




复旦大学




面向对象程序语言C++ 继承和动态绑定

周雅倩
zhouyaqian@fudan.edu.cn
2016/5/24




复旦大学媒体计算研究所

问题




- 学生成绩管理问题
- 变化
 - 学生可以选择研究生或者本科生学分的课程。
 - 要想获得研究生的学分，学生就要做一些额外的工作。
 - 假定所有学生除了要完成家庭作业和考试外，研究生还需要写一篇论文。

2016/5/24




复旦大学媒体计算研究所

目标




- 编写一个新的类来满足这些新的要求
- 也希望在第9章中完成的解决方案可以继续使用

2016/5/24




复旦大学媒体计算研究所

继承




- 继承是面向对象程序设计的基石之一
- 基本思想：一个类和另一个类很象，只是多了一些扩展。

2016/5/24




复旦大学媒体计算研究所

设计




- 我们需要定义两个类：一个类表示核心的要求，另一个类表示研究生学分的要求。
- 第一个类：Student_info→Core
- 第二个类：Grad

2016/5/24



复旦大学媒体计算研究所

class Core



```
class Core{
public:
    Core();
    Core(std::istream&);
    double grade() const;
    std::istream & read(std::istream &);
    std::string name() const{return n;}
private:
    std::istream & read_common(std::istream&);
    std::string n;
    double midterm, final;
    std::vector<double> homework;
};
```

2016/5/24

class Grad

```
class Grad: public Core{
public:
    Grad();
    Grad(std::istream&);
    double grade() const;
    std::istream & read(std::istream&);
private:
    double thesis;
};
```

2016/5/24

说明-1

- Grad从Core派生而来（或继承而来）。
- Core是Grad的基类。
- Core的每个成员也是Grad的成员
 - 构造函数、赋值操作符和析构函数除外
- Grad类
 - 可以添加它自己的成员：thesis, Grad构造函数
 - 也可以重定义基类的成员：grade, read

2016/5/24

说明-2

- public Core中的public说明Grad从Core继承而来的是接口的一部分，而不是实现的部分。
 - Core的公有成员也会成为Grad的公有成员。

2016/5/24

保护标签

```
class Core{
public:
    Core();
    Core(std::istream&);
    double grade() const;
    std::istream & read(std::istream&);
    bool valid() const {return homework.empty();}
    std::string name() const {return n;}
protected:
    std::istream & read_common(std::istream&);
    double midterm, final;
    std::vector<double> homework;
private:
    std::string n;
};
```

2016/5/24

说明

- 由于Grad中的成员函数无法访问Core的私有成员，使用protect保护标签来重写Core类。
- protect保护标签使派生类可以访问其基类对象的保护成员，但用户对这些成员没有访问权限。
- private成员只有类本身的成员和它的友元才能访问
- public成员所有用户都可以访问，包括其派生类。

2016/5/24

操作-Core

```
std::string Core::name() const {return n;}

double Core::grade() const
{
    return ::grade(midterm, final, homework);
}

istream & Core::read_common(std::istream &in)
{
    in >> n >> midterm >> final;
    return in;
}

istream & Core::read(std::istream &in)
{
    read_common(in);
    read_hw(in, homework);
    return in;
}
```

2016/5/24

操作-Grad

```
double Grad::grade() const
{
    return _cpp_min(Core::grade(),thesis);
}

istream & Grad::read(std::istream &in)
{
    read_common(in);
    in >> thesis;
    read_hw(in, homework);
    return in;
}
```

2016/5/24

派生类对象的构造过程

- 为整个对象分配空间（包括基类成员和派生类成员）
- 调用基类的构造函数来初始化对象中的基类部分
- 使用构造函数初始化列表直接初始化派生类的成员
- 执行派生类构造函数体中语句

2016/5/24

构造函数-Core

```
class Core{
public:
    Core(): midterm(0), final(0){}
    Core(std::istream & is){read(is);}
    .....
};
```

2016/5/24

构造函数-Grad

```
class Grad: public Core{
public:
    Grad():thesis(0){}
    Grad(std::istream&is){read(is);}
    .....
};
```

2016/5/24

多态和虚函数-问题

```
vector<Core> students;
sort(students.begin(),students.end(),compare);

Grad g(cin);
Grad g2(cin);

Core c(cin);
Core c2(cin);

compare(g, g2);
compare(c, c2);
compare(g, c);
```

2016/5/24

说明

- 我们之所以可以把一个Grad对象传递给需要Core&参数的函数，是因为Grad是继承自Core类的，所以每个Grad对象都有一个Core类的部分。

2016/5/24

不知道对象类型的情况下取得对象的值

```
bool compare_grade(const Core &x, const Core &y)
{
    return x.grade() < y.grade();
}
```

- 问题：如何在不知道对象类型的情况下取得对象的值？
- 解决方法：使用虚函数

```
class Core{
public:
    virtual double grade() const;
};
```

2016/5/24

说明

- 把grade声明为虚函数，当调用compare_grades函数时，系统会通过查看引用c1和c2实际绑定的类型，来确定执行哪个类的grade函数。
- 关键字virtual用在基类定义中。
- 虚函数会继承。

2016/5/24

动态绑定

- 动态绑定：虚函数在运行时，通过引用或者指针来调用这个函数。

```
bool compare_grade(const Core &x, const Core &y)
{
    return x.grade() < y.grade();
}
```

- 静态绑定：通过一个对象来调用一个虚函数，可以在编译时知道这个对象的准确类型。

```
bool compare_grade(const Core x, const Core y)
{
    return x.grade() < y.grade();
}
```

2016/5/24

更多例子

```
Core c;
Grad g;
Core * p=&g;
Core &r=g;
```

```
c.grade();//静态绑定
g.grade();//静态绑定
p->grade();//动态绑定
r.grade();//动态绑定
```

2016/5/24

多态

- 多态：我们可以在指向基类的指针或引用的地方使用派生类类型。
- C++是通过虚函数的动态绑定特性来支持多态的。
- 我们可以通过一个单独的函数来调用多个类型中的某个函数。

2016/5/24

使用继承来解决我们的问题

```
int main()
{
    vector<Core> students;
    Core* record;
    char ch;
    string::size_type maxlen=0;
    while(cin >> ch){
        if(ch == 'U') record= new Core;
        else record= new Grad;
        record->read(cin);
        if(record->name().size() > maxlen) maxlen= record->name().size();
        students.push_back(record);
    }
    sort(students.begin(), students.end(), compare_Core_ptr);
    for(std::vector<Core>::iterator iter=students.begin(); iter!=students.end(); iter++){
        cout << (*iter)->name() << "stringmaxlen=" << maxlen << "iter->name().size()=" <<
            (*iter)->name().size() << endl;
        double final_grade = (*iter)->grade();
        streamsize prec=out.precision(3);
        cout << "Your final grade is " << setprecision(3) <<
            final_grade << setprecision(prec) << endl;
    }
    catch(domain_error e){
        cout << e.what();
    }
    delete *iter;
}
```

2016/5/24

虚析构造函数

- 执行 `delete students[i]` 时，系统会根据 `students[i]` 实际指向的类型来调用相应的析构函数。

```
class Core{
public:
    virtual ~Core(){}
    .....
};
```

2016/5/24

一个简单的句柄类

```
class Student_info{
public:
    Student_info():cp(0){}
    Student_info(std::istream & is):cp(0){read(is);}
    Student_info(const Student_info&);
    Student_info & operator=(const Student_info&);
    ~Student_info(){delete cp;}
    //operations
    std::istream & read(std::istream &);
    std::string name() const{
        if(cp)return cp->name();
        else throw std::runtime_error("uninitialized Student");
    }
    double grade() const{
        if(cp) return cp->grade();
        else throw std::runtime_error("uninitialized Student");
    }
    static bool compare(const Student_info& s1, const Student_info& s2){
        return s1.name() < s2.name();
    }
private:
    Core * cp;
};
```

Core对象指针初始化

静态成员函数，不在类的一个具体对象上进行操作，所以这里比较函数有两个参数

2016/5/24

读取句柄

```
istream & Student_info::read(istream & is){
    delete cp;
    char ch;
    is >> ch;
    if(ch == 'U')
        cp = new Core(is);
    else
        cp = new Grad(is);
    return is;
}
```

2016/5/24

复制句柄对象

```
class Core{
    friend class Student_info;
protected:
    virtual Core * clone()const{
        return new Core(*this);
    }
    //as before
};

class Grad{
    //friend class Student_info;
protected:
    Grad * clone()const{
        return new Grad(*this);
    }
    //as before
};

Student_info::Student_info(const Student_info& s):cp(0){
    if(s.cp) cp = s.cp->clone();
}

Student_info& Student_info::operator=(const Student_info& s){
    if(&s != this){
        delete cp;
        if(s.cp) cp = s.cp->clone();
        else cp = 0;
    }
    return *this;
}
```

2016/5/24

使用句柄类

```
int main()
{
    vector<Student_info> students;
    Student_info record;
    char ch;
    string::size_type m_axlen = 0;
    while(!record.eof()){
        if(record.dname().size() > m_axlen) m_axlen = record.dname().size();
        students.push_back(record);
    }
    sort(students.begin(), students.end(), Student_info::compare);
    for(std::vector<Student_info>::iterator iter = students.begin(); iter != students.end(); iter++){
        cout << iter->name() << string(m_axlen + 1 - iter->name().size(), ' ');
        try{
            double final_grade = iter->grade();
            streamsize prec = out.precision(3);
            cout << "Your final grade is " << setprecision(3) << final_grade << setprecision(prec) << endl;
        }catch(domain_error e){
            cout << e.what();
        }
    }
    return 0;
}
```

2016/5/24

纯虚函数和抽象类-1

- 在许多场合，基类中的虚函数不给出有意义的代码，而只仅仅是提供一个接口，具体实现留给它的派生类，这时可将它说明成纯虚函数。
- 说明纯虚函数的一般形式是：

```
class 类名 {
    virtual 类型 纯虚函数名 (形参列表) = 0;
}
```
- 有纯虚函数的类称为抽象类。由于抽象类还有许多不完全在地方，不能定义抽象类的对象。

2016/5/24

纯虚函数和抽象类-2

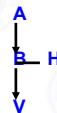
- 也有这样的情况，基类抽象类给出了所有可能的纯虚函数，而具体的某个派生类可能不具有某纯虚函数对应的接口，为此允许派生类不给出某些基类纯虚函数的实现。
- 反之，派生类也可定义新的纯虚函数，或对基类的纯虚函数再作纯虚函数定义，而使派生类也是抽象类。
- 抽象类除可用作基类之外，另一个主要用途是用作指针或引用的基类型。
- 定义抽象类的意义是能使进入继承层次的每个类都具有纯虚函数所表示的行为，能有效提高继承层次各类的概念清晰。

2016/5/24

三种访问控制方式

公有继承(public)、私有继承(private)和受保护继承(protected)。

设类A为基类，类B是类A的派生类，另有建立B类对象的程序模块H，以及类B的派生类V，并设H不是类A的直接或间接派生类。现在考虑因类B对类A的不同继承方式，H与V对类A公有成员的可访问性。



2016/5/24

三种访问控制方式

- 公有继承**
 - 当B类是通过公有继承从A类派生时，H和V可以不受限制地访问A类的公有成员。
- 私有继承**
 - 当B类是通过私有继承从A类派生时，H和V不可以访问A类的公有成员。
- 受保护继承**
 - 当B类是通过受保护继承从A类派生时，H不可以访问A类的公有成员，而V可以不受限制地访问A类的公有成员。

2016/5/24

访问控制

私有：只在类内可访问

公有：类内外都可访问

受保护：派生类中等价于公有，派生类外等价于私有

		派生类B	B的派生类V	类外函数H
公有继承	公	√	√	√
公有继承	私	×	×	×
公有继承	受保护	√	√	×
私有继承	公	√	×	×
私有继承	私	×	×	×
私有继承	受保护	√	×	×
受保护继承	公	√	√	×
受保护继承	私	×	×	×
受保护继承	受保护	√	√	×

2016/5/24