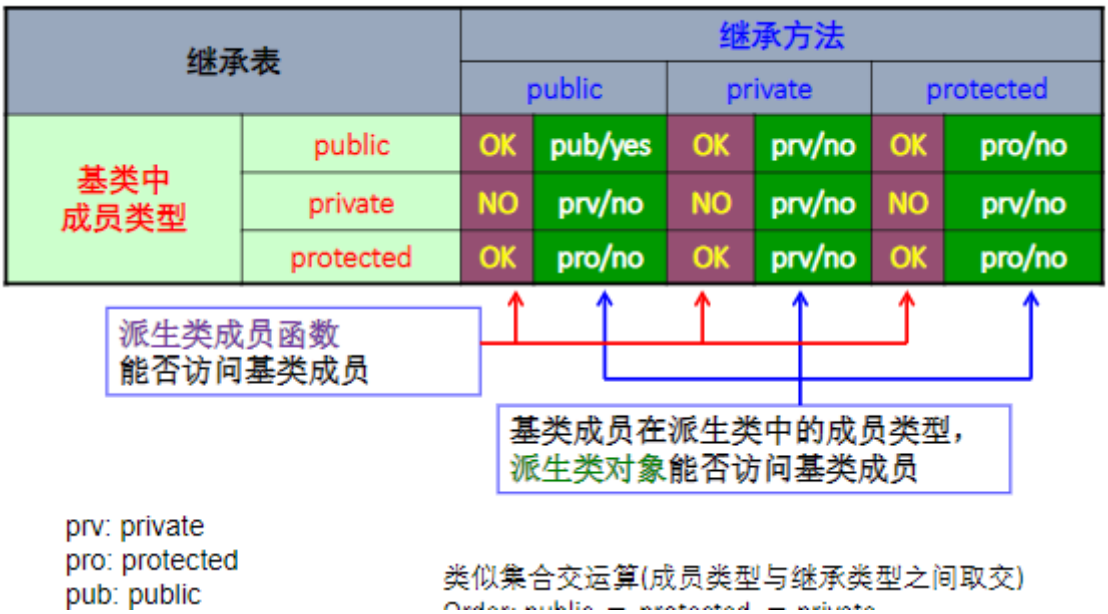
- 若子类构造函数中**无显示调用基类构造函数**，也没有定义基类的默认构造函数（基类也无自动生成的构造函数），则将导致**编译错误。**
- 显式调用基类构造函数，只能在派生类初始化列表中进行。
- using Base::Base继承基类构造函数。若派生类使用了私有构造函数，则不可以通过此种方式继承。

```
class Base {
public:
    Base(int i, int j=1) //基类构造函数的参数默认值不会被派生类继承
        { cout << "Base::Base(" << i << "," << j << ")\n"; }
};

class Derive : public Base {//默认为private继承
public:
    using Base::Base;   //相当于 Derive(int i):Base(i){};
                        // + Derive(int i, int j):Base(i, j){};
};
int main(){
    //Derive d1;  Derive array[10];   //没有默认构造函数，编译错误
    Derive obj(356);
    Derive obj2(356,789);
    return 0;
```

```
}
```

继承的访问权限



多重继承

钻石型继承树

- 数据冗余
- 二义性

多重继承会有多个虚函数表，几重继承，就会有几个虚函数表。这些表按照派生的顺序依次排列。

如果子类有新的虚函数，那么就添加到第一个虚函数表的末尾。

多态

- 静多态：编译器实现，包括重载、模板。
- 动多态：运行期实现，包括虚函数。

重载

同名函数在同一作用域下，至少一个参数类型不同或修饰符(const,static,etc.)不同，**返回值**不能作为区分，不能导致二义性。不能重载具有相同参数类型的静态与非静态成员函数。

运算符重载

```
istream& operator>> (istream& in, Test& dst) {
        return in;
}
ostream& operator<< (ostream& out, const Test& src) {
    return out;
}

int operator() (int a, int b); //重载()的对象也称函数对象
int& operator[] (const char* name); //注意：返回值如不是引用，则只能出现在等号左侧

ClassName operator++();//前缀自增
```

```
ClassName operator++(int dummy);//后缀自增，通过哑元参数区分前缀与后缀的同名重载

ClassName& operator= (const ClassName& right) {//赋值运算符
   if (this != &right) {}// 避免自己赋值给自己
   return *this;
}
ClassName& operator= (ClassName&& right); //移动赋值运算符
```

- 运算符“=、()、[ ]、->”只能作为类的非静态成员函数，不能是友元函数
- 所有重载为类的成员函数的运算符都可以用obj.operator(运算符)(参数)的形式调用。

```
class test{
public:
    int a = 0;
    int operator() (int a, int b);
    int& operator[] (const char* name);
    void operator--();
    void operator--(int dummy);
    istream& operator>> (istream& in);
    friend ostream& operator<< (ostream& in, const test& dst);
};
int main(){
    test a;
    a.operator--();//前缀自减
    a.operator--(0);//后缀自减
    a.operator()(1, 2);
    a.operator[]("");
    a.operator>>(cin);
    //a.operator<<(cout); //未声明未友元，不能以此种方式调用
    return 0;
}
```

重写隐藏

派生类中重写基类函数，导致基类所有重名函数被**隐藏**(仍可通过对象切片调用)。使用using base:: 恢复基类成员。

```
class B {
public:
  void f() { cout << "B::f()\n"; }
  void f(int i) { cout << "B::f(" << i << ")\n"; }
  void f(double d) { cout << "B::f(" << d << ")\n"; }
};
class D1 : public B {
public:
  void f(int i) { cout << "D1::f(" << i << ")\n"; }
  using B::f;
};
int main() {
  D1 d;
  d.f(10); //D1::f(10)
  d.f(4.9); //若不用using，此处将输出D1::f(4)，现在输出B::f(4.9)
  d.f(); //若不用using，将编译出错，现在输出B::f()
  return 0;
}
```

虚函数

对于被派生类重新定义(函数名，**参数类型相同**，返回值可以不同，即**重写覆盖**)的成员函数，若它在基类中被声明为虚函数，则通过基类**指针或引用**调用该成员函数时，编译器将根据所指（或引用）对象的实际类型决定是调用基类中的函数，还是调用派生类重写的函数.

基类中被声明为虚函数的函数，无论在派生类中是否声明为虚函数，仍然为虚函数。

重写隐藏时，虚函数指针不会发生覆盖；重写覆盖时，派生类的虚函数表中原基类的虚函数指针会被重新定义的虚函数指针覆盖掉。

原理

- 编译期间：建立虚函数表VTABLE，记录**本类**或**基类**中所有已声明的虚函数入口地址。
- 运行期间：在构造函数中建立虚函数指针VPTR，指向相应的VTABLE。与没有虚函数的类相比，有虚函数的类中多了一个指针(void*)。

构造函数不能是虚的，析构函数应当是虚的。虚函数在构造，析构函数中无效。

静态方法不能是虚函数。

override和final关键字

- override关键字判断重写覆盖是否正确。
- final指定虚函数不可被重写或类不可被继承。

纯虚函数与抽象类

定义了纯虚函数的类是抽象类，后者不允许定义对象。

抽象类只提供接口，能避免对象切片：保证只有指针和引用能被向上类型转换。

继承一个抽象类时，必须重写覆盖所有纯虚函数。纯虚析构函数除外：编译器可以自动生成派生类的默认析构函数，不必显式实现。

纯虚析构函数必须有函数体，一般的纯虚函数也可以有函数体。

模板

只能定义在.h文件中。

函数模板

```
template <typename T> ReturnType Func(Parameters) {};

template <typename T> T sum(T a, T b) { return a + b; }
sum(9,1);
//sum(9, 2.1); //编译错误，无法自动类型推导
sum<int>(9, 2.1)
```

类模板

```
  template<typename T, unsigned size>//模板参数
  class array {
      T elems[size];
      ... };
  array<char, 10> array0; //必须显式指定类型

  class normal_class {//普通类的成员函数，也可以定义为模板函数
```

```
 public:
     int value;
     template<typename T> void set(T const& v) {
         value = int(v);
   }     /// 在类内定义
     template<typename T> T get();
 };
 template<typename T>        /// 在类外定义
 T normal_class::get() {
   return T(value);
 }

template<typename T0> class A { //普通模板类的成员函数，也可有额外的模板参数
    T0 value;
public:
    template<typename T1> void set(T1 const& v){
        value = T0(v);
    }         /// 在类内定义
    template<typename T1> T1 get();
};
template<typename T0> template<typename T1>
T1 A<T0>::get(){ return T1(value);}  // 类外定义
//注意！不要写成template<typename T0, typename T1>
```

- 函数模板**不能**部分特化，可以用重载替代。
- 类模板可以部分特化。

设计模式

行为型模式

以最少的代码变动完成功能的增减。

模板方法（Template Method）

开放封闭原则

对扩展开放，对修改封闭。

核心就是在结构层面上解耦，对抽象进行编程，而不对具体编程。

策略（Strategy）

单一责任原则

一个类（接口）只负责一项职责，不要存在多于一个导致类变更的原因。

核心：在功能上解耦。

迭代器（Iterator）

分离底层数据结构和上层算法。

结构型模式

在结构上尽可能的解耦合。

适配器(Adapter)

变换接口

- 组合实现，即**对象适配器**.
- 继承实现，即**类的适配器**. 继承Adaptee的接口(private)和Target的实现(public)

代理/委托(Proxy)

分割访问对象与被访问对象以减少耦合，并在中间增加各种控制功能。

一方面提供被代理类所有**接口**，另一方面可以进行额外的**控制**操作。

装饰器(Decorator)

创建了一个装饰类，用来包装原有的类，并在保持类方法完整性的前提下，提供了额外的功能。 且**装饰类与被包装的类继承于同一基类**，这样装饰之后的类可以被再次包装并赋予更多功能。

 1561345597755

创建型模式

以简短的代码完成对象的高效创建。

单例（Singleton)

构造函数、析构函数设为private，不对外开放；显式删除拷贝构造函数和赋值运算符。

通过一个静态方法或枚举返回单例对象。

惰性初始化：将单例作为静态函数内的静态变量。

CRTP(模板): 在基类中构造单例，派生类中完成功能。

工厂

构造单个类的对象。

抽象工厂

以特定的组合方式构造多个类的对象，