复习: 面向对象程序设计

徐枫

清华大学软件学院

feng-xu@tsinghua.edu.cn

编写实际程序所需要的.....

- 算法思路
  - 问题的分析、表示、求解的方法
  - 变量、判断、循环、函数的应用
- 编程语言
  - 如何定义和实现算法中的元素
  - 语法规范

- 操作系统知识
  - 操作系统运作的内在机制、机理
  - 操作系统提供的底层功能调用库
- 专门领域知识
  - 如：网络通信、硬件接口

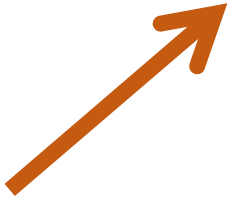
- 方法论
  - 如：结构化，基于对象，面向对象，泛型，组件

OOP是一种编程设计的方法论

- 如何直观分析问题？
- 如何快速实现算法？
- 如何方便修改代码？

高效实现程序，解决程序员的开发效率

OOP（Object Oriented Programming）



FOP（Functional Oriented Programming）

实现高效程序，解决计算机的运行效率

FOP课程与OOP课程的区别

- 程序设计基础（FOP）
  - 数字化
  - 可计算

重点讲解“怎么算？”

➔ 培养“计算思维”，只有计算才能解决复杂问题

- 面向对象程序设计基础（OOP）
  - 人性化
  - 易认知

重点讲解“怎么看？”

➔ 培养“抽象思维”，只有抽象才能认知复杂世界

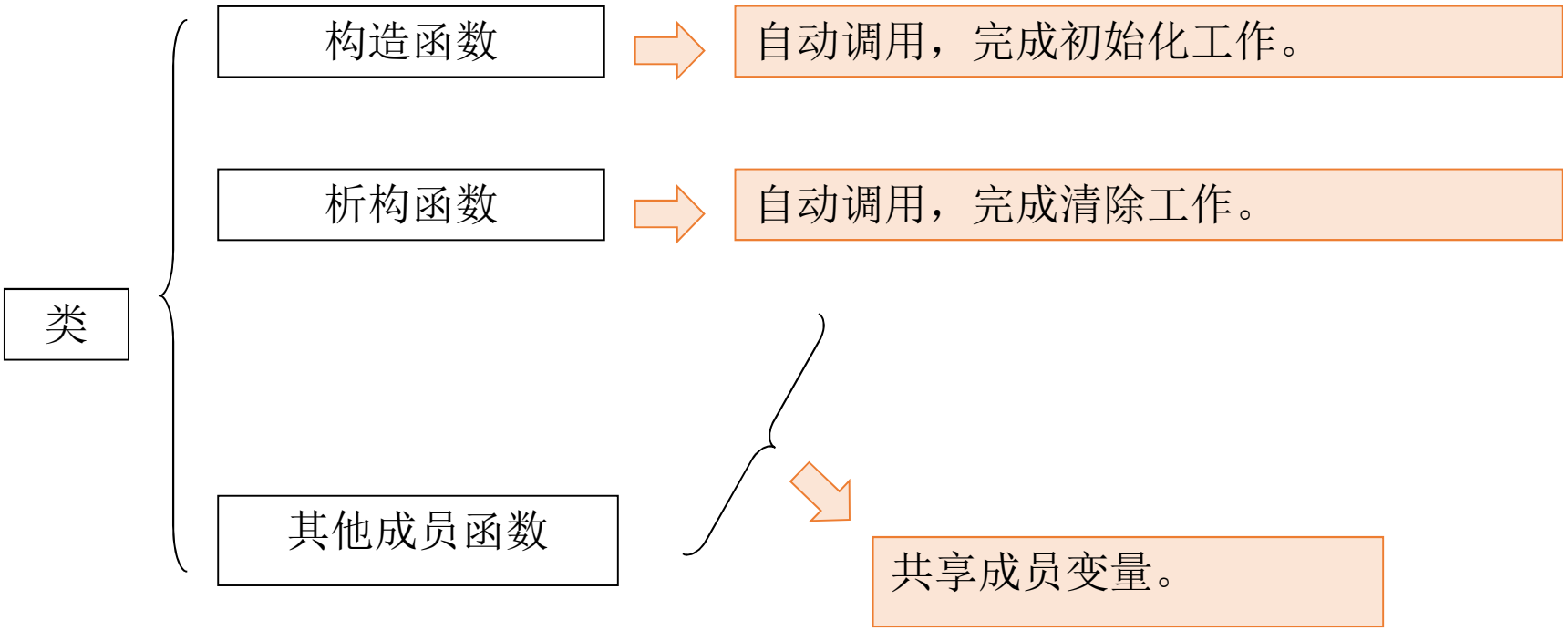
面向对象程序设计方法

- 面向对象程序设计是建立在结构化程序设计基础上的
- 面向对象程序设计以对象（类的实例）作为构造程序的基本单元，将程序和数据封装其中，
  - 重用性
  - 灵活性
  - 扩展性

OOP从认识“对象”开始

- 对象，是对现实世界中实际存在事物的抽象描述，它可以是有形的，也可以是无形的
  - 对象具有自己的静态特征和动态特征
    - 静态特征，是可以用某种数据来描述的特征；
    - 动态特征，是对象所表现的行为或所具有的功能。
  - 对象是由一组属性和对这组属性进行操作的一组服务构成的结合体：
    - {一组属性，一组操作} ➔ OOP中的“封装”的概念！

面向对象程序设计: 类



面向对象程序设计: 对象

- 什么是对象?

共享成员变量。

特指恋爱的对方

为什么伊会让人喜欢

构造函数

自动调用，
完成初始化工作。

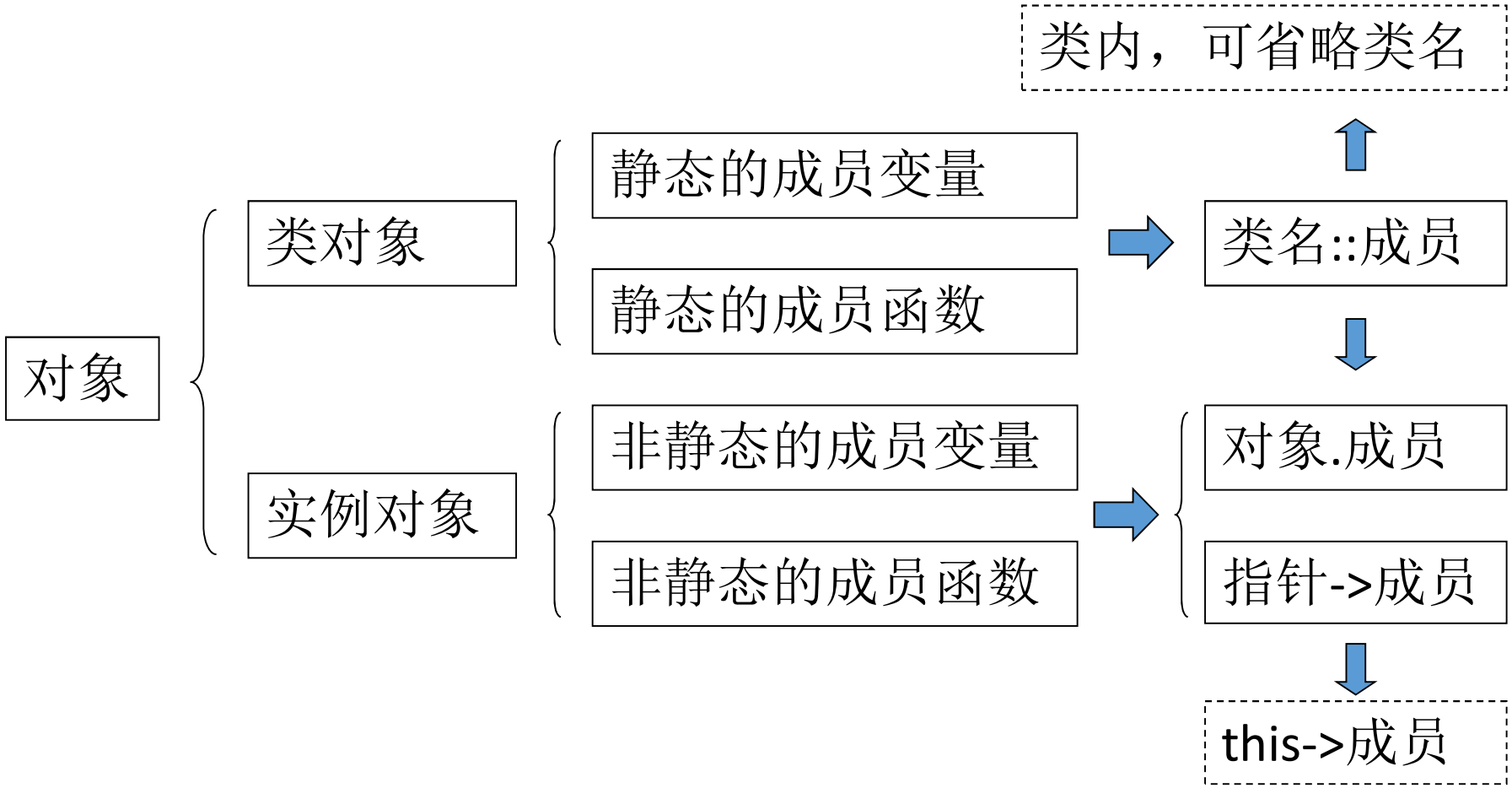
析构函数

自动调用，完成善后清除工作。

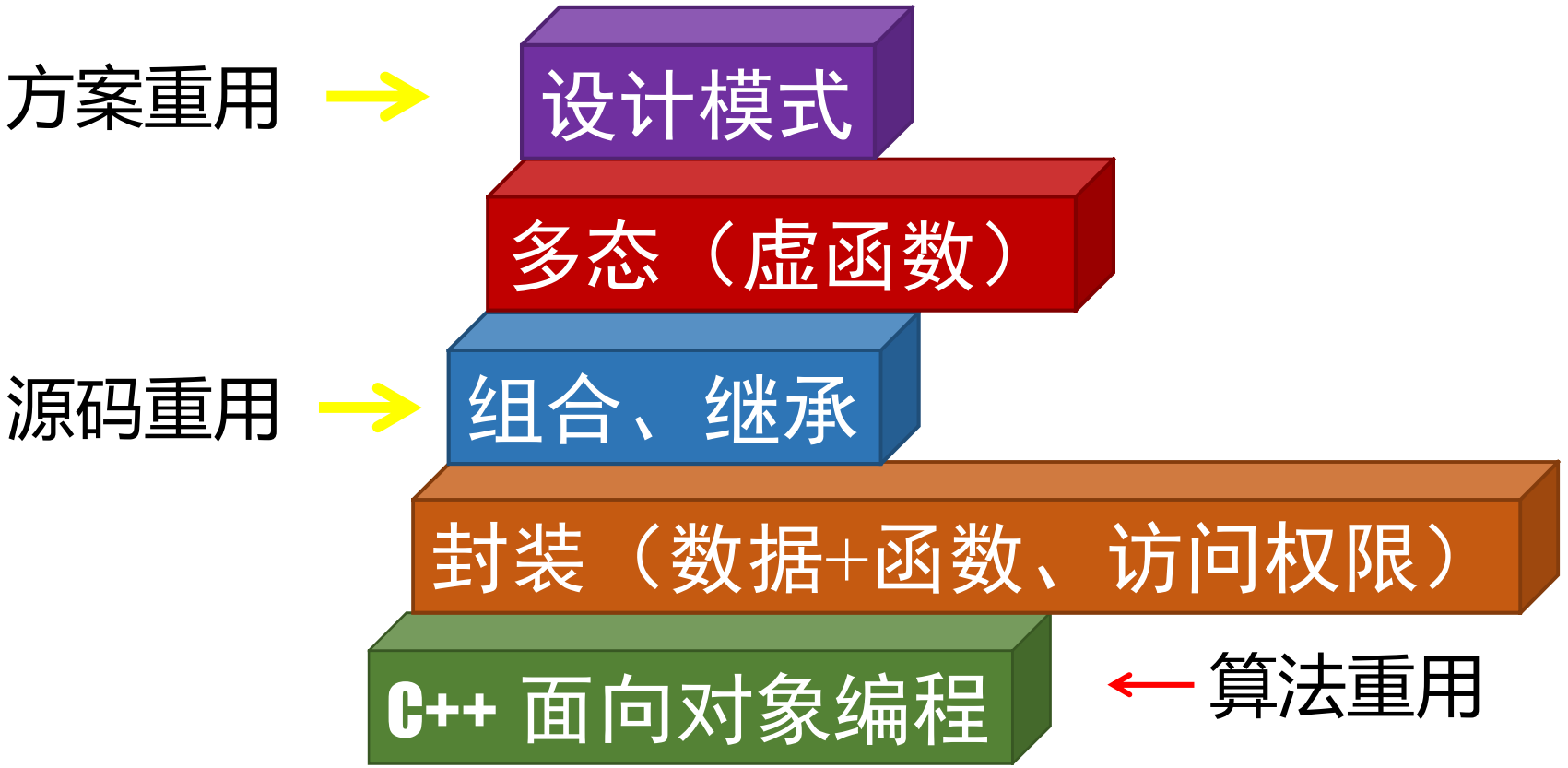
例子

```
class Lover
{
        int m_age;
        String m_name;
        String m_password;
Public:
        Lover(String name);
        ~Lover(String password); //有问题，析构函数不能带有参数且没有返回值
        String getPassword();
        int getAge();
        String getLoverName();
}
```

面向对象程序设计: 对象


类内，可省略类名
对象
类对象
静态的成员变量
静态的成员函数
类名::成员
实例对象
非静态的成员变量
非静态的成员函数
对象.成员
指针->成员
this->成员

课程内容—关键概念与技术



关键概念：封装、组合、继承、多态

面向对象程序设计: 三大特性

- 封装性
- 继承性
- 多态性

封装的“装”——数据抽象

- 对象——从程序语言角度看
  - 一个独立的、有约束的、有自己的记忆（数据成员）和活动能力（函数成员）的实体
  - 或：用用户自定义的类作为类型的变量！
- 数据+函数 ➔ 称为封装，亦称“数据抽象”
  - 是面向对象程序的基本特征
  - 封装之后，结构便既能描述属性（数据），又能描述行为（函数）。

封装性:C++中表达“封装”的方法

- 用结构class将变量定义（属性）和函数定义（操作）包含到一起，使属性与操作不仅在逻辑上是一个整体，而且在物理上也是一个整体。语法规则如下所示：
- 文件前后的条件编译选项，是为了防止头文件被重复包含时出错。

```
#ifndef __STUDENT_H__
#define __STUDENT_H__

class Student
{
  public:
        char Name[20];
        void show();
};
#endif // __STUDENT_H__
```

```
#ifndef __PERSON_H__
#define __PERSON_H__

class Person
{
  public:
        char Name[20];
        void show();
};
#endif // __PERSON_H__
```

```cpp
#include "person.h"
#include <iostream>
using namespace std;
void Person::show()
{
      cout << "Name: "
           << Name << endl;
}
// person.cpp
```

```cpp
#include "student.h"
#include <iostream>
using namespace std;
void Student::show()
{
      cout << "Name: "
           << Name << endl;
}
// student.cpp
```

```
// TEST-EX1.cpp - 包含头文件时，编译开发环境提供的库头文件在后
// g++ test-ex1.cpp person.cpp student.cpp
#include "person.h"    // person struct
#include "student.h"   // student struct
#include <cstring>     // strcpy

int main()
{
        Person manA, manB;
        strcpy(manA.Name, "Zhang San");
        strcpy(manB.Name, "Li Si");
        manA.show();
        manB.show();

        Student s1, s2; // 不要放到函数最前面，最好是随用随定义
        strcpy(s1.Name, "Wang Wu");
        strcpy(s2.Name, "Zhao Liu");
        s1.show();
        s2.show();

        return 0;
}
```

头文件编程规范

在包含头文件时，编译开发环境提供的库头文件（例如C++标准库）应放在后面

如果编译开发环境提供的库头文件在前，则可能会有隐患

头文件编程规范

例如头文件head.h中有用到std::string，但在编写写head.h时忘记了包含<string>，此时如果在使用时先#include <string>再#include “head.h”就不会编译报错，该问题也就难以被发现，容易引发之后的错误

一般来说头文件要做到self-contained，当其他部分要用到head.h的时候，只要#include “head.h”，而不需要再添加其他的依赖项。编写头文件时，可以把头文件放在最前面来检测是否做到了自给

头文件编程规范

如果可以的话，尽量将包含头文件语句放在.cpp文件中而不是.h文件中

这样做的优点有：

- .h文件可能在其他地方被多次引用，不在.h文件中包含过多的其他头文件能够减少冗余
- 能够更好的避免头文件互相引用或循环引用带来的编译错误

成员函数中隐含的this指针

- 为什么不同对象调用同一成员函数会产生不同的结果输出？
  - Person manA, manB;
  - manA.show(); manB.Show();
- 类的每个成员函数，有一个编译器传入的隐含参数，其类型为类的指针，名称为this（C++语言的一个关键字）。即
  - manA.show() → show(&manA)，即在函数体中实际上“知道”被操作的对象在哪里！

成员函数中隐含的this指针

```
void Person::show()
{
        cout << "Name: "
             << Name << endl;
}
```

编译器实际生成的代码相当于：

```
void show(Person* this)
{
        cout << "Name: "
             << this->Name << endl;
}
```

封装的“封”——权限控制

修饰词	同一个类	子类	类的对象
public	允许访问	允许访问	允许访问
protected	允许访问	允许访问	
private	允许访问		

```
struct A {
  int i;
  char j;
  float f;
  void func();
};

void A::func() {}

struct B {
public:
  int i;
  char j;
  float f;
  void func();
};

void B::func() {}

int main() {
  A a; B b;
  a.i = b.i = 1;
  a.j = b.j = 'c';
  a.f = b.f = 3.14159;
  a.func();
  b.func();
}
```

```
struct B {
private:
  char j;
  float f;
public:
  int i;
  void func();
};

void B::func() {
  i = 0;
  j = '0';
  f = 0.0;
};

int main() {
  B b;
  b.i = 1;          // OK, public
//!  b.j = '1';     // Illegal, private
//!  b.f = 1.0;     // Illegal, private
}
```

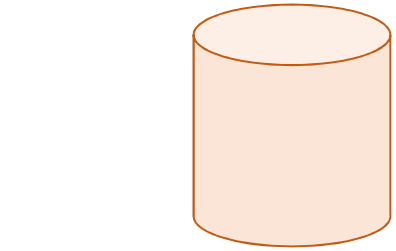
封装的“封”——接口概念

能通过对象来访问到的部分称为类的“接口”

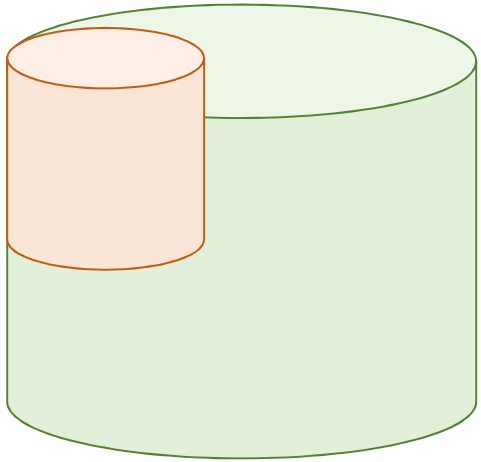
- 使用者能避开一些他们不需要使用的东西（细节）
  - 这实际上是方便了别人（使用类的编程人员）--- 接口使他们可以很容易知道类的哪些部分对他们是重要的，哪些是不重要可以忽略的
- 设计者能改变类的内部实现，而不必担心会对使用者产生影响
  - 这实际上是为了方便我们（设计和实现类的人）--- 接口使我们可以很容易修改具体实现（允许我们犯错误），而不会影响最终的功能

继承性

- 是关系原则:
  - 子类的实例对象同时通常也被认为是父类的实例对象。
- 扩展性原则:
  - 子类在其父类的基础上新增自己的特性。



父类实例对象



子类实例对象

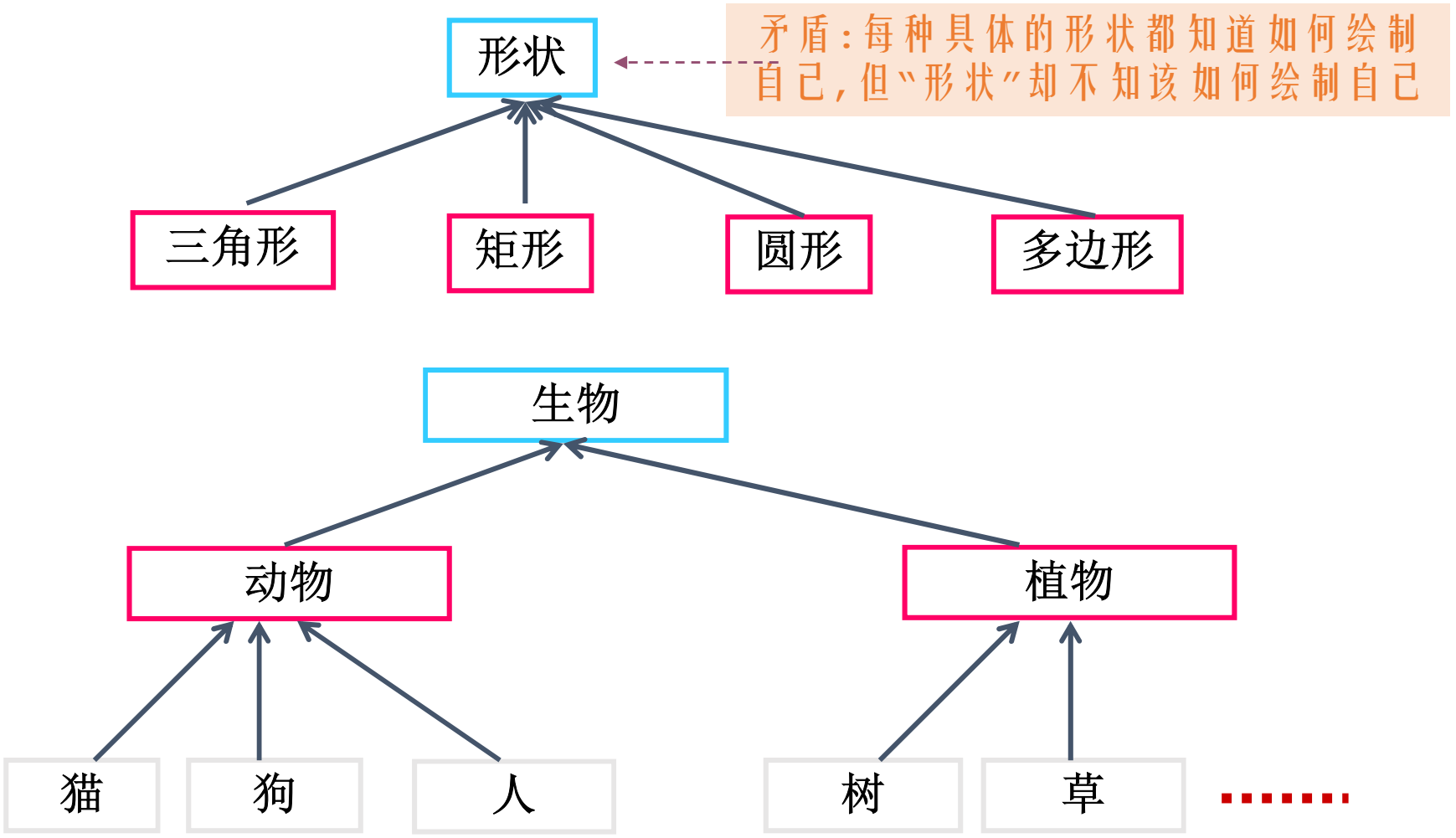
支持增量式开发

对象组合

- 汽车 =
  - 引擎 +
  - 轮子 +
  - 车窗 +
  - 车门 +
  - ……

对象之间有没有其它的关系呢？……

考虑下面的一些关系


形状
矛盾：每种具体的形状都知道如何绘制自己，但“形状”却不知该如何绘制自己
三角形
矩形
圆形
多边形
生物
动物
植物
猫
狗
人
树
草

继承

- 新的类“像”老的类
- 三种继承方式: **public，private，protected**

```
class Derived1 : public Base {
  // ...
};

class Derived2 : private Base {
  // ...
};

class Derived3 : protected Base {
  // ...
};
```

可见性

不同继承方式对子类对象及对外接口的影响

继承方式		public		private		protected	
父类成员类型	public	OK	pub	OK	prv	OK	pro
	private	--	--	--	--	--	--
	protected	OK	pro	OK	prv	OK	pro

派生类成员函数是否可访问基类成员（仅与成员类型有关）

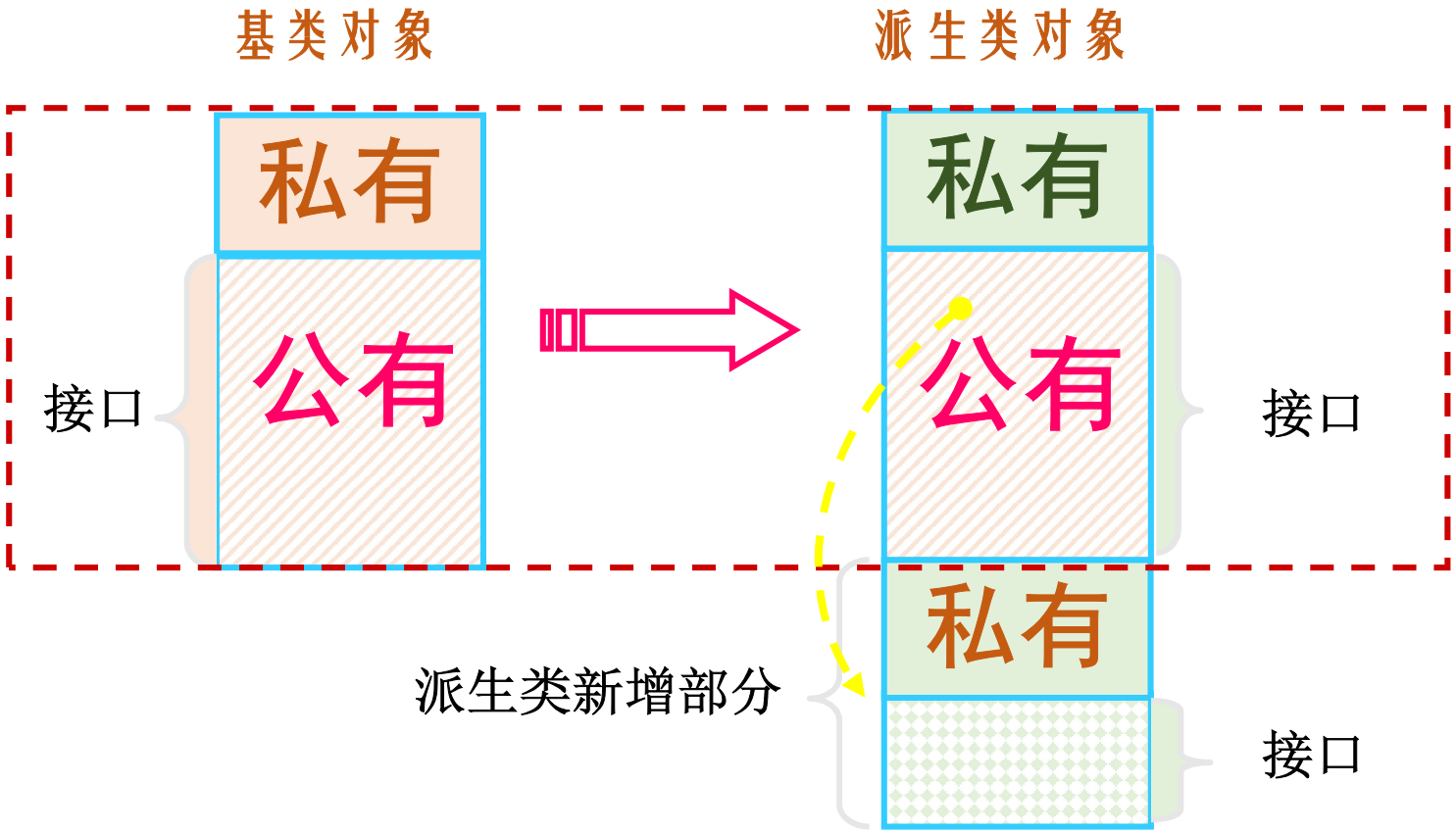
接口继承到派生类后的访问权限的变化（仅与继承方式有关）

派生类成员函数中是否可以访问基类的变量?

派生类对象是否可以访问原基类的变量?

```
Derived : public Base { }
```

“喂！派生类，我那些公有接口你可别独享，要允许别人用！”



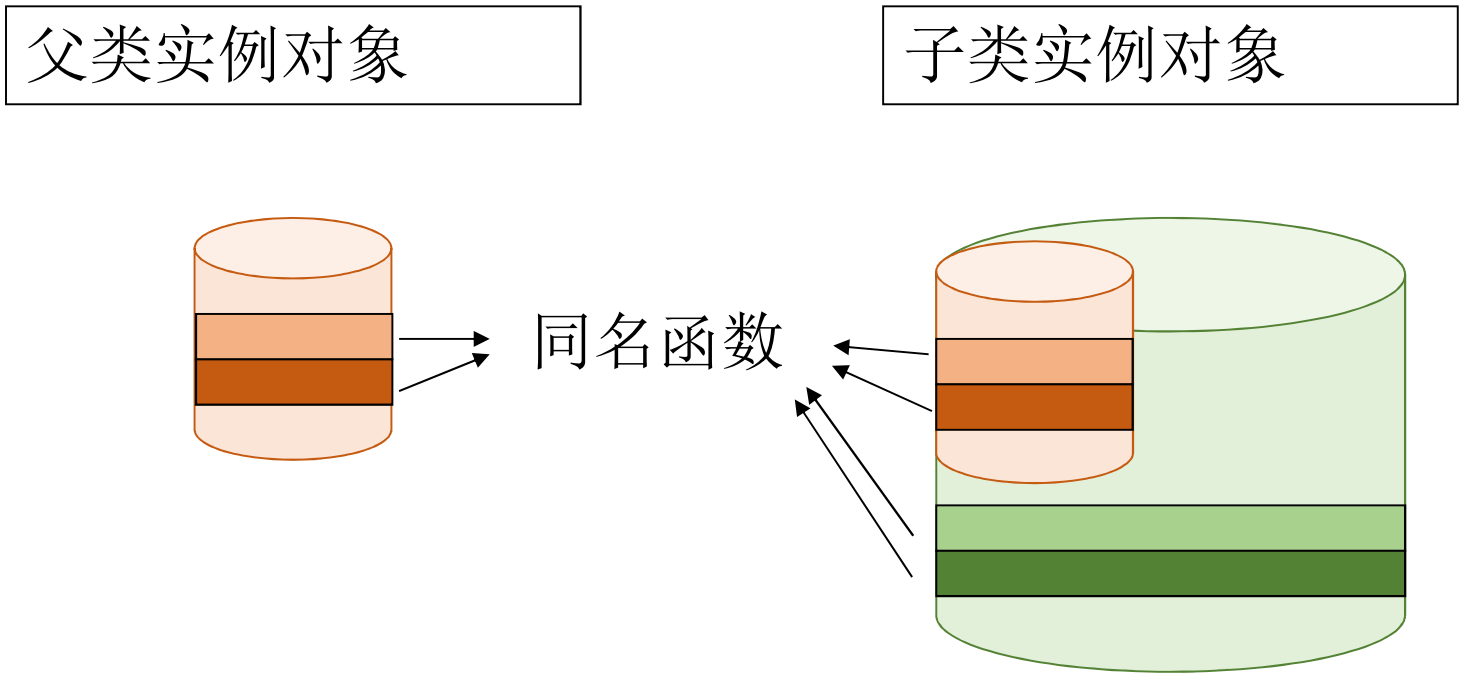
基类的私有成员，派生类能访问吗？

多态性

- 多态性是C语言不具备的特性。
- 多态性包括
  - 编译时的多态性——称为静态多态性
  - 运行时的多态性——称为动态多态性

静态多态性

- 在C++中，允许出现多个构造函数或多个同名的其他成员函数，它们的参数类型、参数个数、或者参数类型的排列顺序不同，在函数调用时可以区分/视为不同的函数。这称为构造/成员函数的重载，也称为静态多态性。



静态多态性的优缺点

- 优点
  - 效率较高，编译器也可以进行优化
- 缺点
  - 编译耗时、代码膨胀

如何在调试中避免长时间代码编译？

如何用release模式去debug？

到底要解决什么问题？！

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat };

class Instrument {
public:
  void play(note) const {
    cout << "Instrument::play" << endl;
  }
};
// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}
int main() {
  Wind flute;
  tune(flute); // Upcasting
}
```

Instrument::play

这个信息不是我们想要的

到底要解决什么问题？！

原因是函数调用是在编译期间确定的

- 编译器只知道传给tune的类型是instrument，所以它只会安排调用基类函数
- 希望能根据对象的实际类型来调用正确的函数！
- 怎么改程序呢？

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat };

class Instrument {
public:
  void play(note) const {
    cout << "Instrument::play" << endl;
  }
};
// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
  // Redefine interface function:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}
int main() {
  Wind flute;
  tune(flute); // Upcasting
}
```

解决之道：虚函数！

virtual关键字说明函数是虚函数

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
};
```

只要某个函数在基类中声明为虚函数，则在派生类中也是虚函数，而不论是否在派生类中也这样声明

程序输出
Wind::play

这样行不行？为什么？

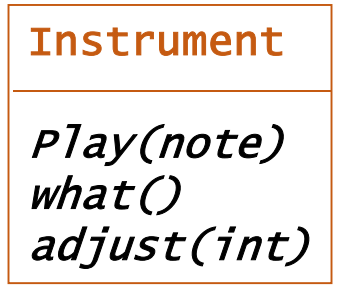
```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const
  { cout << "Instrument::play" << endl; }
};

class Wind : public Instrument {
public:
  void play(note) const { cout << "Wind::play" << endl; }
};

void tune(Instrument i) {
  // ...
  i.play(middleC);
}
int main() {
  Wind flute;
  tune(flute); // Upcasting
}
```

此处不加引用号，输出：Instrument::play；
此处加引用号，输出：Wind::play；

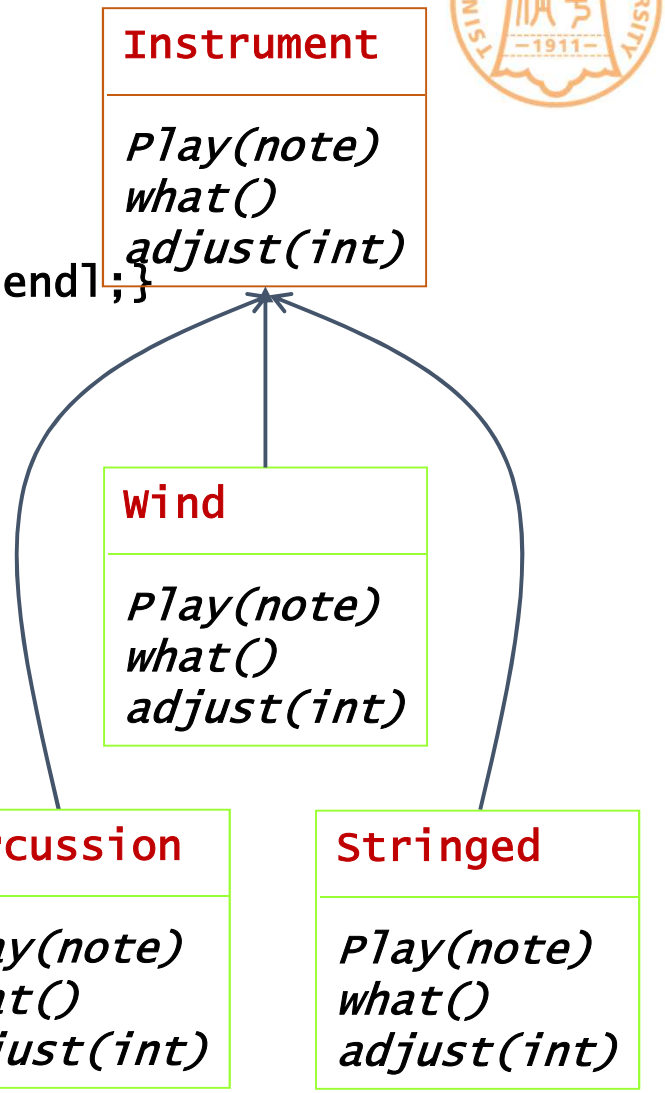
引用！指针！
只有引用和指针才能实现
动态多态性

多态与虚函数示例

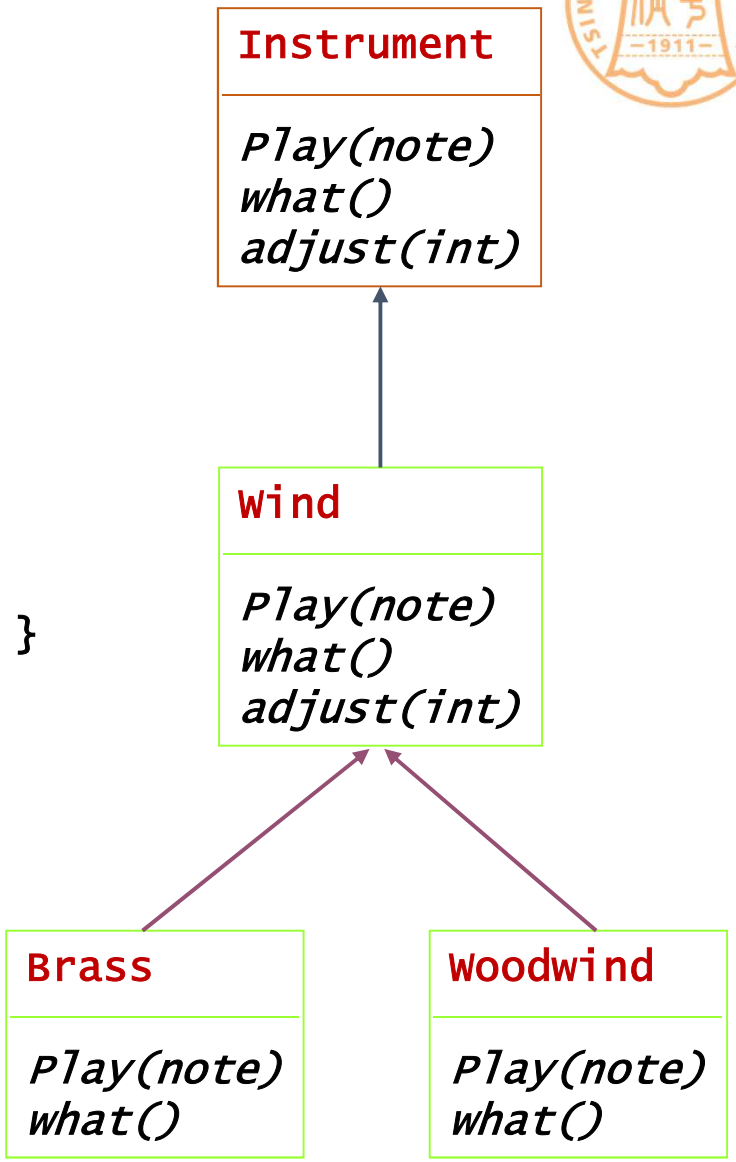
```
#include <iostream>
using namespace std;

enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }

  virtual char* what() const {
    return "Instrument";
  }

  // Assume this will modify the object:
  virtual void adjust(int) {}
};
```



```
class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const {return "Wind";}
  void adjust(int) {cout << "Wind::adjust" << endl;}
};

class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const {return "Percussion";}
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};
```



```
class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};


// Identical function from before:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }
```



```
// Upcasting during array initialization:
Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
}
```

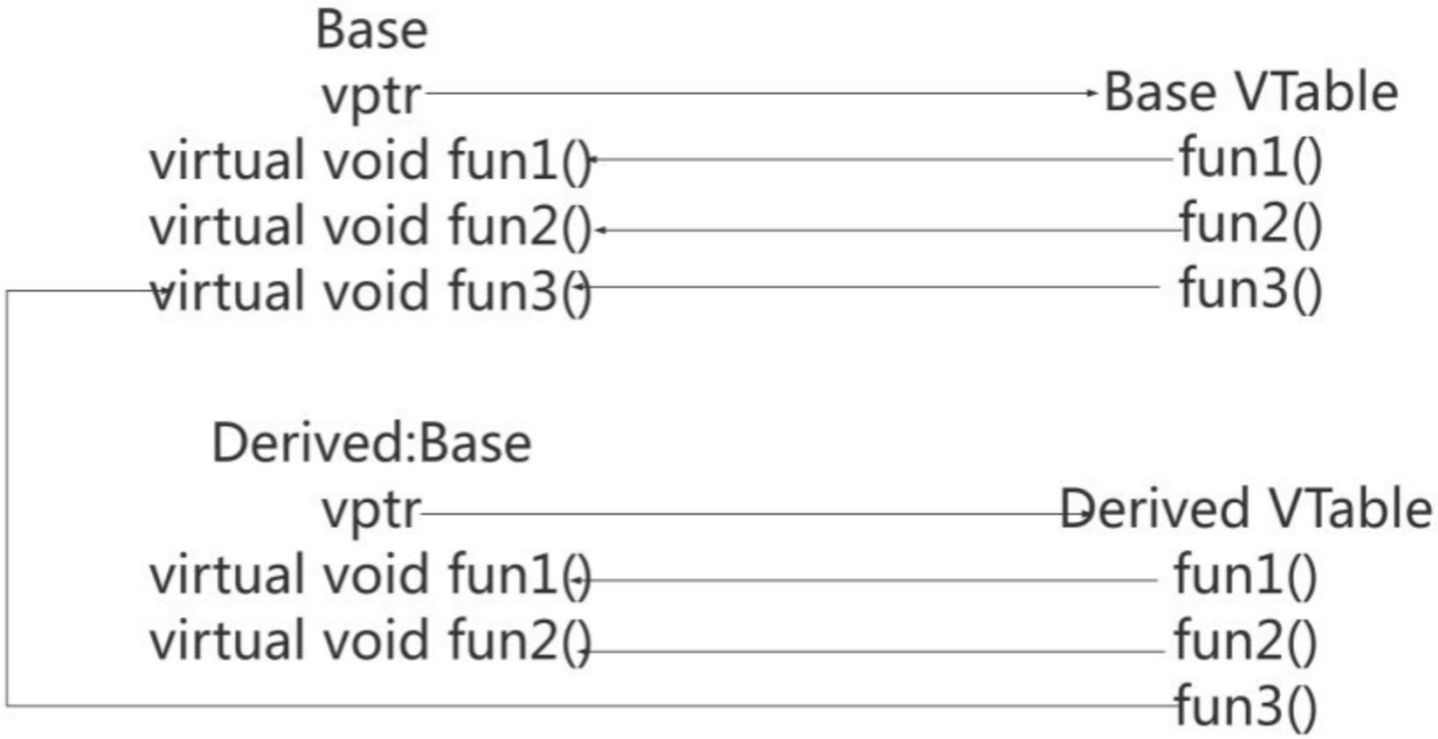
```
Wind::play
Percussion::play
Stringed::play
Brass::play
Woodwind::play
Wind::adjust
```

为什么？

- 函数绑定（Binding）
  - 确定函数调用到底要运行哪个函数体
  - 前期绑定（Early binding）
    - 编译时的函数绑定
  - 后期绑定（Late binding）
    - 运行时的函数绑定
    - 什么时候使用后期绑定：虚函数和指针/引用

怎么实现？

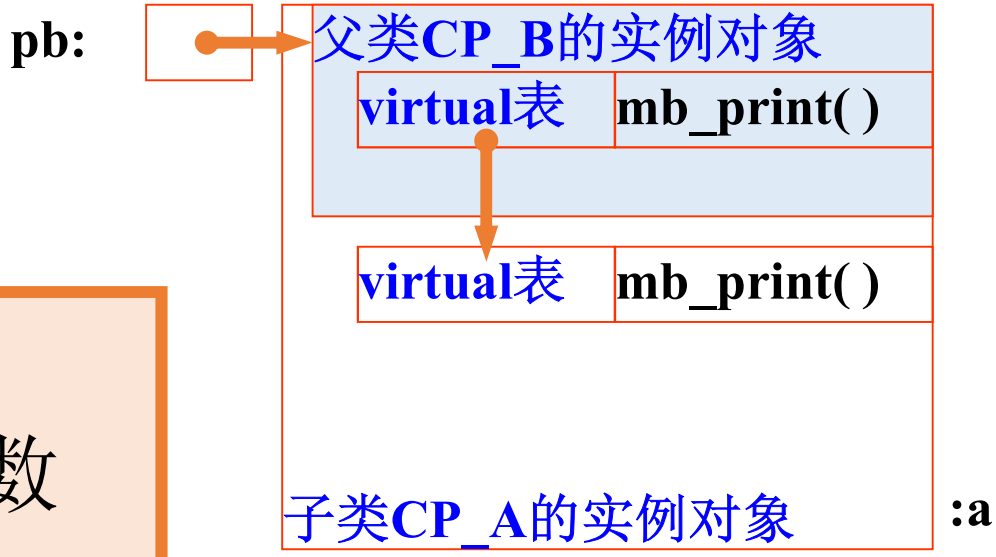
- C++中使用VTABLE来实现后期绑定
  - 每个含有虚函数的类都定义一个VTABLE
  - 这个VTABLE中包含定义该对象的类的虚函数实现指针
  - 类的对象中存储一个执行VTABLE的指针，称作VPTR



动态多态性的优缺点

- 优点
  - OO设计，是对客观世界的直觉认识
  - 实现与接口分离，可复用，处理异类集合
  - 可执行代码小（只有一个多态函数）
- 缺点
  - 运行期绑定，导致一定程度的运行时开销
    - VTABLE、VPTR、dynamic_cast
  - 编译器无法对虚函数进行优化

动态多态性



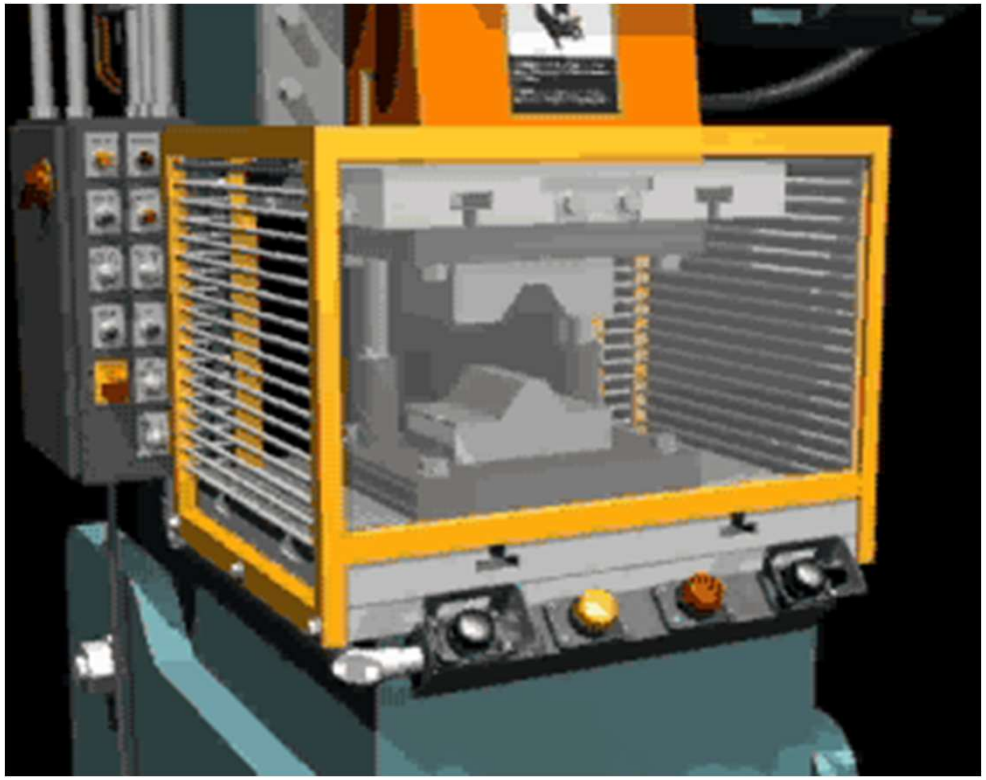
构造函数
非虚的成员函数

因为类还没有被创建，没有VTABLE，而析构函数可以为虚

虚拟函数调用只需要“部分的”信息，即只需要知道函数接口，而不需要对象的具体类型。但是构建一个对象，却必须知道具体的类型信息。如果你调用一个虚拟构造函数，编译器怎么知道你想构建是继承树上的哪种类型呢？所以这在逻辑上是一个悖论。

模板(Template)

- 模板是泛型编程的基础



C++中的模板包括两类

- 函数模板 ——用来生产函数
- 类模板 ——用来生产类

C++中的模板包括两类

- 函数模板 ——用来生产函数
- 类模板 ——用来生产类

任务1

- 编写一个函数，求两个整数中较大的

任务1

```
int my_max(int a, int b) {
    if (a > b)
        return a;
    return b;
}
```

任务2

- 编写一个函数，求两个实数中较大的

任务2

```
double my_max(double a, double b) {
    if (a > b)
        return a;
    return b;
}
```

任务3

- 编写一个函数，求两个字符中较大的

任务3

```
char my_max(char a, char b) {
    if (a > b)
        return a;
    return b;
}
```

小结

- 问题：这些函数的名字都一样，不会出错吗？
- 这么多函数，长得都一样，但是不得不写很多次
- 程序员是很懒的，有没有简洁的写法？

函数模板

```
template <typename T>      // 这是一个关于类型T的模板
T my_max(T a, T b) {
    if (a > b)
        return a;
    return b;
}

int main() {
    int a = 3, b = 5;
    cout << my_max(a, b) << endl;
    double c = 3.3, d = 5.5;
    cout << my_max(c, d) << endl;
    return 0;
}
```

函数模板

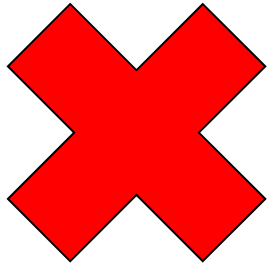
- 注意：函数模板本身并不是一个函数！
- 只有调用了适合函数模板的函数时，才会通过函数模板生成相应的函数
- 前例中，由my_max函数模板生成了两个函数，T = int、T = double
- 如果main函数中不调用my_max，则函数模板没有任何作用

任务4

- 编写一个函数，求两个字符串中较大的

任务4

```
char * my_max(char * a, char * b) {
    if (a > b)
        return a;
    return b;
}
```



任务4

```
char * my_max(char * a, char * b) {
    if (strcmp(a, b) > 0)
        return a;
    return b;
}
```

任务4

- 如果这个函数和之前编写的函数模板同时存在，会有什么效果？会不会冲突？

函数模板被重载的调用规则

- 先寻找一个参数完全匹配的函数，如果找到了就调用它；
- 否则寻找一个函数模板，使其实例化，产生一个匹配的函数，并调用它；
- 再否则，试一试低一级的对函数的重载方法，例如通过类型转换可产生参数匹配等，并调用它；
- 最后，如果还没找到，表明这是一个错误的调用。

函数模板的显示特化

```
template<> char * my_max<char *>(char * a, char * b) {
    if (strcmp(a, b) > 0)
        return a;
    return b;
}
```

- 与函数模板一样，这个函数也只在有调用的时候生成

函数模板小结

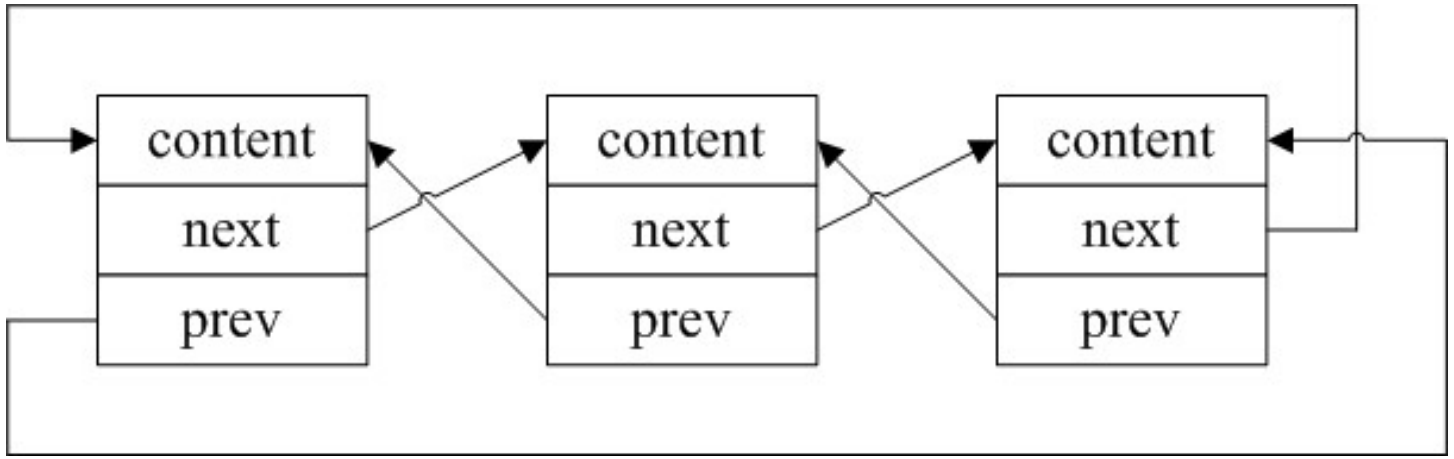
- 函数模板用来生产函数
- 关键字template
- 在函数前加上template <typename T>
- 函数定义时使用T做为一个类型使用

C++中的模板包括两类

- 函数模板 ——用来生产函数
- 类模板 ——用来生产类

任务5

- 设计一个循环双链表节点类，节点中包含一个int类型的内容，包含3个方法：在当前节点前添加一项，在当前节点后添加一项，从当前节点开始向后遍历输出所有项



任务5

```
class LinkNode {
public:
    LinkNode(int n);
    ~LinkNode();
    void AddBefore(LinkNode * p);
    void AddAfter(LinkNode * p);
    void Print();
private:
    int content;
    LinkNode * next, * prev;
};
```

任务5

```
LinkNode(int n) {
    content = n;
    next = this;
    prev = this;
}
~LinkNode() {}
```

```
void LinkNode::AddBefore(LinkNode
* p) {
    p->prev = prev;
    p->next = this;
    prev->next = p;
    prev = p;
}
void LinkNode::AddAfter(LinkNode *
p) {
    p->prev = this;
    p->next = next;
    next->prev = p;
    next = p;
}
```

任务5

```
void LinkNode::Print() {
    cout << content;
    for (LinkNode * p = next; p != this; p = p->next)
        cout << " <-> " << p->content;
    cout << endl;
}
```

任务6

- 设计一个循环双链表节点类，内容类型未知

任务6

```
class LinkNode {
public:
    LinkNode(?? n);
    ~LinkNode();
    void AddBefore(LinkNode * p);
    void AddAfter(LinkNode * p);
    void Print();
private:
    ?? content;
    LinkNode * next, * prev;
};
```

类模板

- 与函数模板类似，能否声明时先不指定某个类型？

任务6

```
template <typename T>
class LinkNode {
public:
    LinkNode(T n);
    ~LinkNode();
    void AddBefore(LinkNode * p);
    void AddAfter(LinkNode * p);
    void Print();
private:
    T content;
    LinkNode * next, * prev;
};
```

任务6

```
int main() {
    LinkNode<double> * head = new LinkNode<double>(0.5);
    head->Print();
    head->AddAfter(new LinkNode<double>(2.4));
    head->Print();
    head->AddBefore(new LinkNode<double>(1.3));
    head->Print();
    return 0;
}
```

当从一个类模板创建类时，必须提供一个具体类型以完成新类型的定义

类模板小结

- 用来生产类
- 关键字template
- 类定义前面加上template <typename T>
- 使用类时加上<类型>，指定实际的类型T

补充

- template <typename T>
- 等价于
- template <class T>

课堂练习1

- 编写一个函数模板，交换两个变量的值

课堂练习2

- 为LinkNode类模板编写一个方法，找出其中的最大值

课堂练习3

- 使用冒泡排序，为LinkNode类模板编写一个排序方法

OOP设计思想小结

- 封装之“数据抽象”，通过将数据与操作放到结构（类）中，使它们从逻辑上在一起，变成物理上在一起（形成类）
- 封装之“信息隐藏”，通过权限控制，形成明确的对外接口，从而隐藏实现细节，使我们在更改类实现时不会影响到使用类的代码。
- 类的接口定义很重要！它是使用类的依据，其变动会影响使用类的代码，应努力在分析设计阶段正确地确定，尽量一次到位。

OOP设计思想小结

- 若确有必要，接口部分还是可以修改的
  - 尽量采取“添加新接口函数”的方式
  - 上述方法好处是：不影响使用原有接口的现有代码
  - 所以，在开始设计一个类时，应使它的接口尽可能小而精，以后根据需要再扩大（即增加新的接口函数）

C++版本

Year	C++ Standard	Informal name
1998	ISO/IEC 14882:1998[23]	C++98
2003	ISO/IEC 14882:2003[24]	C++03
2011	ISO/IEC 14882:2011[25]	C++11, C++0x
2014	ISO/IEC 14882:2014[26]	C++14, C++1y
2017	ISO/IEC 14882:2017[9]	C++17, C++1z
2020	to be determined	C++20,[17] C++2a

集成开发环境

- Visual Studio
- VS 2017？
  - https://www.visualstudio.com/zh-cn
- 同学们目前都用的版本？

谢谢！