深蓝学院
shenlanxueyuan.com

C++ 基础

第 12 章：类进阶

主讲人 李伟

微软高级工程师

《C++ 模板元编程实战》作者

目录

运算符重载

```
auto operator + (Str x, Str y) {
  Str z;
  z.val = x.val + y.val;
  return z;
}
```

```
//operator()可以有缺省参数
struct Str {
  int val = 3;
  auto operator () (int y=3) {
    return val + y; }};
int main() {
  x()//6;
}
```

- 使用 operator 关键字引入重载函数
  - 重载不能发明新的运算，不能改变运算的优先级与结合性，通常不改变运算含义
  - 函数参数个数与运算操作数个数相同，至少一个为类类型
  - 除 operator() 外其它运算符不能有缺省参数

```
auto operator + (Str x, Str y = Str{}) {
  Str res;
  res.val = x.val - y.val;
  return res;}
//error,不能有缺省参数
```

  - 可以选择实现为成员函数与非成员函数
    - 通常来说，实现为成员函数会以 *this 作为第一个操作数（注意 == 与 <=> 的重载）
- 根据重载特性，可以将运算符进一步划分（参考资料）：
  - 可重载且必须实现为成员函数的运算符（=,[],(),-> 与转型运算符）
  - 可重载且可以实现为非成员函数的运算符
  - 可重载但不建议重载的运算符（ &&, ||, 逗号运算符）

这些符号的特点：1.基本上是二元运算符，2.左右操作数的顺序是有规定的。

    - C++17 中规定了相应的求值顺序但没有方式实现短路逻辑

这种规定的求值顺序（函数输入的两个参数，将会先算第一个参数，再算第二个参数）只对&&，||和逗号运算符

  - 不可重载的运算符（如？： 运算符）

不止这个运算符，还有别的运算符

运算符重载——重载详述 1

+ - * /，老师个人理解：接收的2个参数可以互换。

- 对称运算符通常定义为非成员函数以支持首个操作数的类型转换
- 移位运算符一定要定义为非成员函数，因为其首个操作数类型为流类型
- 赋值运算符也可以接收一般参数
- operator [] 通常返回引用 模拟数组索引操作
- 自增、自减运算符的前缀、后缀重载方法
- 使用解引用运算符（ * ）与成员访问运算符（ -> ）模拟指针行为
  - 注意“ .” 运算符不能重载
  - “→” 会递归调用“→”操作
- 使用函数调用运算符构造可调用对象

如果能使用前缀自增，就不要使用后缀自增。从这个例子也可以看出来，后缀自增会构造一个新的对象，比较浪费空间和时间

s++；1.首先构造一个副本，然后把自身相加，把副本返回

```
//移位运算符一定要定义为非成员函数
auto& operator << (std::ostream& ostr, Str input) {
  ostr << input.val;
  return ostr;}//要返回ostr,因为要支持串起来的操作
```

```
//赋值预算夫也可以接受一般参数
Str& operator= (const Str& input) {
  val = input.val;
  return *this; }
Str& operator= (const std::string& input) {
  val = static_cast<int>(input.size());
  return *this;}
int val;
```

```
struct Str {
  Str(int x): val(x) {}
  auto operator + (Str input) {
    return Str(val + input.val);  }
//但是由于通常来说数据都是private类型，因此我们应该定义一个友元函数，使其operator+可以访问私有变量
friend auto operator + (Str input1, Str input2);//可以把定义也移到struct内部，这样这个友元函数在全局就不能访问，必须通过adl（实参依赖查找）
//运算符重载就是友元函数在类内定义的典型应用
private:
  int val; };
//非成员函数版
auto operator + (Str input1, Str input2) {
 return Str(input1.val + input2.val);}
int main() {
  Str x = 3;
  Str y = 4;
  Str z = x + 4;//这个也会调用那个加法运算，4会被隐式转换为str，但是如果写Str z = 4+x;就会报错。因此，对称运算符通常定义为非成员函数
}
  int& operator[] (int id) {
    return val;}
  //不带引用会导致x[0]只能读，不能被赋值
  int operator[] (int id) const {
    return val;}
  //引入重载，方便const类型的对象调用这个函数.因为对于const的类型，我们是不会修改他的值的，所以就没有返回引用
  const Str cx;
  std::cout << cv[0];
  Str& operator++() {++val; return *this;}
  //如果括号里是空的，对应前缀自增
  Str operator++(int) {Str tmp(*this};
  ++val; return tmp;
  //对应后缀自增，但是不建议大家使用这个int参数
```

```
//使用*, -> 模拟指针行为
struct Str {
  Str(int* p) : ptr(p){}
  int& operator *() {
  //解引用操作符
  return *ptr;
  }
  int operator *() const{
  //解引用操作符，只读
  return *ptr;
  }
  Str* operator -> () {
    return *this;
  }

  int val = 5;
private:
  int* ptr;
};

int main() {
  int x = 100;
  Str ptr(&x);
  std::cout << *ptr << std::endl;
  *ptr = 101; //就是因为返回了引用，
才能被赋值为101

  int x = 1000;
  Str ptr(&x);
  std::cout << ptr->val;//可以放到c++
insight来看一下c++是如何处理的
//(ptr.operator->()->val)
//当编译期看到了ptr->,同时ptr是一个类型
的对象，他就会尝试找->的重载。如果他能
找到这个重载，同时这个重载返回一个对象
指针，那么他就会使用这个指针来调用val
}
```

左边是比较简单的一种->的重载，返回的是结构体的指针。还可以返回另一个结构体

```
->会递归调用->
struct Str2 {
  Str2* operator->() {
    return this;}
  int blabla = 123;
};
struct Str {
  Str(int* p) : ptr(p) {}
  Str2 operator -> () {
    return Str2{}; }
  int val = 5;
private:
  int* ptr;
};
int main() {
  int x = 1000;
  Str ptr(&x);
  ptr->blabla;
// c++insight
//ptr.operator->().operator->()->blabal
}
```

函数调用运算符结构不定个数的参数（唯一一个）

c++引入函数构造运算符就是为了构造可调用对象

```
struct Str {
  Str(int p): val(p) {}
  int operator() () //函数调用运算符 {
    return val;
  }
  int operator() (int x, int y) {
    return val + x + y; }
  bool operator(int inpu) {
    return val < input;
  //好处：相比于单独的function来说，这种方
式使用起来更加灵活
//缺点：为了引入灵活性，不得不引入更多的代
码。因此我们后来引入了lambda表达式。本质上
就是对于这个用法的简化
  }
private:
  int val;
};
int main() {
  Str obj(100);
  obj();
  obj(1,2);
}
```

运算符重载——重载详述 2

- 类型转换运算符
  - 函数声明为 operator type() const　和普通函数区别：没有返回类型
  - 与单参数构造函数一样，都引入了一种类型转换方式
  - 注意避免引入歧义性与意料之外的行为
    - 通过 explicit 引入显式类型转换
  - explicit bool 的特殊性：用于条件表达式时会进行隐式类型转换
- C++ 20 中对 == 与 <=> 的重载　在c++20之前，你需要写6个重载，>,<.=,>=,<=,因此c++20进行了改进
  - 通过 == 定义 !=
  - 通过 <=> 定义多种比较逻辑
  - 隐式交换操作数
  - 注意 <=> 可返回的类型： strong_ordering, week_ordering, partial_ordering

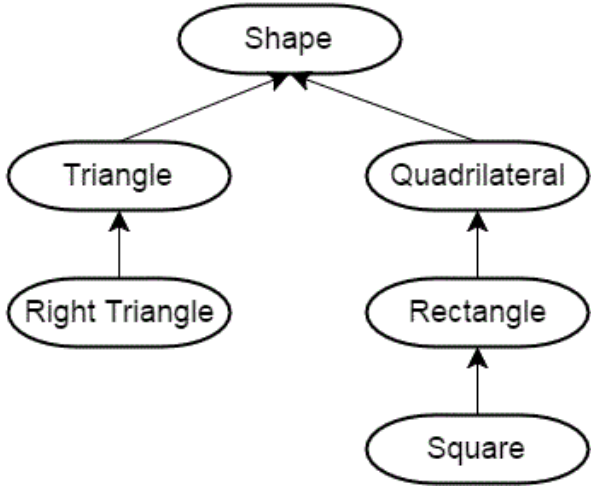
```
//类型转换运算符
struct Str {
  Str(int p): val(p) {}
  operator int() const {
    return val;
  }

private:
  int val;
};
int main() {
  Str obj(100);
  int v = obj;//调用Int()
}
```

类的继承

class缺省情况下采用private继承

- 通过类的继承（派生）来引入“是一个”的关系（参考图片来源）
  - 通常采用 public 继承（ struct V.S. class ）
  - 注意：继承部分不是类的声明
  - 使用基类的指针或引用可以指向派生类对象
  - 静态类型 V.S. 动态类型
  - protected 限定符：派生类可访问
- 类的派生会形成嵌套域
  - 派生类所在域位于基类内部
  - 派生类中的名称定义会覆盖基类
  - 使用域操作符显式访问基类成员 Base::val
  - 在派生类中调用基类的构造函数

得使用初始化列表来调用基类的构造函数



```
//使用基类的指针或引用可以指向派生类对象
class Derive: public Base {
  void fun2() {}
};

int main() {
  Derive d;
  Base& ref = d;
  Base* ptr = &d;//ref的动态类型是Derive
  ref.fun2();//编译器会报错
}
```

静态类型：编译期就可以确定的类型.静态类型是说给编译器听的
如果你使用了一个操作，静态类型不支持的话，编译就会报错
动态类型：在运行期为对象实际赋予的类型

```
//派生类中调用基类的构造函数
struct Base{
  Base() {
    std::cout << "Base Constructor is called.";
  }
};

struct Derive: public Base {
public:
  Derive() {
    std::cout << "Derive Constructor is called.";
  }
};
int main() {
  Derive d; // 两个构造函数都会被调用，Base先被调用
Derive 再被调用
}
```

```
//派生类中名称定义会覆盖基类
struct Base {
  int val = 2;
};
class Derive: public Base {
public:
  void fun2() {
    std::cout << val << std::endl;
//  int val = 3;
};
int main() {
  Derived d;
  d.val; //如果Derive里面有val就会优先寻找Derived中的
val，如果Derived中没有val，就会寻找Base中的val
```

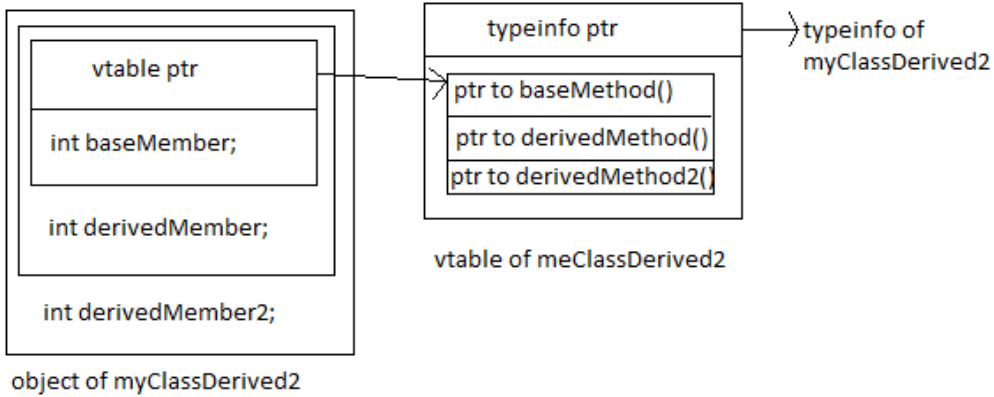
Derive 会调用Base的构造函数，但是缺省的情况下，
只会调用Base缺省的构造函数。
因此我们需要在Derived中显示的调用构造函数

```
struce Base {
  Base() {
    std::cout << "Base constructor is called.";
  }
};
class Derive: public Base {
public:
  Derive(int a) : Base(a) {
    //不能在这里直接写Base::Base(a); 因为构造函数是分两个部分
一个是初始化，一个是赋值.当执行到这句话的时候，初始化已经完成
了。在这里只是做一些赋值的工作，改变一些值。因此不能在这里调用。
应该使用初始化列表
    std::cout << "";
  }
};
int amin() {
  Derived d(1);
}
```

类的继承——虚函数

- 通过虚函数与引用（指针）实现动态绑定
  - 使用关键字 virtual 引入
  - 非静态、非构造函数可声明为虚函数
  - 虚函数会引入vtable结构
    - dynamic_cast

      转换是在运行期发生的，并且很耗时



- 虚函数在基类中的定义
  - 引入缺省逻辑
  - 可以通过 = 0 声明纯虚函数，相应地构造抽象基类
- 虚函数在派生类中的重写（override）
  - 函数签名保持不变（返回类型可以是原始返回指针 / 引用类型的派生指针 / 引用类型）
  - 虚函数特性保持不变
  - override 关键字

在什么情况下我们会为一个纯虚函数给出定义：（使用较少）
我们定义一个纯虚函数，就是因为我们不知道它的缺省的实现是什么样子的。但是虽然我们不知道缺省是如何实现的，但是我们可能知道它的缺省实现会包含一些辅助逻辑，这些辅助逻辑应该放到实现的开头去调用的。因此我们可以把这些辅助逻辑引入纯虚函数的定义中。

```
struct Base {
  virtual void baseMethod() {}
  int baseMember;
};
class myClassDerived: public Base {
  virtual void derivedMethod() {}
  int deriveMember;
} ;
struct myClassDerived2: public myClassDerived{
  virtual void derivedMethod2() {}
  int derivedMember2;
};
int main() {
  myClassDerived2 d;
  Base& b = d;//从派生类转化为基类直接转化是可以的
  //但是从基类转化为派生类就需要引入dynamic_cast
  myClassDerived2& d2 = dynamic_cast<myClassDerived2&>(b);
  //可以转化就是因为有typeinfo这个东西，typeinfo是从vtable
引入的，vtable是从virtual引入的
}
```

我们可以引入override关键字来保证重写。如果不是
重写，编译器就会报错。方便编译器检查,和自己调试

虚函数好处：
1.会引入动态类型叫typeinfo,就会引入dynamic_cast的动态类型转换

```
struct Base {
  virtual void fun() {
    std::cout << "Base::fun()" << std::endl;
  }
  int baseMember;
};
struct Derive: Base {
  void fun() {
    std::cout << "derived fun()" << std::endl;
  }
};
//虚函数使用方法
void proc(Base& b) {
  b.fun();
}
int main() {
  Derive d;
  d.fun();
  proc(d); // 合法,动态多态，因为是在运行期实现的多态

  Base& b= d;
  b.fun(); // derived::fun()被打印

}
```

类的继承——虚函数续

- 由虚函数所引入的动态绑定属于运行期行为，与编译期行为有所区别
  - 虚函数的缺省实参只会考虑静态类型
  - 虚函数的调用成本高于非虚函数　java所有都是虚函数。。。
    - final 关键字　告诉编译器，我这个函数，我接下来会有一些类派生于这个类，但是所有的新类都不会对这个函数有改变
  如果有一个派生类所有的虚函数都是final的，那么意味着我这个类不会有别的类去派生了。我们可以在类定义的时候加ginal. struct Derive final : Base {};
  - 为什么要使用指针（或引用）引入动态绑定
  - 在构造函数中调用虚函数要小心
  - 派生类的析构函数会隐式调用基类的析构函数
  - 通常来说要将基类的析构函数声明为 virtual 的
  - 在派生类中修改虚函数的访问权限

```
在Base中虚函数是protected的，在derived类中
虚函数是public类型的。
访问权限是编译期行为。
int main() {
  Derive d;
  d.fun(); //可以
  Base& b = d;
  b.fun();//不可以，编译失败
}
```

```
  struct Derive: Base {
    void fun(int x = 4) override final {
      std::cout << "Derive: " << x << std::endl;
    }
  };
```

为什么使用指针或者引用的时候才会引入动态编译：
因为指针和引用都不会实际构造一个base对象（如果你去看汇编的话，引用实际上也是通过指针来构造的），在使用指针的时候，我们没有需要把derived变成base的这个过程。因此才只能使用指针或者引用引入动态绑定

在构造函数中引入虚函数要小心,如果在基类构造函数中调用虚函数，那么一定不会调用派生类的函数

```
struct Base {
  Base {
    fun();
  }
  virtual void fun() {
    std::cout << "Base: " << std::endl;
  }
};
struct Derive final : Base {
  Derive() : Base() {fun();}
  void fun() override {
    std::cout << "Derive: " << std::endl;
  }
};
int main() {
  Derive d;//打印出来的是Base，后打印出来derived
}
```

```
//派生类的析构函数
struct Base {
  ~Base() {
    std::cout << "Base" << std::end;
  }
};
struct Derive final: Base {
  ~Derive() {
    std::cout << "Derive" << std::endl;
  }
};
int main() {
  Derived* d = new Derive();
  Base* b = d;//c++标准说这样，你的程序行为是未定义的,大概率只会输出Base.
```

类的继承——继承与特殊成员函数

- 派生类合成的……
  - 缺省构造函数会隐式调用基类的缺省构造函数
  - 拷贝构造函数将隐式调用基类的拷贝构造函数
  - 赋值函数将隐式调用基类的赋值函数
- 派生类的析构函数会调用基类的析构函数
- 派生类的其它构造函数将隐式调用基类的缺省构造函数
- 所有的特殊成员函数在显式定义时都可能需要显式调用基类相关成员
- 构造与销毁顺序
  - 基类的构造函数会先调用，之后才涉及到派生类中数据成员的构造
  - 派生类中的数据成员会被先销毁，之后才涉及到基类的析构函数调用

类的继承——补充知识

- public 与 private 继承（参考资料）
  - public 继承：描述“是一个”的关系
  - private 继承：描述“根据基类实现出”的关系
  - protected 继承：几乎不会使用
- using 与继承
  - 使用 using 改变基类成员的访问权限
    - 派生类可以访问该成员
    - 无法改变构造函数的访问权限
  - 使用 using 继承基类的构造函数逻辑
  - using 与部分重写
- 继承与友元：友元关系无法继承，但基类的友元可以访问派生类中基类的相关成员

类的继承——补充知识续

- 通过基类指针实现在容器中保存不同类型对象
- 多重继承与虚继承
- 空基类优化与[[no_unique_address]]属性

深蓝学院
shenlanxueyuan.com

感谢聆听！

Thanks for Listening