

Practice of Programming 2

Class II

Haopeng Chen

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

Shanghai Jiao Tong University

Shanghai, China

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

- 子类化
- 常见问题
- 类设计
 - 优良设计与欠佳设计分析

- 如果编写的程序被证明非常好，但是想在其中添加新特性，这时怎么办？
 - 直接修改这个程序吗？
 - 或者创建该程序的一个副本，然后在这个副本上修改
 - 以使得该程序可以被复用？
- 在现实生活中，我们都想将一样事物扩展为另一样事物
- 在程序设计中
 - 我们通过子类型化(sub-typing)来实现此目的

- 如果我们有两个类型 S 和 T
- S 是 T 的子类型，记作 $S <: T$ ，当且仅当：
 - 对于希望是 T 类型对象的任何实例
 - 都可以向其提供 S 类型的对象
 - 不需要修改原来的任何计算代码以保证程序的正确性
- 如果 S 是 T 的子类型，那么：
 - T 类型的对象可以被 S 类型的对象所替换
 - 并没有改变程序的任何属性，例如正确性

- 如果 $S <: T$ ，那么也可以说：
 - T 是 S 的超类型(supertype)
- 超类型**不能**替换子类型
 - 所有的苹果都是水果，但是并非所有水果都是苹果
- 这与类型转换不同
- 例如：
 - 当需要double类型时
 - 可以将int转换为double，从而改变int的物理表示
- 在需要使用超类型的地方使用子类型
 - 并不会将其转换为超类型，而是将其当做超类型使用

- C++拥有支持子类型化的机制
 - 成为子类化(subclassing), 有时也称为继承(inheritance)
- 其基本思想是:
 - 如果有一个ADT类foo
 - 并且需要创建其子类型bar, 那么就可以声明:
`class bar : public foo{...};`
- 这在声明:
 - bar是一个foo, 可能还包含额外的状态
 - 并且可能还包含新的或重定义的成员函数

- 如果需要在导出类中访问基类(base class)的成员
 - 同时需要阻止其他从基类外部的访问
- 怎么解决呢？
 - 在基类中使用protected修饰符
- 如果一个成员是protected的，那就意味着：
 - 它可以被这个类以及所有的导出类的成员访问
- Protected 数据成员使得导出类变得异常脆弱
 - 这使得我们需要好好考虑是否值得这么做

- 如果需要多个方法该怎么办呢？
 - 例如，我们希望能够在IntSet类的子类MaxIntSet中处理空集合
- 通过修改新的导出类中的方法
 - 或者重载/重定义这些方法
- 子类型必须遵守替换原则
- 这就意味着新方法必须执行旧方法中的所有操作
 - 但是可以做更多的事情
- 类似地，它对调用者的要求不能比旧方法多
 - 但是可以更少
- 换句话说，能够正常使用超类型的代码
 - 必须仍旧能够正常使用超类型


```
class PosIntSet : public IntSet {  
    // OVERVIEW: a subclass, but not a subtype  
public:  
    void insert(int v);  
    // EFFECTS: if v is non-negative and s has room to include it, s=s+{v}  
    // if v is negative throw int -1; if s is full thrown int MAXELTS  
};  
  
void PosIntSet::insert(int v) {  
    if (v < 0) throw -1;  
    IntSet::insert(v);  
}
```

- 能够正确使用IntSet的代码在使用PosIntSet时有可能会失败
 - 它并没有遵守替换原则
- C++使用的是“子类”替换，而不是“子类型”替换
- 这意味着编译器允许子类：
 - 用于任何希望使用超类型的地方
 - 即使它是不可替换的
- 例如，下面的代码是完全合法的：

```
PosIntSet s;  
IntSet& r = s;
```

- 当代码中包含引用时，需要区分明显类型和实际类型
 - Apparent type vs. Actual type
- 明显类型：引用的声明类型
- 实际类型：引用的真实类型
 - `PosIntSet s;`
 - `IntSet& r = s;`
- 注意：即使编译器“看到”这种赋值
 - 它也不认为它了解的比“声明”的更多
- 在这个例子中，`r = s` 不会改变 `r` 的明显类型
 - 这对于方法选择有十分重要的含义

- C++静态地选择要运行的方法
 - 基于在编译时了解的情况
 - 或者基于明显类型
- ```
PosIntSet s;
IntSet& r = s;
```
- 打破了s集合的表示不变式，这十分糟糕
  - 静态
    - 在程序执行之前编译器即知道
  - 动态
    - 不能静态地知道，只有在运行时才能了解到

```
void foo(IntSet &bar);
```

```
IntSet s;
```

```
PosIntSet p;
```

```
foo(s);
```

```
foo(p);
```

- 由于同一个引用同时指向了IntSet和PosIntSet
  - 所以编译器无法选择
- 有办法解决吗?
  - 有，通过使用虚函数强制编译器在运行时选择

- 在函数声明前增加关键字"*virtual*"

```
class IntSet {
```

```
 ...
```

```
 public:
```

```
 ...
```

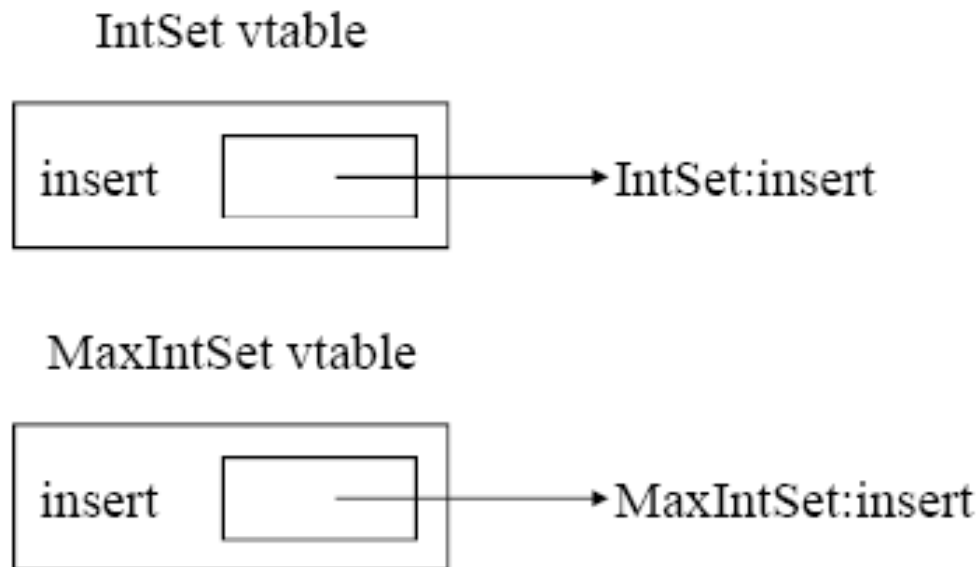
```
 virtual void insert(int v);
```

```
 ...
```

```
};
```

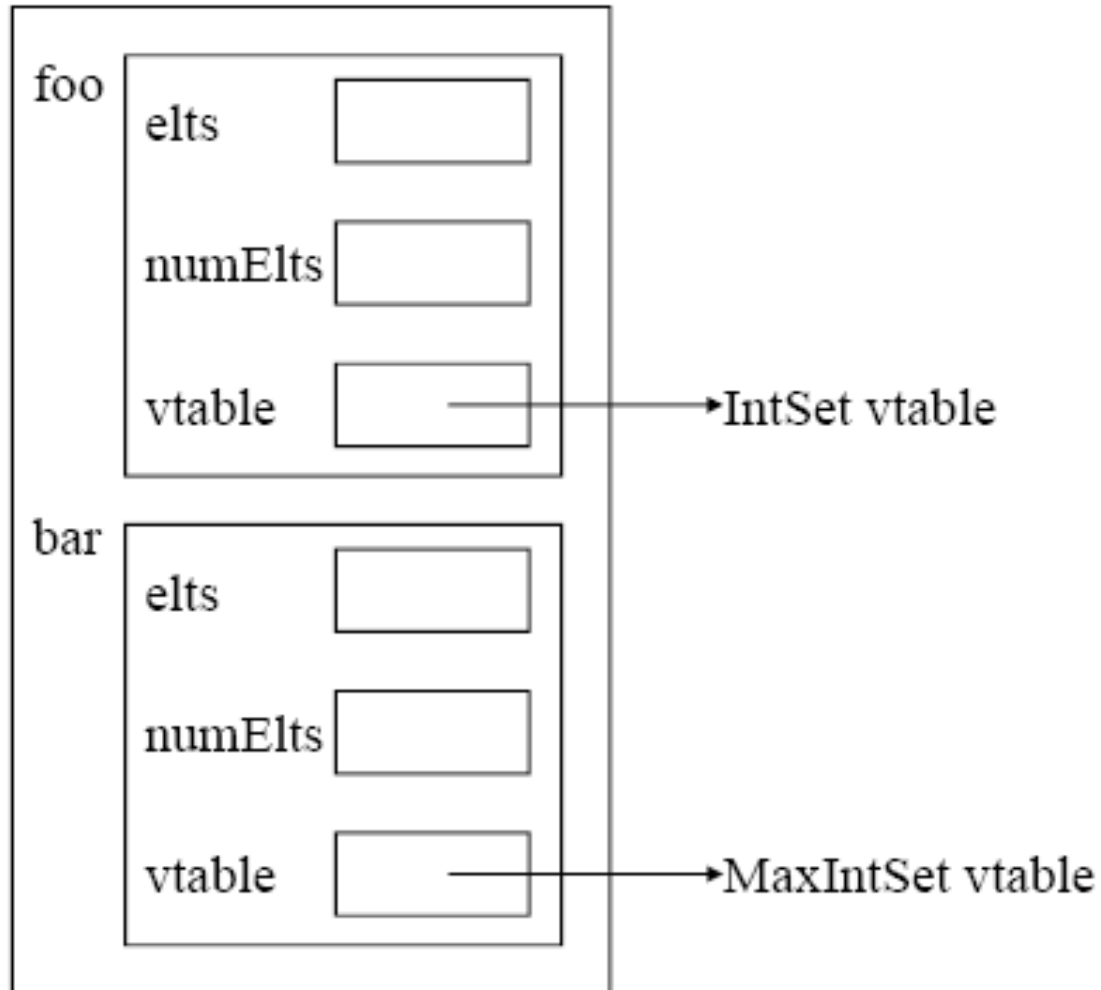
- 告知编译器某个类可能会覆盖我的实现：
  - 总是在运行时检查需要调用哪个版本
- 在PosIntSet的定义中不需要加上virtual关键字
  - 因为“虚”是可继承的，就像别的成员一样

- 对于拥有虚函数的类
  - 编译器将创建一个“vtable”，即“虚表(virtual table)”
  - 其中包含一组函数指针
  - 每个指针指向一个虚函数
- 这些指针都将初始化为指向恰当的实现





# IntSet foo; MaxIntSet bar;



- 虚函数强制编译器
  - 将选择函数实现的工作延迟到运行时
- 这种函数选择的延迟成为延迟绑定(late binding)
- 这种将一个函数的多个实现关联起来的机制
  - 称为多态(Polymorphism)
- 多态、延迟绑定、虚函数实际上是一回事

- 如果我们正在创建一个导出类的实例
  - 基类的构造器将先被调用
  - 然后导出类的构造器才被调用
- 这条规则可以递归地作用
  - 知道我们到达导出链的根类(root class)

- 因此，如果我们有下面的类：

```
class foo {
 int f;
 public:
 foo();
};
```

```
class bar : public foo {
 int r;
 public:
 bar();
};
```

```
class baz : public bar {
 int z;
public:
 baz();
}
```

```
baz *bzp = new baz;
```

- 将会指向下面的操作
  - 为baz创建内存空间(三个整数, z、r 和 f)
  - 调用foo()的构造器()
  - 调用bar()的构造器()
  - 调用baz()的构造器()
  
- 这意味着导出类的构造器可以认为:
  - 它已经是其基类的一个“正确(proper)”实例了

# 为什么不把所有方法都变成虚函数？

- 虚函数的优点已经很明显了
- 但是它也有缺点！
- 主要缺点之一：开销
  - 使用了更多的存储空间
  - 延迟绑定是“操作中处理(on the fly)”的，因此程序运行较慢
- 因此，虚函数如果不是必需的，则不应该使用

- 假如：
- Derived是Base的导出类
  - Derived对象可以赋值给Base类型的对象
  - 但是反之不可以！



```
class Pet {
 public:
 string name;
 virtual void print() const;
};
```

```
class Dog : public Pet {
 public:
 string breed;
 virtual void print() const;
};
```

- 成员 **name** 和 **breed** 是公有的，这只是为了举例，属非典型

- 现在假设有如下声明:

```
Dog vdog;
```

```
Pet vpet;
```

- 任何是狗的对象都是宠物:

```
vdog.name = "Tiny";
```

```
vdog.breed = "Great Dane";
```

```
vpet = vdog;
```

- 这些语句是允许的
- 子类型可以赋值给父类型，但是反之不然
  - 宠物不都(必须)是狗

- 注意，赋值给 `vp` 的对象，就丢失了其 `breed` 域：  
`cout << vpet.breed;`
- 这称为切片问题
- 看起来似乎是合适的
  - Dog 被当做了 Pet 变量，因此它不应该被当做 Pet
    - 因此也就不需要有 “dog” 特有的属性
- 但是对于严谨的计算机科学家来说，这么做是有问题的
  - 因为信息丢失了

- 在C++中，切片问题是令人讨厌的
  - 它仍旧是一条纯种的狗
  - 我们希望在它被当做Pet时仍旧可以引用到其breed属性
- 有什么方法可以实现这个目的吗？
- 是的，虚函数！
- 但是如何做呢？
  - 可以将虚函数与指向动态变量的指针结合使用

```
Pet *ppet;
Dog *pdog;
pdog = new Dog;
pdog->name = "Tiny";
pdog->breed = "Great Dane";
ppet = pdog;
```

- 不能访问ppet指针指向的对象的breed域  
`cout << ppet->breed; //ILLEGAL!`

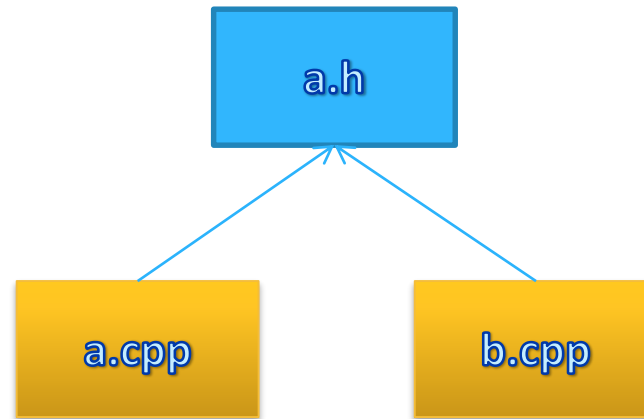
- 必须使用虚成员函数:

```
ppet->print();
```

- 调用Dog类中的print成员函数
  - 因为它是虚函数
- C++在绑定调用前会先观察ppet指向何种对象

- 不需要了解如何使用它！
  - 信息隐藏的原则
- 虚函数表
  - 编译器创建的
  - 包含指向每个虚成员函数的指针
  - 指向函数的正确的代码位置
- 这种类的对象也都有一个指针
  - 指向虚函数表

- 函数重复定义问题





- 头文件末尾不要忘记 “;”
  - 否则会报编译错误

- Newpoint.h

```
#include "Point.h"
class Newpoint : Point{
 Image im;
public:
 Newpoint(int x, int y);
};
```

- Newpoint.cpp

```
#include <cmath>
#include <sstream>
#include "Simple_window.h" // get access to our window library
#include "Graph.h" // get access to our graphics library facilities
#include "Newpoint.h"
```

```
Newpoint::Newpoint(int xx, int yy):im(Point(), "c:\1.png"){
 x=xx;
 y=yy;
}
```

- 冒号初始化与括号初始化的区别

- Newpoint.cpp

```
Newpoint::Newpoint(int xx, int yy):im(Point(), "c:\1.png"){
 x=xx;
 y=yy;
}
```

或

```
Newpoint::Newpoint(int xx, int yy){
 x=xx;
 y=yy;
 im(Point(), "c:\1.png");
}
```

或

```
Newpoint::Newpoint(int xx, int yy):x(xx),y(yy),im(Point(), "c:\1.png"){}
```

- 冒号初始化与括号初始化基本没什么区别
  - 但在非静态`const`类型以及引用型成员变量必须在初始化列表里面初始化,不能在`{}`里面初始化
  - 从父类继承而来的属性不能进行冒号初始化

- 探究

```
int a=10;
```

```
int a(10);
```

- 括号赋值只能在变量定义并初始化中,不能用在变量定义后再赋值

```
int a;
```

```
.....
a=10; \\正确
```

```
int b;
```

```
.....
b(10); \\错误
```

From: <http://www.cppblog.com/luyulaile/archive/2011/02/14/140059.html>

- 类构造函数的作用是创建一个类的对象时，调用它来构造这个类对象的数据成员
  - 一要给数据成员分配内存空间
  - 二要给数据成员初始化，按数据成员在类中声明的顺序进行构造
- 冒号初始化与函数体初始化的区别在于：
  - 冒号初始化是给数据成员分配内存空间时就进行初始化
  - 函数体初始化是在所有的数据成员被分配内存空间后才进行的

From: <http://www.cppblog.com/luyulaile/archive/2011/02/14/140059.html>

```
class student {
public :
 student ()
protected:
 const int a;
 int &b;
}
student ::student (int i,int j)
{
 a=i;
 b=j;
}
```

- **Student**类不能通过编译
- 因为常量初始化时必须赋值，它的值是不能再改变的，与常量一样引用初始化也需要赋值，定义了引用后，它就和引用的目标维系在了一起,也是不能再被赋值的。
- **Student**类的构造函数应为:

```
student ::student(int i,int j):a(i),b(j){}
```

```
class teach
{
 public :
 teach(char *p="name",int a=0)
 :
 .
 .

 protected:
 char name[30];
 int age;
}
teach ::teach(char*p,int a)
{
 strcpy(name ,p);
 age=a;
 cout>>name>>endl;
}
```

```
class student
{
 public:
 student (char *p="name");
 .
 .
 .

 protected;
 char name[30];
 teach teacher;
};
student::student(char *p)
{
 strcpy(name,p);
 cout>>name>>endl;
}
```

- 在上面的程序中通不过编译，编译系统会告诉你teacher这个类对象缺默认构造函数，因为在teach类中没有定义默认的构造函数。

From: <http://www.cppblog.com/luyulaile/archive/2011/02/14/140059.html>

```
student::student(char *p,char *pl,int ag):teacher(pl,ag)
{
 strcpy(name,p);
 cout<<name<<endl;
}
```

- 在没有默认构造函数的时候，如果一个类对象是另一个类的数据成员，那么初始化这个数据成员，就应该放到冒号后面。

- 黑白棋

```
//Main.cpp
```

```
int main(){
 Start_window win(450,312,"Made by Sonny");
 return Fl::run();
}
```

```
//Start_window.h
```

```
struct Start_window :public Fl_Window{
 Start_window(int w, int h,const char *head);

private:
 Fl_Box *Title;
 Fl_Box *Background;
 Fl_Button *Start;

 Main_window Main;

 static void Start_cb(Fl_Widget *, void *);
 inline void Start_cb_i ();
};
```



- 黑白棋

```
//Main_window.h
struct Main_window :public Fl_Window{
 Main_window(int w, int h,const char *head);
private:
 Fl_Box *Title;
 Fl_Box *Background;
 Fl_Button *vs_com;
 Fl_Button *vs_player;
 Fl_Button *Quit;
 Fl_Button *Show;

 static void vs_com_cb(Fl_Widget *, void *);
 inline void vs_com_cb_i ();

 static void vs_player_cb(Fl_Widget *, void *);
 inline void vs_player_cb_i ();

 static void Quit_cb(Fl_Widget *, void *);
 inline void Quit_cb_i ();

 static void Show_cb(Fl_Widget *, void *);
 inline void Show_cb_i ();

 Play_window *Play;
};
```

- 黑白棋

```
//Play_window.h
struct Main_window :public Fl_Window{
//用于记录棋子位置，在AI中有用到
struct Position{...};
//胜利窗口
struct Win_window :public Fl_Double_Window{};
//用于记录每一步棋盘的信息
struct Checkerboard {};
struct My_desk :public Fl_Box{};

struct Play_window :public Fl_Double_Window{
 Play_window(int w, int h,const char *head,bool com);

private:
 bool show_time;

};
```

- 黑白棋

```
const int Dx[8] = {1,1,1,-1,-1,-1,0,0}; /* x方向数组*/
const int Dy[8] = {1,0,-1,1,0,-1,1,-1}; /* y方向数组*/
bool Start,First,Auto; /* Start记录是否开局, First记录玩家先后手*/
/* Turn=1代表黑子下, =2代表白子下, Step记录步数,
 Now=1表示当前为近卫方选择英雄 =2表示为天灾方,
 Hero_C表示当前选中的英雄编号, S表示选中的背景编号*/
int Turn,Step1,Step,Now,Hero_C,S;
/* Level记录难度等级, Mode=1代表PvP, =2代表PvE,
 Top表示状态上限,
 Hero_1记录近卫方选中的英雄, Hero_2记录天灾方选中的英雄*/
int Level,Mode,Top,Hero_1,Hero_2;
int Map[8][8]; /* 当前棋盘状态数组*/
int State[64][9][9]; /* 记录整个过程棋盘状态*/
```

```
class EventWindow: public Fl_Box
{
 public:
 EventWindow(int t, int l, int width, int height);
 int handle(int e);
 int handle_key(int e,int key);
};

/**-----以下为功能函数的声明-----***/

bool Is_Inside(int x,int y); /* Is_Inside函数用于判断坐标是否在棋盘内 */

bool Is_Chip(int x,int y,int i); /* Is_Chip函数用于判断坐标在方向i上是否夹主对方棋子 */

bool Is_OK(int x,int y); /* Is_OK函数用于判断坐标处能否落子(黑白棋规定必须夹住对方一子以上方能落子) */

bool Is_Turn(); /* Is_Turn函数用于判断玩家是否轮到下子,过滤一些AI下子时的无用按钮点击 */

bool Could_Play(); /* Could_Play函数用于判断当前棋盘能否落子, 如果任意位置都无法下子返回false */

int Calculate(int x,int y); /* Calculate函数为AI估价函数, 用于计算坐标处下子的价值 */

void AI_Turn(void *); /* AI_Turn函数为人工智能函数, 寻找当前难度模式下最符合要求的落子 */

.....
```



Thank You!