

Practice of Programming 0 Overview

Haopeng Chen

***RE**liable, **IN**telligent and **Scalable** Systems Group (**REINS**)*

Shanghai Jiao Tong University

Shanghai, China

e-mail: chen-hp@sjtu.edu.cn

- 《程序设计课程设计》
 - 软件学院必修课程，面向本科一年级的学生
- 本课程将以C++为程序设计语言
 - 巩固和扩展《程序设计1》中的知识点
 - 通过实践训练学生面向对象编程的技能
 - 为《程序设计2》做铺垫和准备

- 学生
 - 软件学院本科一年级
- 教师
 - 陈昊鹏 13917262561 chen-hp@sjtu.edu.cn, 软件大楼1111
- 助教
 - 刘宇宸 13916376016 liuyuchen@sjtu.edu.cn, 软件大楼3407
 - 张钢 13917062387 infear@163.com, 软件大楼5403
 - 戴文韵 13671700142 scorpiodwy@163.com, 软件大楼5403
 - 梅诚 13661876152 cheermc@gmail.com, 软件大楼5403
 - 沈雨茜 13817715524 cece0205@gmail.com, 软件大楼5403

- 上课时间&地点

- 第19、20、21周 周一、三、五 上午 1-2 节, E2-102

- 上机地点

- 软件大楼 5号楼机房
- 5110379001~5110379025 /5120/刘宇宸
- 5110379026~5110379050 /5120/张钢
- 5110379051~5110379070 /5103/戴文韵
- 5110379071~5110379090 /5103/梅诚
- 5110379091~5110379104及其他学号段 (约9名) /5120/沈雨茜

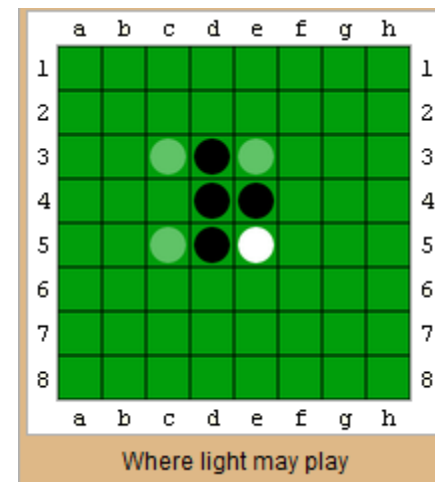
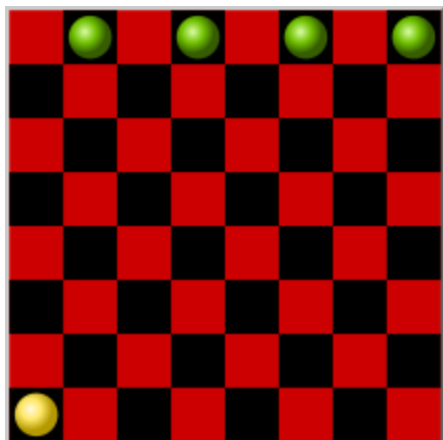
- 答疑时间&地点

- 周一、周三、周五上午10:00-12:00 教师办公室
- 平时有问题也可以通过邮件和教师或助教联系

- 第 1 周：问题抽象与分解
 - 类的设计
 - 子类化的运用、虚函数与继承的运用
 - 接口与实现分离原则的运用、纯虚函数与抽象类的运用
- 第 2 周：逐步求精
 - 指针与引用的运用
 - 自由存储与对象生命周期管理
 - 容器类型的运用
- 第 3 周：迭代开发
 - 模板与代码优化
 - 模板函数与模板类的运用
 - 异常处理的运用

- 课件
 - www.se.sjtu.edu.cn 上的elearning
 - <http://courses.cs.tamu.edu/teresa/engr112/engr112-index.html>
 - <http://www.stroustrup.com/Programming/>
- 教科书
 - Programming : principles and practice using C++, Bjarne Stroustrup, Addison-Wesley
 - 中文版: C++程序设计原理与实践, 王刚译, 机械工业出版社, 2010-6-1
- 参考书
 - 完美的C++教程, Walter Savitch著, 江山等译, 机械工业出版社, 2007
 - 《C++ Primer (第四版)》, Stanley B. Lippman、Barbara E. Moo、Josée LaJoie著, 李师贤等人译, 人民邮电出版社, 2006
 - 《C++编程规范》, Herb Sutter、Andrei Alexandrescu, 人民邮电出版社, 2005

- 使用C++编程语言，以面向对象编程方式，开发一个狐狸与猎狗棋、五子棋或黑白棋游戏



- 开发分为两个阶段进行迭代，对每个阶段的要求如下：
- 第一阶段：为期2周
 - 实现功能：
 - 能够提供图形化界面供棋手下棋，推荐但不限定使用FLTK；
 - 能够实现人-人在同一台机器上下棋，即不需要实现网络对战，也不需要实现人-机对战；
 - 能够判断棋手所下的每一手棋是否合法；
 - 能够准确地判断棋局的输赢，并能够终止棋手在有输赢结果的棋局上继续下棋。
 - 提交制品：
 - 源代码：中间必须编写有助于程序理解的注释；
 - 设计文档：描述设计思路和方案，根据给定的模板写。
- 第二阶段：为期2周
 - 实现功能：
 - 能够实现人-机对战，但是不要求机器一定要每局都赢；
 - 能够对棋谱进行记录，并能够在棋盘上显示棋谱；
 - 能够按照棋谱的记录，按照行棋顺序复盘，复盘时应该注意每步棋之间的时间间隔。
 - 提交制品：
 - 源代码：中间必须编写有助于程序理解的注释；
 - 设计文档：描述设计思路和方案，根据给定的模板写；
 - 总结文档：对课程设计全过程的总结。

- 扩展功能
 - 同学们还可以在上述基础上有选择地设计并实现扩展功能，包括：
 - 指定扩展功能：
 - 换肤：更换棋盘背景和棋子形状、颜色、图案等；
 - 回合限时：对每步棋限定时间，超时惩罚，例如判输、随机走棋，或者是空过等；
 - 棋手等级管理：记录每位棋手的输赢局数，并自定义等级评定等；
 - 机器角色扮演：可以让机器棋手进行角色扮演，在人类棋手思考过程中出现对话，例如“快点呀！”“没招了吧！”等。可以参考三国五子棋中的类似功能。
 - 其他自定义定扩展功能：这部分功能完全由个人发挥，没有限制。

- 本课程的目标是培养学生面向对象程序设计能力和问题解决能力，因此将从程序设计的角度而非人工智能的角度评分。
- 最终成绩由平时成绩和答辩成绩组合而成：
- 平时成绩：50%
 - 第一阶段提交制品：15%
 - 程序设计质量：10%，考核依据为源代码和设计文档
 - 程序编写质量：5%，考核依据为源代码
 - 第二阶段提交制品：25%
 - 程序设计质量：15%，考核依据为源代码和设计文档
 - 程序编写质量：10%，考核依据为源代码
 - 学生参与课程的程度：10%

- 本课程的目标是培养学生面向对象程序设计能力和问题解决能力，因此将从程序设计的角度而非人工智能的角度评分。
- 最终成绩由平时成绩和答辩成绩组合而成：
- 答辩成绩：50%
 - 程序演示：30%
 - 实现第一阶段与第二阶段的所有功能：20%
 - 实现扩展功能：10%
 - 答辩：20%
 - 宣讲：10%
 - 回答问题：10%
 - 答辩形式：每位同学先进行自述，并演示程序，然后回答答辩委员组的提问。其中，自述和演示程序的时间为10分钟，回答问题10分钟，共20分钟
 - 答辩委员组：即课程指导小组成员，各成员的打分需进行加权平均，其中指导教师占60%，每个学生指导助教占20%，其他所有助教占20%

- 《软件工程职业素养》第一周的讲座课安排：
 - 周三下午2:00~4:30
- 讲座内容：团队凝聚力
- 讲座方式：团队体验式训练
- 讲座地点：软件学院演播厅
- 讲座老师：陈猛

Practice of Programming 1 Class

Haopeng Chen

***RE**liable, **IN**telligent and **Scalable** Systems Group (**REINS**)*

Shanghai Jiao Tong University

Shanghai, China

e-mail: chen-hp@sjtu.edu.cn

- 回顾
- 类设计
 - 接口要易于使用
 - 复制构造器
 - 设计类如同设计类型

- 思想

- 类表示的是程序中的概念
 - 如果可以将“事物”当做单独的实体看待，那么它就可以是一个类或者是一个类的对象
 - 例如: vector, matrix, input stream, string, FFT, valve controller, robot arm, device driver, picture on screen, dialog box, graph, window, temperature reading, clock
- 类是 (用户定义) 类型，它指定了其对象可以如何创建和使用
- 在 C++ (以及大多数现代语言)中，类是构造大型程序的关键组成部分
 - 对小程序也非常有用
- 这个概念最初是在 Simula67中引入的

- 看待类的方式

```
class X {           // this class' name is X
    // data members (they store information)
    // function members (they do things, using the information)
};
```

- 例如

```
class X {
public:
    int m; // data member
    int mf(int v) { int old = m; m=v; return old; } // function member
};
```

```
X var;           // var is a variable of type X
var.m = 7;       // access var's data member m
int x = var.mf(9); // call var's member function mf()
```


// simple Date (control access)

```
class Date {
    int y,m,d;      // year, month, day
public:
    Date(int y, int m, int d); // constructor: check for valid date and initialize

    // access functions:
    void add_day(int n);      // increase the Date by n days
    int month() { return m; }
    int day() { return d; }
    int year() { return y; }
};
```

```
// ...
Date my_birthday(1950, 12, 30);
cout << my_birthday.month() << endl;
my_birthday.m = 14;
```

// ok
// we can read
// error: Date::m is private

Date:

my_birthday:

<i>y</i>	1950
<i>d</i>	12
<i>m</i>	30

// simple Date (some people prefer implementation details last)

```
class Date {
public:
    Date(int y, int m, int d); // constructor: check for valid date and initialize
    void add_day(int n);      // increase the Date by n days
    int month();
    // ...
private:
    int y,m,d;      // year, month, day
};
```

*my_birthday: **Date:***

<i>y</i>	1950
<i>d</i>	12
<i>m</i>	30

```
Date::Date(int yy, int mm, int dd)
    :y(yy), m(mm), d(dd) { /* ... */ };
void Date::add_day(int n) { /* ... */ };
```

// definition; note :: “member of”
// note: member initializers
// definition

- 为什么要有public和private的区别呢？
- 为什么不能所有东西都是public的？
 - 为了提供整洁的接口
 - 数据和复杂的函数可以设成private的
 - 为了维护不变式
 - 只有固定的函数集可以访问数据
 - 为了使调试更加容易
 - 只有固定的函数集可以访问数据
 - (称为“围捕嫌犯”技术)
 - 为了允许类的表示的变更
 - 只需要变更固定的函数集
 - 不用确切知道谁在使用 public的成员

```
class Date {  
public:  
    // ...  
    int day() const { return d; }    // const member: can't modify  
    void add_day(int n);             // non-const member: can modify  
    // ...  
};
```

```
Date d(2000, Date::jan, 20);
```

```
const Date cd(2001, Date::feb, 21);
```

```
cout << d.day() << " - " << cd.day() << endl;    // ok
```

```
d.add_day(1);    // ok
```

```
cd.add_day(1);    // error: cd is a const
```

```
class Date {
    int y,m,d; // year, month, day
public:
    Date(int y, int m, int d); // constructor: check for valid date and
    initialize

    // access functions:
    void add_day(int n); // increase the Date by n days
    int month() { return m; }
    int day() { return d; }
    int year() { return y; }
};
```

- Date d(30, 3, 1995); // 参数顺序不正确
- Date d(2, 20, 1995); // 参数值非法

```
struct Day {  
    explicit Day(int d)  
    :val(d) {}  
    int val;  
};  
  
struct Month {  
    explicit Month(int m)  
    :val(m) {}  
    int val;  
};  
  
struct Year {  
    explicit Year(int y)  
    :val(y){}  
    int val;  
};
```

```
class Date {  
    public:  
        Date(const Month& m, const Day& d, const Year& y);  
    ...  
};
```

```
Date d(30, 3, 1995); // error! wrong types  
Date d(Day(30), Month(3), Year(1995)); // error! wrong types  
Date d(Month(3), Day(30), Year(1995)); // okay, types are correct
```

```
class Month {
public:
    static Month Jan() { return Month(1); } // functions returning all valid
    static Month Feb() { return Month(2); } // Month values;
    ...
    static Month Dec() { return Month(12); }
    ... // other member functions
private:
    explicit Month(int m); // prevent creation of new Month values
    ... // month-specific data
};

Date d(Month::Mar(), Day(30), Year(1995));
```

```
class IntSet {  
    int *elts; // pointer to dynamic array  
    int sizeElts; // capacity of array  
    int numElts; // current occupancy  
public:  
    IntSet::IntSet(int size=MAXELTS); // constructor  
    // REQUIRES: size > 0  
    // EFFECTS: create a set with size capacity  
    ...  
};
```


- 执行下面的代码会发生什么？

```
void foo(IntSet x) {  
    // do something  
}
```

```
int main() {  
    IntSet s;  
    s.insert(5);  
    foo(s);  
    s.query(5);  
}
```

- ◆ 类是按值传递的，就像struct一样
- ◆ 它们是逐位复制的，就像struct一样
- ◆ 当 `foo` 执行结束后，`x` 被删除
- ◆ 同时会删除`x`指向的动态数组
- ◆ 这使得 `s.elts` 成为了悬空指针
- ◆ 不良事件！

- 下面的代码将出现相同的情况:

```
void foo()
{
    IntSet s(5);
    s.insert(7);
    {
        IntSet x;
        x = s;
    }
    s.query(7); // Undefined!
```

- 赋值语句会将s的元素复制给x的元素
 - 但是它们共享了s的elts数组
 - x的elts数组就成为了孤儿
- 当x作用域执行结束时，就被销毁，s.elts就悬空了

- 那么，这是怎么回事呢？
- 按值传递参数和赋值操作的语法要求：
 - 应该将s的内容复制给x
- 但是，实际情况并非如此
 - 最终这两个对象共享了elts[]数组
- 我们真正想要的是：
 - 除了复制元素外，还要复制这个数组

- 当一个类包含指向动态元素的指针时
 - 复制它就会棘手一些
- 如果只复制类中的成员
 - 这就是浅复制(shallow copy)
- 通常，我们希望得到一个所有事物的完全副本
 - 这就称为深复制(deep copy)

- C++有两种机制来实现深复制
- 复制构造器：
 - 一块无形(formless)内存空间(blob, 二进制字节大对象), 以及一个样本实例
 - 创建这个样本的一个blob副本
- 复制构造器与其他构造器所起的作用是一致的

- 我们可以声明为IntSet声明一个复制构造器

```
class IntSet {  
    int *elts; // array of elements  
    int numElts; // number of elements in array  
    int sizeElts; // capacity of array  
public:  
    IntSet(int size = MAXELTS); // client optionally names size  
    IntSet(const IntSet &is); // copy constructor  
    ...  
};
```

- 引元之所以是常量，有两个原因：
 - 首先，如果按值传递
 - 那么最好传递它之后不要对其做任何修改!
 - 其次，这可以确保该类的任何实例都可以当做一个复制源
 - 而不仅仅只是一个可变化的对象
- 复制构造器必须完成两个任务：
 - 分配一个与源集合同尺寸的数组
 - 从源数组中复制每一个元素到新数组中
 - 复制 numElts/sizeElts 域
- 复制工作必须在复制构造器和赋值语句中都执行
- 因此，我们将复制工作抽取到一个工具函数中
 - 这样就需要添加一个private方法


```
class IntSet {  
    int *elts; // array of elements  
    int numElts // number of elements in array  
    int sizeElts // capacity of array  
    void copyFrom(const IntSet &is);  
    // REQUIRES: this has enough capacity to hold is  
    // MODIFIES: this  
    // EFFECTS: copies is contents to this  
public:  
    IntSet(int size = MAXELTS); // client optionally names size  
    IntSet(const IntSet &is); // copy constructor  
    ...
```

- 在编写这个函数前
 - 考虑copyFrom函数应该做些什么
- 首先，copyFrom 应该像别的方法一样
 - 在调用它之前表示不变式必须得到满足
- 其次，copyFrom 要求源集合与目标集合具有相同的容量

```
void IntSet::copyFrom(const IntSet &is)
{
    for (int i = 0; i<is.sizeElts; i++) { // Copy array
        elts[i] = is.elts[i];
    }
    numElts = is.numElts;
}
```

- 复制构造器需要调用copyFrom.

```
IntSet::IntSet(const IntSet &is)
```

```
{  
    elts = new int[is.sizeElts];  
    sizeElts = is.sizeElts;  
    numElts = 0;  
    copyFrom(is);  
}
```

- 将复制构造器与缺省的复制方法相比较
 - 缺省的复制方法只做了一件事，即复制elts/numElts/sizeElts域
- 复制构造器跟踪指针
 - 并复制了指针指向的对象
 - 而不是仅仅只复制了指针
- 这就被称为深复制
 - 与浅复制的缺省行为形成了对比

- 我们如何处理赋值呢？

```
IntSet s1(5);
```

```
IntSet s2(10);
```

```
s1 = s2; // assign s2 to s1
```

- 缺省的C++编译器在执行这个赋值操作时仍旧会使用浅复制
- 这与复制构造器的工作紧密相关
- 就像复制构造器一样
 - 赋值必须使用 rvalue 到 lvalue 的深复制

- 但是，lvalue的类实例必须先被销毁
 - 否则会造成内存泄露
- 要实现这个目的，需要为IntSet类重新定义赋值操作符
- 这称为操作符重载

- 赋值操作符
 - 向其提供目标实例和源实例
 - 将目标实例变成源实例的一个副本

- 下面是我们定义的赋值操作符:

```
class IntSet {  
    // data elements  
    ...  
public:  
    // Constructors  
    ...  
    IntSet& operator= (const IntSet& is);  
    ...  
};
```


- 我们不需要知道：
 - 为什么赋值操作符会返回lvalue的引用
 - 只需要知道它就是这样做的即可
- 这里“酷”的是
 - 由于我们已经写好了copyFrom
 - 所以赋值操作符很容易实现
- 我们只需直接使用copyFrom即可

```
IntSet& IntSet::operator=(const IntSet &is)
```

```
{  
    if (sizeElts != is.sizeElts) {  
        delete [] elts;  
        numElts = is.numElts;  
        elts = new int[numElts];  
    }  
    copyFrom(is);  
    return *this;  
}
```

- 最后一行显得有些奇怪
- 每个方法都有一个“隐式地(implicit)”局部变量**this**
 - **this**是一个指针，指向在其上调用该方法的当前实例
- 这一行会对该指针解地址，并返回它所指向的对象的引用
- 我们不能只返回**this**
 - 因为指针与引用并不完全一样
- 但是，我们必须返回被赋值对象(*assigned-to object*)的引用
 - 而不是赋值对象(*assigned-from object*)的引用

- 如果我们执行下面的代码，会发生什么呢？

```
IntSet s(50);
```

```
s = s;
```

- 答案是没什么不良后果，但是显得很浪费
- 应该执行检查避免这种操作
 - 但是因为我们可能并不经常会“自赋值(self-assign)”
 - 所以每次都检查与偶尔将数组复制给其自身相比，代价可能还要高昂

- 设计新类时需要像设计新类型一样考虑下列问题：
 - 新类的对象应该如何被创建和销毁？
 - 构造器和析构器设计
 - 对象的初始化和对象的赋值是否需要深复制？
 - 复制构造器和赋值操作符重载
 - 新类的对象如果被“按值传递”意味着什么？
 - 复制构造器
 - 新类的对象的“合法值”是什么？
 - 要进行有效性检查
 - 是否存在继承关系？
 - 子类化与转型
 - 成员与访问权限
 - Private或public
 - 真的需要新类型吗？



Thank You!