2018-2019年度第二学期 00106501

计算机图形学

童 伟 华　管理科研楼1205室

E-mail: tongwh@ustc.edu.cn

中国科学技术大学　数学科学学院

http://math.ustc.edu.cn/

附讲三　C/C++编程（二）

C++特点

- C++是面对对象（object oriented）编程语言
  - 纯面向对象语言：指不管什么东西，都应该存在于对象之中，例如JAVA和Small Talk语言
- 三大特性：
  - 封装（encapsulation）
  - 继承（inheritance）
  - 多态（polymorphism）
- 其它特性：
  - 动态生成（dynamic creation）
  - 异常处理（exception handling）
  - 泛型编程（template）

面向对象的分析与设计

- 大英百科全书：人类在认识和理解现实世界的过程中普遍运用着三个构造法则
  - 区分对象及其属性，例如，区分一棵树和树的大小或空间位置关系
  - 区分整体对象及其组成部分，例如，区分一棵树和树枝
  - 不同对象类的形成及区分，例如，所有树的类和所有石头的类的形成和区分
- 面向对象分析（Object-oriented analysis, OOA）与设计（Object-oriented design, OOD）方法是建立在对象及其属性、类属及其成员、整体及其部分这些基本概念的基础上

继承与组合

- 代码重用
  - 不同层次
  - C语言：函数、库等
  - C++语言：类、模板、设计模式等
- 如何重用已有的类？
  - 组合：创建一个包含已存在类的对象的新类，即将已存在类视为新类的属性，has-a关系
  - 继承：创建一个新类作为一个已存在类的类型，即将新类视为已存在类的特殊类型，is-a关系

组合（has-a）

```
//: C14:Useful.h
// A class to reuse
#ifndef USEFUL_H
#define USEFUL_H

class X
{
   int i;

public:
   X() {i = 0;}
   void set(int ii) {i = ii;}
   int read() const {return i;}
   int permute() {return i = i * 47;}
};

#endif // USEFUL_H ///:~
```

组合（has-a）

```
//: C14:Composition.cpp
// Reuse code with composition
#include "Useful.h"

class Y
{
   int i;
public:
   X x; // Embedded object
   Y() {i = 0;}
   void f(int ii) {i = ii;}
   int g() const {return i;}
};

int main()
{
   Y y;
   y.f(47);
   y.x.set(37); // Access the embedded object

   return(0);
} ///:~
```

继承（is-a）

```
//: C14:Inheritance.cpp
// Simple inheritance
#include "Useful.h"
#include <iostream>
using namespace std;

class Y : public X
{
   int i; // Different from X's i
public:
   Y() {i = 0;}
   int change()
   {
           i = permute(); // Different name call
           return i;
   }
   void set(int ii)
   {
           i = ii;
           X::set(ii); // Same-name function call
   }
};
```

继承（is-a）

```
int main()
{
   cout << "sizeof(X) = " << sizeof(X) << endl;
   cout << "sizeof(Y) = " << sizeof(Y) << endl;

   Y D;
   D.change();
   // X function interface comes through:
   D.read();
   D.permute();
   // Redefined functions hide base versions:
   D.set(12);

   return(0);
} ///:~
```

多态（polymorphism）

- **多态性：一般特殊结构中对象所体现的多态性**，即在一般类中定义的属性或操作被特殊类继承之后，可以具有不同的数据类型或表现出不同的行为
- **为什么要使用多态性？**例如：在“几何图形”中定义了一个操作“绘图”，但是并不确定在执行时究竟要画一个什么图形。特殊类“椭圆”和“多边形”都继承了几何图形的绘制操作，但其功能却不同：前者将画一个圆，后者画一个多边形。这样，当系统请求画一个几何图形时，消息中给出的操作名称都是“绘图”（因而消息的书写方式可以统一），而椭圆、多边形等类对象在接受到这个消息时却各自执行不同的绘图算法。

多态（polymorphism）

- 与多态性的实现相关的C++语言功能有：
  - 重载（overload）：函数、运算符重载等
  - 动态绑定（dynamic binding）：虚函数（virtual function）、运行时类型信息（run-time information）
- Evolution of C++ programmers
  - Better C
  - Object-based C++
  - True object-oriented programming: if you don't use virtual functions, you don't understand OOP yet.

多态（polymorphism）

```
//: C15:Instrument4.cpp
// Extensibility in OOP
#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) {}
};
```

多态（polymorphism）

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

多态（polymorphism）

```
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);
}
```

多态（polymorphism）

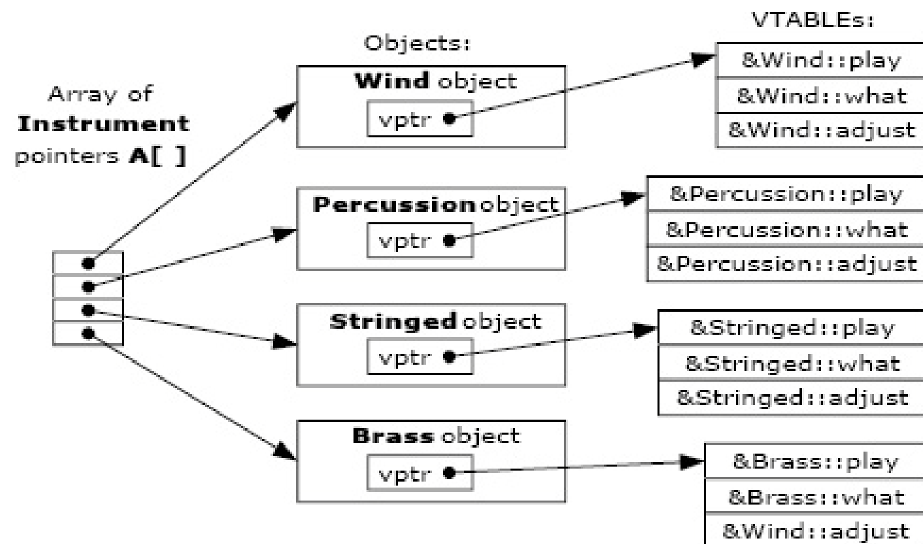
```
// New function:
void f(Instrument& i) {i.adjust(1);}

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

多态（polymorphism）

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

   tune(flute);
   tune(drum);
   tune(violin);
   tune(flugelhorn);
   tune(recorder);
   f(flugelhorn);

   int NumA = sizeof(A)/sizeof(*A);
   for (int i = 0; i < NumA; i++)
   {
        tune((*A[i]));
   }

   return(0);
} ///:~
```

多态（polymorphism）

- C++如何实现动态绑定（dynamic binding）或延迟绑定（late binding）？
  - 虚函数表：编译器对每个包含虚函数的类创建一个表（VTABLE），用于放置虚函数地址，并秘密的置一个指针（VPTB）指向该表
  - 当通过基类指针调用虚函数时，编译器静态的取得VPTB，通过它查找函数地址，进行调用
  - 关键点：基类指针调用虚函数，接口的统一，行为的多态
- 虚函数：除了函数调用时有一点额外的开销外，一切皆好！



运行时类型识别

- 运行时类型识别（run-time type identification, RTTI）：在我们只有一个指向基类的指针或引用时确定一个对象的准确类型
- 两种使用方法：
  - Typeid：typeid()带有一个参数，可以是一个对象引用或者指针，返回全局typeinfo类的产量对象的一个引用，编译器实现
  - 安全类型向下映射（type-safe downcast）：类继承的顺序，向上映射（upcast）总是安全的，譬如一个circle*到shape*，而shape*到circle*则不然，语法：circle* cp = dynamic_cast<circle*>(sp)，其中shape* sp = new circle
- RTTI的实现依赖于虚函数表中的信息
- 尽量使用虚函数，必要时才使用RTTI

Thanks for your attention!