第13章　使用继承和动态绑定

刘　卉

huiliu@fudan.edu.cn

前 言

- 成绩问题
  - 通用课程：研究生 & 本科生
  - 研究生：除了完成作业和考试外，还要写一篇论文.
- 编写一个新类满足新的要求
  - 之前完成的解决方案继续使用→新类使用之前的代码读取/计算/输出成绩.

13.1 继承 (Inheritance)

- 继承是OOP的基石之一
  - 基本思想：A类与B类很相似，只是多了一些扩展.
- 定义三个类

1）满足核心课程要求的学生：与Student_info相似，重命名为Core.

2）满足研究生课程的要求：Grad类，从Core继承而来.

3）Student_info：句柄类，表示所有学生.

确保资源的合适调度和利用

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

```
class Grad: public Core {
public:
  Grad();
  Grad(std::istream&);
  double grade() const;
  std::istream& read(std::istream&);
private:
  double thesis;  // 增加派生类自己的数据成员
};
```

Grad从Core派生(继承)而来
→Core是Grad的基类

重载基类成员

- 继承：Core的成员也是Grad的成员(构造函数、赋值操作符和析构函数除外).
- Grad可添加自己的成员，重定义基类成员，但不能删除基类成员.

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

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

- public Core: Core的接口也是Grad的接口.
- Grad对象有5个数据成员.
- Grad对象有2个构造函数和4个成员函数.

- public Core: Grad继承了Core的公有接口，使其成为自身的公有接口.

```
Grad g;
cout << g.name() //直接调用继承而来的公有接口
```

- Grad对象有5个数据元素：其中4个继承自Core，thesis是新添加的.
- Grad对象有2个构造函数和4个成员函数：其中2个成员函数重定义了Core中对应的成员，另外2个直接从Core继承.

13.1.1 操作

- Core的操作

```
string Core::name() const { return n; }
double Core::grade() const
{ return ::grade(midterm, final, homework); }
istream& Core::read_common(istream& in)
{
    // read and store the student's name and exam grades
    in >> n >> midterm >> final;
    return in;
}
```

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

- Grad的操作

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

禁止访问

禁止访问

派生类也不能访问基类的私有成员

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

- 使用生存空间操作符：调用基类的grade函数计算不包含论文的成绩.

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

- 递归调用

13.1.2 保护标签

- 派生类的成员函数无法访问基类的私有成员
  - 只有基类本身和它的友元才能访问.

    e.g. Grad的成员函数read无法访问Core的4个数据成员和read_common函数;
  - 解决方法：使用保护标签.

```
class Core {
public:
   Core();
   Core(std::istream& is);
   std::string name() const;
   double grade() const;
   std::istream& read(std::istream&);
protected:
   // accessible to derived classes
   std::istream& read_common(std::istream&);
   double midterm, final;
   std::vector<double> homework;
private:
   // accessible only to Core
   std::string n;
};
```

派生类可以访问基类对象的保护成员，但基类用户不能访问.

Grad类不能直接访问n，只能通过Core类的访问器函数访问.

- 派生类可以访问基类对象的保护成员，但基类的用户不能访问

```
istream& Grad::read(istream& in)
{
   read_common(in);
   in >> thesis;
   read_hw(in, homework);
   return in;
}
Core c;
c.read_common(in);
cout << c.midterm << c.final << endl;
```

无需特别说明，即可访问基类成员，因为它们也是Grad的成员.

基类用户不能访问基类的保护成员.

13.1.3 继承和构造函数

- 派生类对象的构造方式

  step1. 为整个对象分配空间(包括基类成员和派生类成员);

  step2. 调用基类的构造函数，初始化对象的基类部分;

  step3. 使用构造函数初始化列表直接初始化派生类的成员;

  step4. 执行派生类构造函数体中的语句.

如何选择基类的构造函数

1）使用构造函数初始化列表指定.

```
class A{
public:
    A(): a(0), b(0){}
    A(int _a, int _b): a(_a), b(_b){}
private:
    int a, b;
};
```

```
class B: public A{
public:
    B(): c(0){}
    B(int _a, int _b, int _c):
        A(_a, _b), c(_c){}
private:
    int c;
};
```

```
A a1, a2(1, 2);
B b1, b2(1, 2, 3);
```

2）若未指定，使用默认构造函数构造对象的基类部分.

```
class Core {
public:
   Core(): midterm(0), final(0) { }   // default constructor for Core
   Core(std::istream& is){ read(is); }   // build a Core from an istream
   // ...
};
class Grad: public Core {
public:
   // both constructors implicitly use Core::Core() to initialize the base part
   Grad(): thesis(0) { }
   Grad(std::istream& is) { read(is); }
   // ...
};
```

Grad g;

step1. 系统分配足够空间保存Grad的5个数据成员;

step2. 执行Core的默认构造函数，初始化g的Core部分数据成员;

step3. 执行Grad()的初始化列表，将thesis初始化为0;

step4. 执行Grad的默认构造函数的函数体(空).

Grad g(cin);

step1~2. 同上;

step3. 执行Grad(istream&)的初始化列表(无)，将thesis初始化(值取决于g是全局/局部变量);

step4. 执行Grad(istream&)的函数体，为5个数据成员读取值.

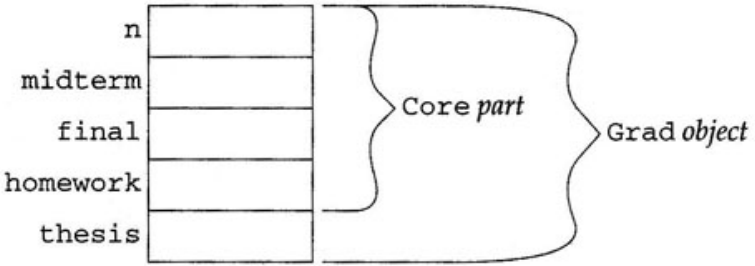
13.2 多态和虚函数

- 使用compare函数比较两条学生记录

```
bool compare(const Core& c1, const Core& c2)
{ return c1.name() < c2.name(); }
Grad g(cin);   // read a Grad record
Grad g2(cin);  // read a Grad record
Core c(cin);   // read a Core record
Core c2(cin);  // read a Core record
compare(g, g2);   // compare two Grad records
compare(c, c2);   // compare two Core records
compare(g, c);    // compare a Grad record with a Core record
```

把Grad对象传递给需要Core&参数的函数→每个Grad对象都有一个Core类部分

静态绑定



- 把compare函数的const Core&参数绑定到Grad对象的Core部分

```
bool compare(const Core* c1, const Core* c2)
bool compare(const Core c1, const Core c2)
```

也可以用Grad对象调用上述函数.

13.2.1 不知道对象类型的情况下获得对象的值

- 按最终成绩排序生成列表

```
bool compare_grades(const Core& c1, const Core& c2)
{  return c1.grade() < c2.grade(); } //始终调用Core类的grade函数
```

  - 对象的类型在运行时才知道，如何根据对象的实际类型，调用对应的grade函数?

- 虚函数(Virtual function)
  - 支持运行时选取.

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

- 调用compare_grades函数时，系统会查看c1和c2实际绑定的对象类型，确定执行哪个类的grade函数.
- 关键字virtual只用在类定义中→Core::grade()的定义不需要改变.
- 派生类继承了虚函数→Grad类中声明grade函数时，无需使用virtual关键字.

13.2.2 动态绑定

- 必须以引用/指针调用虚函数
  - 才能实现运行时选取.
  - 基类的引用/指针可以引用或指向基类/派生类的对象.

静态绑定——编译时绑定

```
// incorrect implementation!
bool compare_grades(Core c1, Core c2)
{ return c1.grade() < c2.grade(); }
```

- 调用时，即使实参是Grad对象，也会被裁剪为它的Core部分，绑定到Core::grade上.
- 用对象调用虚函数→对象的类型不可能在执行过程中发生变化→静态绑定.

- 动态绑定——运行时绑定
  - 通过引用/指针调用虚函数→引用/指针所绑定对象的类型，决定了调用哪个版本的虚函数.

```
Core c;   Grad g;
Core* p;  Core& r = g;
c.grade();  // statically bound to Core::grade()
g.grade();  // statically bound to Grad::grade()
p->grade(); // dynamically bound, depending on the type of
            // the object to which p points
r.grade();  // dynamically bound, depending on the type of
            // the object to which r refers
```

多态(polymophism)

- 一个类型可代表多种类型.
- 在需要基类指针/引用的地方，使用派生类类型.
- C++支持多态：通过虚函数的动态绑定特性.
- 引用/指针的类型确定，但它们引用/指向的对象的类型不确定→通过单个类型调用多个函数.

虚函数必须定义

- 非虚函数可以只声明，不定义，只要程序不调用.
- 如果没有定义虚函数，编译器就会报告奇怪的错误.
- 纯虚函数：没有函数体，而是“= 0”，表示该虚函数在此处没有定义.

```
class shape_base{
public:
   virtual double area() const = 0;
};
```

抽象基类

- shape_base仅作为具体图形的共同基类而存在→没有shape_base类对象.
- 纯虚函数使得shape_base类成为抽象基类→不能创建抽象基类的对象.
- 抽象基类：在继承层次上提供抽象接口.

抽象基类的派生类

- 纯虚函数被派生类继承
  - 如果派生类定义了它所继承的全部纯虚函数→成为具体类→可以创建派生类对象.
  - 否则，该派生类仍然是抽象类.

13.2.3 回顾

- 把read函数也声明为虚函数
  - 根据对象的类型决定执行哪个read函数—读入混合数据

```
class Core {
public:
  Core(): midterm(0), final(0) { }
  Core(std::istream& is) { read(is); }
  std::string name() const;
  virtual std::istream& read(std::istream&);
  virtual double grade() const;
```

```
protected:
   // accessible to derived classes
   std::istream& read_common(std::istream&);
   double midterm, final;
   std::vector<double> homework;
private:
   // accessible only to Core
   std::string n;
};
```

```
class Grad: public Core {
public:
   Grad(): thesis(0) { }
   Grad(std::istream& is) { read(is); }
   // grade and read are virtual by inheritance
   double grade() const;
   std::istream& read(std::istream&);
private:
   double thesis;
};
bool compare(const Core&, const Core&);  // 全局函数
```

13.3 使用继承解决问题

- 定义vector保存所读取的全部记录

```
vector<Core> students;
// must hold Core objects, not polymorphic types
```

- 定义局部变量保存每次读取的记录

```
Core record; // Core object, not a type derived from Core
```

只能处理Core类记录

- 如何读取包含Core/Grad混合记录的文件?
  - 消除所有的类型依赖
  - 两种方法

方法1: 包含(实质上)未知类型的容器

- 容器元素为基类指针(可指向派生类对象)
  - 缺点：把复杂的工作强加给用户→为每条记录分配/回收空间.
  - 编写新的比较函数，以指针作形参.

```
bool compare(const Core& c1, const Core& c2)
{  return c1.name() < c2.name();  }
bool compare_Core_ptrs(const Core* cp1, const Core* cp2)
{  return compare(*cp1, *cp2);  }
```

  - 区分不同记录：研究生以G开头，本科生以U开头.

```
// this code almost works
int main() {
    vector<Core*> students; // store pointers, not objects
    Core* record; // temporary must be a pointer as well
    char ch;
    string::size_type maxlen = 0;
    // read and store the data
    while (cin >> ch) {
        if (ch=='U')  record = new Core; // allocate a Core object
        else  record = new Grad; // allocate a Grad object
        record->read(cin); // virtual call
        maxlen = max(maxlen, record->name().size()); // dereference
        students.push_back(record);
    }
    // pass the version of compare that works on pointers
    sort(students.begin(), students.end(), compare_Core_ptrs);
```

```
    // write the names and grades
    for(vector<Core*>::size_type i=0; i!=students.size(); ++i) {
        cout << students[i]->name()
             << string(maxlen+1-students[i]->name().size(), ' ');
        try {
            double final_grade = students[i]->grade();   // virtual call
            streamsize prec = cout.precision();
            cout << setprecision(3) << final_grade <<
                    setprecision(prec) << endl;
        } catch (domain_error e) {
            cout << e.what() << endl;
        }
        delete students[i];
        // free the object allocated when reading
    }
    return 0;
}
```

13.3.2 虚析构函数

- delete students[i]
  1）自动调用对象的析构函数；
  2）包含该对象的空间被释放.

```
class Core {
public:
    virtual ~Core() { }
    // as before
};
```

- 在输出循环中逐个删除对象时
  - delete core*⇨调用Core类的析构函数⇨析构Core类成员，即便该指针实际指向Grad对象.
- 解决方法——把析构函数声明为虚函数

虚析构函数

```
delete students[i];
```

- 系统会根据student[i]实际指向的对象类型，调用相应的析构函数.
- 析构函数体为空→系统自动销毁类成员.
- 用基类指针销毁派生类对象时→虚析构函数.
- 毋需为Grad类增加析构函数→继承了析构函数的虚函数性质(像其它虚函数一样).

```
class Core {
public:
    virtual ~Core() { }
    // as before
};
```

virtual destructor

- As with all virtual functions, the fact that the destructor is virtual is inherited. Because neither class has any explicit work to do in order to destroy objects, there is no need to redefine the destructor in the derived class.
- Because the derived class *inherits the virtual property* of its base-class destructor, all we have to do is recompile the program.

virtual destructor

- Like other virtual functions, *the virtual nature* of the destructor is inherited. Therefore, if the destructor in the root class of the hierarchy is virtual, then the derived destructors will be virtual as well. A derived destructor will be virtual whether the class explicitly defines its destructor or uses the synthesized destructor.
- Destructors for base classes are *an important exception* to the Rule of Three. If a base class has an empty destructor in order to make it virtual, then the fact that the class has a destructor is not an indication that the assignment operator or copy constructor is also needed.
- The root class of an inheritance hierarchy *should* define a virtual destructor even if the destructor has no work to do.

P496, C++ Primer, Fourth Edition

方法2: 简单的句柄类 (handle class)

- 直接使用指针的缺点
  - 需完成很多与指针相关的复杂工作→存在错误隐患.
- 句柄类 确保资源的合适调度和利用
  - 编写一个新类Student_info，封装指向Core对象的指针→把隐患隐藏起来.
  - Student_info对象既可以表示Core对象，也可以表示Grad对象→就像一个指针.

- Student_info的用户无需为其绑定的底层对象分配/释放空间
- Student_info类：处理程序中烦琐和有错误隐患的地方.

```
class Student_info {
private:
  Core* cp;
public:
  // constructors and copy control
  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(); // virtual call
      else throw std::runtime_error("uninitialized Student");
    }
    static bool compare(const Student_info& s1,
                        const Student_info& s2)
    {  return s1.name() < s2.name();  }
};
```

```
istream& Student_info::read(istream& is)
{
   delete cp;  //delete previous object, if any
   char ch;
   is >> ch;  //get record type
   if (ch == 'U')
      cp = new Core(is); //分配空间,且调用Core::Core(is)⇨Core::read(is)
   else
      cp = new Grad(is); //分配空间,且调用Grad::Grad(is)⇨Grad::read(is)
   return is;
}
```

首先释放先前绑定的对象

```
vector<Student_info> students;
Student_info record;
while (record.read(cin))  //可以读取任何一种记录
   students.push_back(record);
```

```
~Student_info() { delete cp; }
```

- Core有虚析构函数→Student_info的析构函数能正确地操作.

```
std::string name() const {
   if (cp) return cp->name();
  else throw std::runtime_error("uninitialized Student");
}
```

- 用户使用Student_info对象编写程序→Student_info类必须提供与Core类相同的接口.
- 函数name()和grade(): 除了通过cp调用底层对象的相应函数之外，还需处理cp为0的情况.

Why Student_info::compare?

```
class Student_info{
   ……
     static bool compare(const Student_info& s1, const Student_info& s2)
     { return s1.name() < s2.name(); }
};
```

需要能比较Student_info对象的compare函数

1. `vector<Student_info> students;`
   `sort(students.begin(), students.end(), Student_info::compare);`
2. Why not global?——在core.h中已经定义了全局的compare函数
   `sort(students.begin(), students.end(), compare);`

调用哪个呢?

静态成员函数compare

```
static bool compare(const Student_info& s1, const Student_info& s2)
    { return s1.name() < s2.name(); }
```

- 静态成员：与类而非某个特定对象相关.
- 没有对象与该函数相关→函数体不能访问非静态数据成员.

静态数据成员

- 一个类的所有对象需要共享数据

1．用全局变量表示共享数据

- 违背数据抽象与封装原则，数据缺乏保护．

2．静态数据成员

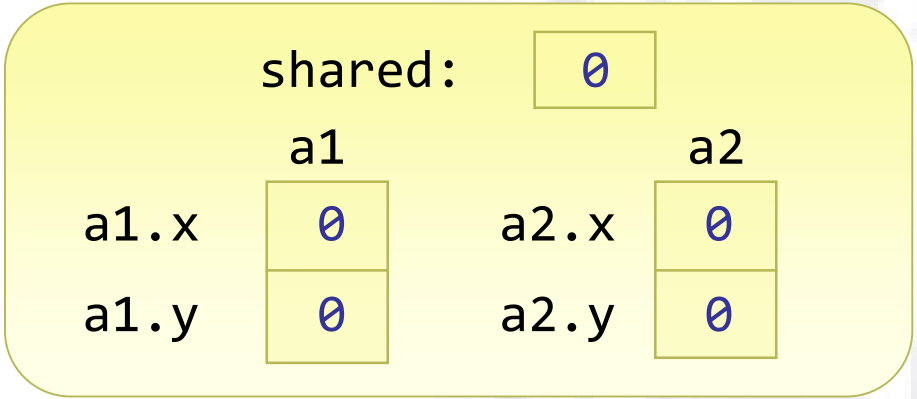
- 为同类对象的数据共享提供了更好的途径．

- 静态数据成员的说明和定义

```
class A {
public:
  A():x(0), y(0) {}
  void increase_all() { x++; y++; shared++; }
  int sum_all() const { return x+y+shared; }
  static int get_shared()  //静态成员函数
  { return shared; }
private:
  int x, y;
  static int shared;  //静态数据成员声明⇨在类定义中声明
};
int A::shared = 0;  //静态数据成员的定义⇨在类实现中定义
```

- 类的静态数据成员对该类的所有对象只有一个拷贝

```
A a1, a2;
```




```
a1.increase_all();  // a1.x, a1.y, A::shared变为1
cout << a2.get_shared() << ', ' << a2.sum_all() << endl;
// 输出: 1, 1
```

注意事项

- 成员函数仅涉及对静态数据成员的访问→将其声明为静态成员函数.
- 静态成员可通过对象访问，也可直接通过类访问

```
sort(students.begin(), students.end(), Student_info::compare);
sort(students.begin(), students.end(), students[0].compare);
```

13.4.2 复制句柄对象

- copy constructor和operator= 复制哪种对象?
  - 事先并不知道Student_info对象所绑定对象的类型.
  - 为Core及其派生类提供一个虚函数: 创建对象, 并使其值为原对象值的副本.

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

类A声明类B为友元→类A的所有成员都成为类B的友元.

重新定义派生类

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

- 基类虚函数返回指向基类对象的指针/引用时，派生类函数可以返回指向派生类对象的指针/引用.
- 毋需将Student_info定义为Grad的友元→Student_info不会直接调用Grad::clone.

拷贝构造函数和赋值操作符

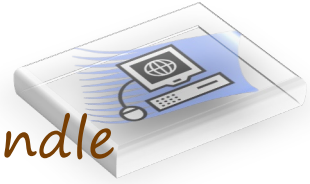
```
Student_info::Student_info(const Student_info& s): cp(0)
{  if (s.cp)  cp = s.cp->clone();  //virtual call  }
Student_info& Student_info::operator=(const Student_info& s)
{
   if (&s != this) {
      delete cp;
      if (s.cp)  cp = s.cp->clone();  //virtual call
      else  cp = 0;
   }
   return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

13.5 使用句柄类

```
int main()
{
   vector<Student_info> students;
   Student_info record;
   string::size_type maxlen = 0;
   //read and store the data
   while (record.read(cin)) {   //可以读取任何一种记录
         maxlen = max(maxlen, record.name().size());
         students.push_back(record);
   }
   //alphabetize the student records
  sort(students.begin(), students.end(), Student_info::compare);
```

```
    //write the names and grades
    for (vector<Student_info>::size_type i = 0;
          i != students.size(); ++i) {
          cout << students[i].name() << string(maxlen + 1 -
                  students[i].name.size(), ' ');
          try {
            double final_grade = students[i].grade();
            streamsize prec = cout.precision();
            cout << setprecision(3) << final_grade
                 << setprecision(prec) << endl;
          } catch (domain_error e) {
            cout << e.what() << endl;
          }
    }
    return 0;
}
```

与之前没有Core类和Grad类的主函数几乎一样



13.6 精妙之处

- 继承和容器

```
vector<Core> students; //容器students只能保存Core对象
Grad g(cin); //read a Grad
students.push_back(g); //store the Core part(!) of g in students
```

  - push_back函数的参数类型是const Core&→可以把g传递给该函数；
  - 添加时，将构造一个Core类型匿名对象，把g的Core部分复制传入该对象，忽略g的Grad部分.

- 我们需要哪个函数?
  - 当基类和派生类有同名函数，但参数个数/类型不同→不同函数.

e.g. 为Core和Grad添加修改成绩的函数

```
void Core::regrade(double d) { final = d; }
void Grad::regrade(double d1, double d2)
{ final = d1; thesis = d2; }
```

如果有Core& r;

```
r.regrade(100);  // ok, call Core::regrade
r.regrade(100, 100);  // compile error,
// Core::regrade takes a single argument
```

如果有Grad& r;

```
r.regrade(100);  // compile error
// Grad::regrade takes two arguments
r.regrade(100, 100);  // ok, call Grad::regrade
```

- 如果想执行基类版本，必须显式调用

```
r.Core::regrade(100);  // ok, call Core::regrade
```

- 如果regrade作为虚函数，则基类和派生类必须提供相同的接口.

```
virtual void Core::regrade(double d, double = 0) { final = d; }
```

小 结

- 继承

```
class base {
public:  //common interface
protected:
  //implementation members accessible to derived classes
private: //implementation accessible to only the base class
};
//public interface of base is part of the interface for derived
class derived: public base { ... };
```

  - 派生类可重新定义基类操作，亦可添加自己的成员.

- 类也可以私有继承

```
class priv_derived: private base { ... };
```

该用法很少，只是为了方便实现才使用.

- 需要基类对象/引用/指针时，皆可使用派生类对象.
- 继承链可以有几层深度

```
class derived2: public derived { ... };
```

derived2对象有derived类和base类对象的属性.

- 派生类对象构造时：分配足够空间→构造基类部分→构造派生类部分．析构时，先析构派生类部分，再析构基类部分.

动态绑定

- 运行时，根据对象的实际类型，决定执行哪个函数.
- 通过指针/引用调用虚函数.
- 基类的虚函数在派生类中不需要重复关键字virtual.
- 派生类不一定重定义虚函数，虚函数首次出现必须定义.
- 纯虚函数→抽象类

- 覆盖
  - 派生类与基类有相同的成员函数(参数个数和类型，返回值类型都相同).
  - 如果参数列表不匹配，函数就是无关的.
- 虚析构函数
  - 指向基类的指针被用来删除派生类对象时，基类就需要一个虚析构函数.

- 构造函数和虚函数
  - 在类的构造函数中，使用指针/引用调用类的虚成员函数→静态绑定.

  调用基类构造函数时，派生类对象尚未构造完成，只能绑定基类成员函数.
- 类的友元关系
  - 一个类可以把另一个类声明为它的友元.
  - 友元不能继承，也不能传递.

- 类的静态成员
  - 不是类对象的成员.
  - this在静态成员函数中不可用.
  - 静态成员函数只能访问静态数据成员.
  - 每个静态数据成员只有一个单独的实例，必须在'类.cpp'源文件中初始化.

```
value_type class_name::static_member_name = value;
```