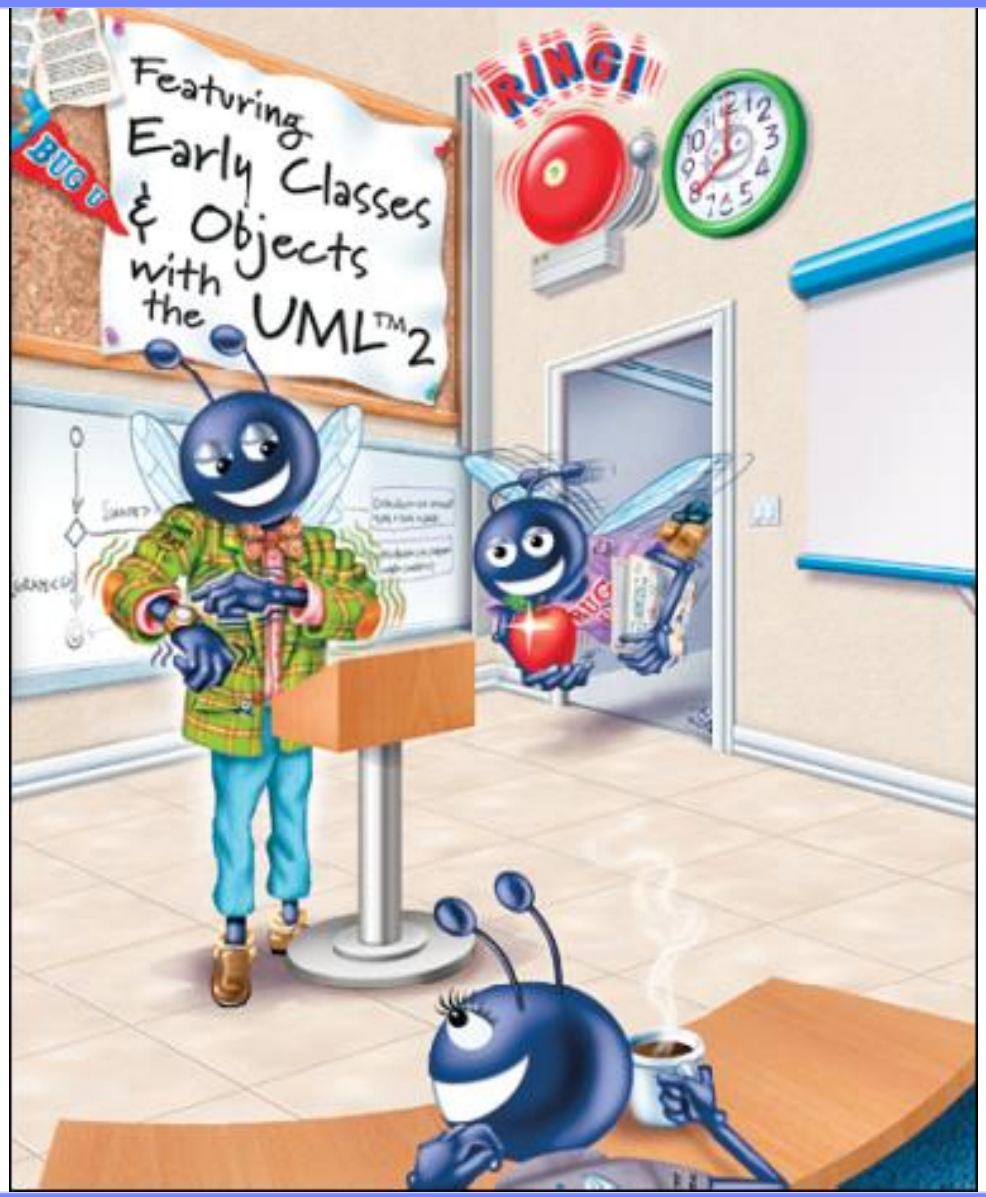


Lecture 10:
多态

第十三讲 面向对象编程：多态

学习目标：

- 理解多态性的概念
- 理解如何声明和利用虚拟函数来实现多态性
- 理解多态性如何扩展和维护系统
- 理解C++如何实现虚拟函数和动态绑定
- 理解运行时类型信息（RTTI）和运算符typeid和dynamic_cast的用法

面向对象的重要特征：多态性

多态性与封装性、继承性构成了面向对象程序设计的三大特征。

多态性是指相同的对象收到相同的消息时，或不同的对象收到相同的消息时，产生不同的处理结果。这里所说的消息是指对类的成员的调用，而不同的行为是指不同的实现，也就是调用了不同的函数。

利用多态性，用户只需发送一般形式的消息，而将所有的实现留给接收消息的对象，对象根据所接收到的消息而做出相应的动作（即操作）。

- 类的成员函数重载就是一种多态性。重载使得一个对象调用相同的成员函数时，会由于参数个数不同或参数类型不同而产生不同的调用结果。
- 类具有继承性，派生类既可以继承基类中定义过的成员函数，也可以覆盖基类中定义过的成员函数，这样，派生类对象调用一个成员函数时，就会因派生类中是否覆盖了基类中定义过的成员函数而有不同的调用结果。
- 派生类还可以重载基类中定义过的成员函数，这样，派生类对象调用一个成员函数时，就会由于参数个数不同或参数类型不同，而产生不同的调用结果。

静态多态与动态多态

- 从实现的角度来看，多态可以划分为两类：编译时的多态和运行时的多态。前者是在编译的过程中确定了同名操作的具体操作对象，而后者则是在程序运行过程中才动态地确定操作所针对的具体对象。这种确定操作的具体对象的过程就是**联编**（binding），也称为**绑定**。
- 联编可以在编译和连接时进行，称为**静态联编**。在编译、连接过程中，系统就可以根据类型匹配等特征确定程序中操作调用与执行该操作的代码的关系，即确定了某一个同名标识到底是要调用哪一段程序代码，函数的重载、函数模板的实例化均属于静态联编。
- 联编也可以在运行时进行，称为**动态联编**。在编译、连接过程中无法解决的联编问题，要等到程序开始运行之后再来确定。

归纳起来，面向对象技术支持的多态性主要包括4种：

(1) 同一个类中成员函数重载实现的多态性。这也称为**重载多态性**。

(2) 派生类对基类成员函数是否覆盖实现的多态性，以及派生类对基类成员函数是否重载实现的多态性。这也称为**继承多态性**。

(3) **运行时的多态性**，即对一个类层次来说，动态确定的类层次中的对象不同，则对象调用的成员函数不同。

(4) **参数多态性**。即用参数方法决定一个类的数据类型。

1 Introduction

- 继承层次中的多态（Polymorphism）
  - “Program in the general” vs. “program in the specific”
  - 在处理继承层次中的对象时，把所有对象都看作是基类的对象
  - 不同动作的执行依赖于对象的类型
  - 新类的添加不会对现有代码进行大幅修改

1 Introduction

- 例如：Animal 层次
  - Animal 基类 – 每个派生类都有 move 功能
  - 不同的 animal 对象用 Animal 指针向量（vector）表示
  - 程序向每个对象发送 move 消息
  - 正确的函数被调用
    - ✓ A Fish will move by swimming
    - ✓ A Frog will move by jumping
    - ✓ A Bird will move by flying

2 Polymorphism Examples

- 多态在程序通过基类指针或引用来调用虚拟函数时出现
  - C++ 动态选择所指向对象的正确函数

2 Polymorphism Examples

- 例如: SpaceObjects
  - 游戏处理从 SpaceObject 继承而来的各种对象，每个对象包含一个 draw 成员函数
  - 不同的类实现不同的 draw 功能
  - 屏幕管理程序维护一个 SpaceObject 指针容器

2 Polymorphism Examples

- 例如: SpaceObjects
  - 使用 SpaceObject 指针调用对象的 draw 函数
    - ✓ 正确的 draw 函数基于对象的类型被调用
  - 一个从 SpaceObject 继承而来的派生类的增加将不会影响到屏幕管理程序

3 Relationships Among Objects in an Inheritance Hierarchy

- 展示
  - 通过派生类对象调用基类函数
  - 将派生类指针指向基类对象
  - 通过基类指针调用派生类成员函数
  - 使用虚拟函数展示多态
    - ✓ 基类指针指向派生类对象

3 Relationships Among Objects in an Inheritance Hierarchy

- 关键概念
  - 一个派生类对象可以看作是其基类的对象

即派生类对象当做基类对象使用

4 Invoking Base-Class Functions from Derived-Class Objects（从派生类对象调用基类函数）

- 基类指针指向基类对象
  - 调用基类函数
- 派生类指针指向派生类对象
  - 调用派生类函数

4 Invoking Base-Class Functions from Derived-Class Objects

- 基类指针指向派生类对象
  - 派生类对象是基类对象
  - 调用基类函数
    - ✓ 函数调用依赖于调用的句柄类型，不依赖于句柄所指向的对象类型
  - 虚拟函数（virtual functions）
    - ✓ 使得函数调用依赖于句柄所指向的对象类型成为可能
    - ✓ 对于实现多态行为至关重要

```
1  // Fig. 13.1: CommissionEmployee.h
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
```

```
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    double earnings() const; // calculate earnings
31    void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
```

```
1  // Fig. 13.3: BasePlusCommissionEmployee.h
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    double earnings() const; // calculate earnings
22    void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
```

```
1  // Fig. 13.5: fig13_05.cpp
2  // Aiming base-class and derived-class pointers at base-class
3  // and derived-class objects, respectively.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10 using std::setprecision;
11
12 // include class definitions
13 #include "CommissionEmployee.h"
14 #include "BasePlusCommissionEmployee.h"
15
16 int main()
17 {
18    // create base-class object  实例化基类对象
19    CommissionEmployee commissionEmployee(
20       "Sue", "Jones", "222-22-2222", 10000, .06 );
21
22    // create base-class pointer  创建基类指针
23    CommissionEmployee *commissionEmployeePtr = 0;
```

```
24
25    // create derived-class object    实例化派生类对象
26    BasePlusCommissionEmployee basePlusCommissionEmployee(
27       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
28
29    // create derived-class pointer    创建实例化指针
30    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
31
32    // set floating-point output formatting
33    cout << fixed << setprecision( 2 );
34
35    // output objects commissionEmployee and basePlusCommissionEmployee
36    cout << "Print base-class and derived-class objects:\n\n";
37    commissionEmployee.print(); // invokes base-class print
38    cout << "\n\n";
39    basePlusCommissionEmployee.print(); // invokes derived-class print
40
41    // aim base-class pointer at base-class object and print
42    commissionEmployeePtr = &commissionEmployee; // perfectly natural
43    cout << "\n\n\nCalling print with base-class pointer to "
44       << "\nbase-class object invokes base-class print function:\n\n";
45    commissionEmployeePtr->print(); // invokes base-clas
```

当编译器编译到上面语句的时候，会参照该对象指针所属的类名和函数名去标识符表中查找该成员函数的首地址，查找到所属类的该成员函数的首地址，并使用call xxxxx语句替换该函数调用语句

Aiming base-class pointer at base-class object and invoking base-class functionality

将基类对象的地址赋给了基类指针，并利用该指针调用了基类的对象的成员函数

```
46
47    // aim derived-class pointer at derived-class object and print
48    basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
49    cout << "\n\n\nCalling print with derived-class pointer to "
50       << "\nderived-class object invokes derived-class "
51       << "print function:\n\n";
52    basePlusCommissionEmployeePtr->print(); // invokes derived-class print
53
54    // aim base-class pointer at derived-class object and print
55    commissionEmployeePtr = &basePlusCommissionEmployee;
56    cout << "\n\n\nCalling print with base-class pointer to "
57       << "derived-class object\ninvokes base-class print "
58       << "function on that derived-class object:\n\n";
59    commissionEmployeePtr->print(); // invokes base-class print
60    cout << endl;
61    return 0;
62 } // end main
```

将派生类对象的地址赋给了派生类指针，并利用该指针调用了派生类对象的成员函数

Aiming derived-class pointer at derived-class object and invoking derived-class functionality

Aiming base-class pointer at derived-class object and invoking base-class functionality

将派生类对象的地址赋给了基类指针，并利用该指针调用了基类对象的成员函数

”交叉赋值“——每个派生类对象都是一个基类对象

```
Print base-class and derived-class objects:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Calling print with base-class pointer to
base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06
```

(Continued at top of next slide…)

(…Continued from bottom of previous slide)

```
Calling print with derived-class pointer to
derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Calling print with base-class pointer to derived-class object
invokes base-class print function on that derived-class object:

commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
```

赋值兼容规则

赋值兼容规则是指在需要基类对象的任何地方都可以使用公有派生类的对象来代替。通过公有继承，派生类得到了基类中除了构造函数、析构函数之外的所有公有成员，而且所有这些成员的访问控制属性也和基类完全相同。这样公有派生类实际上就具备了基类的所有功能。凡是基类能解决的问题，公有派生类都可以解决。

当派生类为公有继承基类时，允许以下四种情况的赋值 (这些称为赋值兼容规则)：

- 规则1　可以用派生类对象为基类对象赋值。
- 规则2　可以用派生类对象初始化基类引用对象。
- 规则3　可以把指向派生类对象的指针赋给基类对象的指针。
- 规则4　可以把派生类对象的地址赋给基类对象的指针

在替代之后，派生类对象就可以作为基类的对象使用，但只能使用从基类继承过来的成员。

如果声明B为基类，D为B的公有派生类，b1为B类的对象，pb1为B类的指针，d1为D类的对象。

```
class B
{ ..... };
class D: public B
{ ......};
B b1, *pb1;
D d1;
```

这时：

①派生类对象可以赋值给基类对象，即用派生类对象中从基类继承来的成员，逐个赋值给基类对象的成员：

```
b1=d1;
```

②派生类的对象可以初始化基类对象的引用：

```
B  &bb=d1;
```

③派生类对象的地址可以赋给基类类型的指针：

```
pb1=&d1;
```

赋值兼容规则3和规则4在程序设计中用途很广，这主要体现在：

(1) 赋值兼容规则为通用化程序设计打开了便捷之门。例如，如果要设计一个函数，其参数适合一个类层次中的任何一个类，就可以把该函数的参数类型设计成该类层次的基类类型指针或基类类型引用。这样，当实参为该类层次中的任何一个类类型时，都因满足赋值兼容规则而不存在类型不匹配问题。

(2) 把赋值兼容规则3或规则4与后边讨论的虚函数方法相结合，可以实现面向对象程序运行时的多态性。运行时的多态性能提供很大的设计灵活性。

- 这种将派生类对象、指针或引用转变为基类的对象、指针或引用的过程称为向上映射。
- 向上映射不需要作显式的说明，或强制的转换。
  - 是安全的
  - 是从特殊的类型转换为一般的类型
  - 出现的最坏情况是失去类成员，而不会增加类成员。

- 由于向上映射是将特殊的类型转换成了一般的类型，对象的类型信息就损失掉了。

本例小结

- 调用函数的句柄（即指针或引用）类型决定了所调用函数的功能（基类或派生类的同名函数），而不是由句柄所指向的对象的类型决定的。

5 Aiming Derived-Class Pointers at Base-Class Objects

- 派生类指针指向基类对象
  - C++ 编译器产生错误
    - ✓ CommissionEmployee 不是 BasePlusCommissionEmployee
  - 如果允许这样做，程序员会尝试访问实际上不存在的派生类成员
  - 由于派生类具有对应每个基类成员的成员，所以把派生类的对象赋值给基类对象（或者是派生类对象的地址赋值给基类指针）是合理的。（即：基类指针可以指向派生类对象，但反过来，派生类指针不可以指向基类对象）

- 定义了两个对象，一个是基类对象，另一个是派生类对象：

```
twoD two(15.0, 15.0);
threeD three(20.0, 20.0, 30.0);
```

- 派生类对象three同时也是基类对象two，派生类对象允许给基类对象赋值：

```
two = three;
```

对象two比对象three少一个数据成员z，这个赋值的效果是对象two收到对象three的x和y坐标值。

- 反过来，用基类对象对派生类对象赋值是不行的。例如：

```
three = two;
```

由于对象two没有z数据成员，使得对象three的数据成员z的值是不定的。所以，**C++编译器不允许基类对象对派生类对象赋值**。

```
1  // Fig. 13.6: fig13_06.cpp
2  // Aiming a derived-class pointer at a base-class object.
3  #include "CommissionEmployee.h"
4  #include "BasePlusCommissionEmployee.h"
5
6  int main()
7  {
8     CommissionEmployee commissionEmployee(
9        "Sue", "Jones", "222-22-2222", 10000, .06 );
10    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
11
12    // aim derived-class pointer at base-class object
13    // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
14    basePlusCommissionEmployeePtr = &commissionEmployee;
15    return 0;
16 } // end main
```

Cannot assign base-class object to derived-class pointer because *is-a* relationship does not apply

- 基类没有提供派生类的成员函数setBaseSalary，也没提供数据成员baseSalary
  - ➢可能覆盖内存中其他对象的重要数据。

Borland C++ command-line compiler error messages:

```
Error E2034 Fig13_06\fig13_06.cpp 14: Cannot convert 'CommissionEmployee *'
   to 'BasePlusCommissionEmployee *' in function main()
```

GNU C++ compiler error messages:

```
fig13_06.cpp:14: error: invalid conversion from `CommissionEmployee*' to
   `BasePlusCommissionEmployee*'
```

Microsoft Visual C++.NET compiler error messages:

```
C:\cpphtp5_examples\ch13\Fig13_06\fig13_06.cpp(14) : error C2440:
   '=' : cannot convert from 'CommissionEmployee *__w64 ' to
   'BasePlusCommissionEmployee *'
         Cast from base to derived requires dynamic_cast or static_cast
```

6 Derived-Class Member-Function Calls via Base-Class Pointers

- 基类指针指向派生类对象
  - 调用存在于基类的函数将会指向基类的函数
  - 调用基类不存在的函数将会导致错误
    - ✓ 派生类成员不能通过基类指针来访问（因为基类指针所能访问的只是基类成员）
    - ✓ 可以通过 downcasting（向下强制类型转换） 来完成

```cpp
// Fig. 13.7: fig13_07.cpp
// Attempting to invoke derived-class-only member functions
// through a base-class pointer.
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
   CommissionEmployee *commissionEmployeePtr = 0; // base class
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // derived class

   // aim base-class pointer at derived-class object
   commissionEmployeePtr = &basePlusCommissionEmployee;

   // invoke base-class member functions on derived-class
   // object through base-class pointer
   string firstName = commissionEmployeePtr->getFirstName();
   string lastName = commissionEmployeePtr->getLastName();
   string ssn = commissionEmployeePtr->getSocialSecurityNumber();
   double grossSales = commissionEmployeePtr->getGrossSales();
   double commissionRate = commissionEmployeePtr->getCommissionRate();

   // attempt to invoke derived-class-only member functions
   // on derived-class object through base-class pointer
   double baseSalary = commissionEmployeePtr->getBaseSalary();
   commissionEmployeePtr->setBaseSalary( 500 );
   return 0;
} // end main
```

Cannot invoke derived-class-only members from base-class pointer

基类指针只能调用与它关联的基类的成员函数

Borland C++ command-line compiler error messages:

```
Error E2316 Fig13_07\fig13_07.cpp 26: 'getBaseSalary' is not a member of
   'CommissionEmployee' in function main()
Error E2316 Fig13_07\fig13_07.cpp 27: 'setBaseSalary' is not a member of
   'CommissionEmployee' in function main()
```

Microsoft Visual C++.NET compiler error messages:

```
C:\cpphtp5_examples\ch13\Fig13_07\fig13_07.cpp(26) : error C2039:
   'getBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp5_examples\ch13\Fig13_07\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
C:\cpphtp5_examples\ch13\Fig13_07\fig13_07.cpp(27) : error C2039:
   'setBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp5_examples\ch13\Fig13_07\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
```

GNU C++ compiler error messages:

```
fig13_07.cpp:26: error: `getBaseSalary' undeclared (first use this function)
fig13_07.cpp:26: error: (Each undeclared identifier is reported only once for
   each function it appears in.)
fig13_07.cpp:27: error: `setBaseSalary' undeclared (first use this function)
```

- 当然，也可以显式将基类指针强制转换为派生类指针（向下强制类型转换downcasting）
  - 允许通过指向派生类对象的基类指针访问只在派生类中拥有的成员

(1) 静态联编工作是在编译连接阶段完成的。在编译、连接过程中，系统根据类型匹配等特征确定程序中操作调用与执行该操作代码的关系，即确定某一个同名标识符到底要调用哪一段程序代码。

优点：执行速度快

(2) 动态联编是联编工作是在程序运行阶段完成。

在编译连接过程中无法解决的联编问题，要等到程序开始运行之后才能来确定。动态联编实际上是进行动态识别，也就是说，要等到程序运行时，确定了指针所指向的对象的类型时，才能够确定。

- 假定一组形状类(如Circle、Trriangle、Rectangle和Square等)都是从基类Shape派生出来的。每个类都有自己draw函数，能够绘制其自身形状
  - 绘制每种形状的draw函数是大不相同的。
- 绘制任何形状，把它作为基类Shape的对象处理统一处理
  - 调用基类Shape的函数draw
  - 让程序动态地确定(即在执行时确定)使用哪个派生类的draw函数
- 那么如何来确定是用静态联编还是动态联编？
- C++规定动态联编是在虚函数的支持下实现的

7 Virtual Functions

- 因此，哪个函数被调用？
  - 在非虚函数中，由句柄决定能够调用的函数
  - 在虚拟函数中（ virtual functions ）
    - ✓是由句柄所指向的对象类型，而不是由句柄的类型决定了函数的哪个版本被调用
    - ✓允许程序动态决定使用哪个函数
      - ◇称为动态绑定或晚绑定

7 Virtual Functions

- 虚拟函数
  - 在基类的成员函数原型前加上 virtual 关键字

    virtual <函数类型> <函数名>(形参表){ [函数体] }
  - 派生类用自己的方式重写该函数
  - 一旦声明为 virtual，该函数在向下的层次中均为 virtual

7 Virtual Functions

- 虚拟函数
  - 静态绑定
    - ✓当使用对象通过点运算符访问虚拟函数，函数调用将在编译阶段被解析
  - 动态绑定
    - ✓动态绑定出现在使用指针或引用作为句柄时

```
1  // Fig. 13.8: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
```

```
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    virtual double earnings() const; // calculate earnings
31    virtual void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

Declaring **earnings** and **print** as **virtual** allows them to be overridden, not redefined

```
1  // Fig. 13.9: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif
```

Functions **earnings** and **print** are already **virtual** – good practice to declare **virtual** even when overriding function

```
1  // Fig. 13.10: fig13_10.cpp
2  // Introducing polymorphism, virtual functions and dynamic binding.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 // include class definitions
12 #include "CommissionEmployee.h"
13 #include "BasePlusCommissionEmployee.h"
14
15 int main()
16 {
17    // create base-class object
18    CommissionEmployee commissionEmployee(
19       "Sue", "Jones", "222-22-2222", 10000, .06 );
20
21    // create base-class pointer
22    CommissionEmployee *commissionEmployeePtr = 0;
23
24    // create derived-class object
25    BasePlusCommissionEmployee basePlusCommissionEmployee(
26       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
27
28    // create derived-class pointer
29    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
```

```
30
31    // set floating-point output formatting
32    cout << fixed << setprecision( 2 );
33
34    // output objects using static binding
35    cout << "Invoking print function on base-class and derived-class "
36       << "\nobjects with static binding\n\n";
37    commissionEmployee.print(); // static binding
38    cout << "\n\n";
39    basePlusCommissionEmployee.print(); // static binding
40
41    // output objects using dynamic binding
42    cout << "\n\n\nInvoking print function on base-class and "
43       << "derived-class \nobjects with dynamic binding";
44
45    // aim base-class pointer at base-class object and print
46    commissionEmployeePtr = &commissionEmployee;
47    cout << "\n\nCalling virtual function print with base-class pointer"
48       << "\nto base-class object invokes base-class "
49       << "print function:\n\n";
50    commissionEmployeePtr->print(); // invokes base-class print
```

指向基类对象的基类指针调用基类成员函数

```
51
52    // aim derived-class pointer at derived-class object and print
53    basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
54    cout << "\n\nCalling virtual function print with derived-class "
55       << "pointer\nto derived-class object invokes derived-class "
56       << "print function:\n\n";
57    basePlusCommissionEmployeePtr->print(); // invokes derived-class print
58
59    // aim base-class pointer at derived-class object and print
60    commissionEmployeePtr = &basePlusCommissionEmployee;
61    cout << "\n\nCalling virtual function print with base-class pointer"
62       << "\nto derived-class object invokes derived-class "
63       << "print function:\n\n";
64
65    // polymorphism; invokes BasePlusCommissionEmployee's print;
66    // base-class pointer to derived-class object
67    commissionEmployeePtr->print();
68    cout << endl;
69    return 0;
70 } // end main
```

指向派生对象的派生类指针调用派生类成员函数

基类指针指向派生类对象，调用了派生类的print成员函数

via polymorphism and **virtual** functions

因为使用了 virtual functions，句柄所指向的对象的类型而不是句柄的类型决定了调用的成员函数归属

```
Invoking print function on base-class and derived-class
objects with static binding

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Invoking print function on base-class and derived-class
objects with dynamic binding

Calling virtual function print with base-class pointer
to base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Calling virtual function print with derived-class pointer
to derived-class object invokes derived-class print function:
```

(Coninued at the top of next slide …)

(…Continued from the bottom of previous slide)

```
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Calling virtual function print with base-class pointer
to derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

8 Summary of the Allowed Assignments Between Base-Class and Derived-Class Objects and Pointers

- 四种用来将基类和派生类指针指向基类和派生类对象的方式
  - 基类指针指向基类对象
  - 派生类指针指向派生类对象
  - 基类指针指向派生类对象
    - ✓ 安全，但只能用来调用基类的成员函数
    - ✓ 如果使用虚拟函数可以实现多态
  - 派生类指针指向基类对象
    - ✓ 产生编译错误

软件工程知识：利用虚拟函数和多态性，程序员可以处理普遍性而让执行环境处理特殊性。即使在不知道一些对象的类型的情况下（只要这些对象属于同一继承层次并且通过一个共同的基类指针访问），程序员也可以命令各种对象表现出适合这些对象的行为。

软件工程知识：多态性提高了可扩展性：调用多态性行为的软件可以用与接收消息的对象类型无关的方式编写。因此，不用修改基本系统就可以把响应现有消息的对象的新类型添加到该系统中。只有实例化新对象的客户代码必须修改，以适应新的类型。

9 Type Fields and switch Statements

- switch 语句用来在运行期间决定对象类型
  - 在基类中包含一个类型域数据成员
  - 使得程序员对于特定对象调用合适的函数
  - 引发的问题
    - ✓ 程序员可能会忘记应有的类型测试；
    - ✓ 在一条switch语句中可能会忘记测试所有可能的情况；
    - ✓ 在修改基于switch语句的系统时可能会忘记在现有的switch语句中插入新类；
    - ✓ 为了处理新的类型，每次修改switch语句都要修改系统中的每一条switch语句，这很费时并且容易出错。

- 利用了虚函数和多态性的程序设计无需使用switch逻辑。程序员可以用虚函数机制自动完成等价的逻辑，因而避免与switch逻辑有关的各种各样的错误。
- 使用虚函数和多态性可简化源代码的长度。为支持更简单的顺序代码，虚函数和多态性包含的分支逻辑更少。这种简化有助于程序的测试、调试和维护。

10 Abstract Classes and Pure virtual Functions（抽象类与纯虚函数）

- 抽象类
  - 有时需要定义那些永远不需要实例化的类
    - ✓ 作为基类时信息不完整—需要由派生类来定义“缺失的部分”
  - 抽象类的惟一用途是为其他类提供合适的基类，其他类可从它这里继承和(或)实现接口。
    - ✓ 可以实例化的类称为具体类（concrete classes）
      - ◈ 必须提供每个成员函数的实现

10 Abstract Classes and Pure virtual Functions

- 纯虚函数（Pure virtual function）
  - 抽象类包含一个或多个纯虚函数
    - ✓ 在声明后加“= 0”表示纯虚函数
      - ◈例如：virtual void draw() const = 0;

10 Abstract Classes and Pure virtual Functions

- 纯虚函数（Pure virtual function）
  - 不提供实现
    - ✓ 每个具体的派生类必须覆盖基类的纯虚函数来提供具体的实现
      - ◈ 如果不覆盖，派生类也将称为抽象类
  - 当在基类提供成员函数的实现没有意义时，且要求具体派生类实现这样的函数时，可以采用纯虚函数

```
class CPolygon {
protected:
  int width, height;
public:
  void set_values (int a, int b)
    { width=a; height=b; }
  virtual int area (void) =0;
};
```

```
class CTriangle: public CPolygon
{
public:
  int area (void)
    { return (width * height / 2); }
};
```

```
class CRectangle: public CPolygon
{
public:
  int area (void)
    { return (width * height); }
};
```

在C++中，还有一种情况是空的虚函数，空的虚函数是指函数体为空的虚函数，请注意它和纯虚函数的区别。纯虚函数根本就没有函数体，而空的虚函数的函数体为空（如：{ }）。

纯虚函数所在的类是抽象类，不能直接进行实例化（不能定义对象），而虚函数所在的类是可以实例化的。它们共同的特点是都可以派生出新的类，然后在新类中给出虚函数的实现，而且这种新的实现可以具有多态特征。

续

- 虚函数为了重载和多态的需要，在基类中是有定义的，即便定义是空，所以子类中可以重写也可以不重写基类中的此函数！
- 纯虚函数在基类中是没有定义的，必须在子类中加以实现。

10 Abstract Classes and Pure virtual Functions

软件工程知识：抽象类为类层次结构中的各个成员定义公共接口。抽象类通常包含一个或多个在派生类中必须重写的纯虚拟函数。

软件工程知识：抽象类必须至少有一个纯虚函数。抽象类也可以有数据成员和具体函数（包括构造函数和析构函数），它们遵从派生类继承的通常规则。

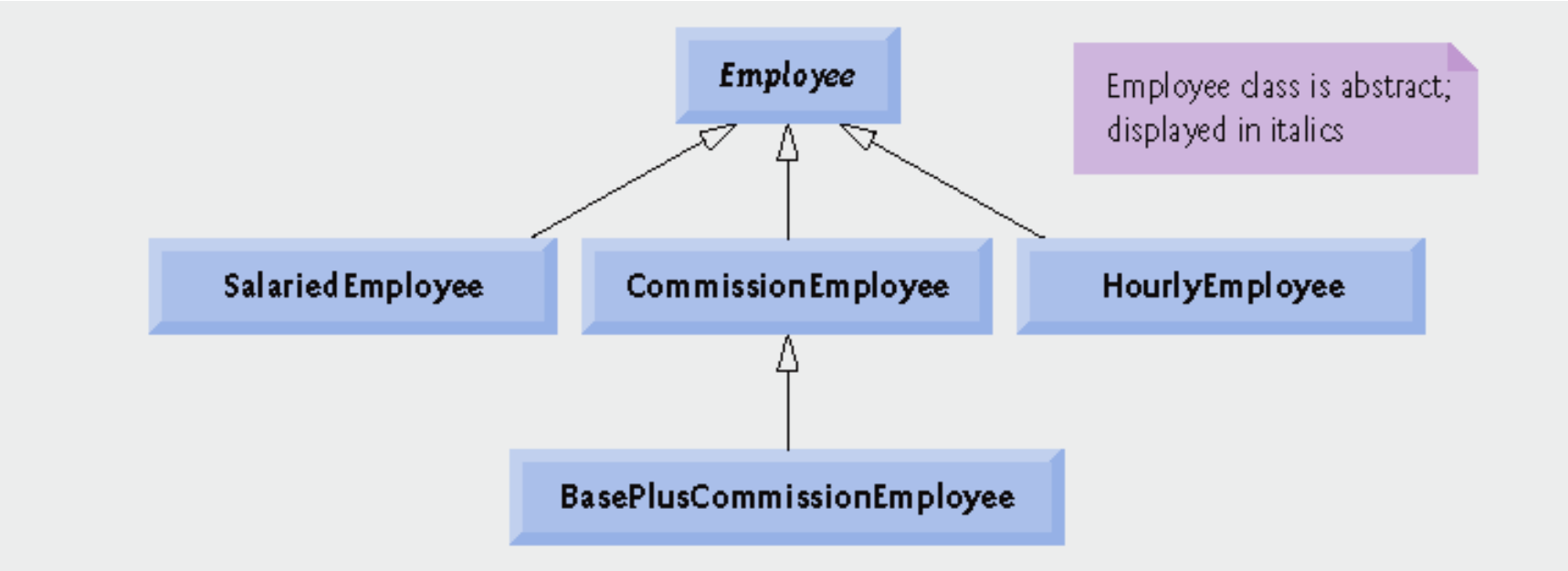
10 Abstract Classes and Pure virtual Functions

- 可以使用抽象类来声明指针和引用
  - 可以引用任何从该抽象类继承而来的具体类
  - 程序员通常使用这样的指针和引用来操纵派生类对象（多态）
- 多态尤其适合实现层次性的软件系统
- 迭代类
  - 可以遍历容器内的每个对象

11 Case Study: Payroll System Using Polymorphism (实例：应用多态性的工资发放系统)

- 使用抽象类的 CommissionEmployee-BasePlusCommissionEmployee 继承层次
  - 抽象类 Employee 代表雇员的一般概念
    - ✓ 声明该层次的接口
    - ✓ 每个雇员有 first name, last name 和 social security number(ssn)
  - Earnings 和 print 处理不同的派生类对象

11 Case Study: Payroll System Using Polymorphism



11 Case Study: Payroll System Using Polymorphism

软件工程知识：派生类可以从基类继承接口或实现。为“实现继承”而设计的类层次结构往往将功能设置在较高层，即每个新派生类继承定义在基类中的一个或多个成员函数，并且派生类使用这些基类定义；而为“接口继承”设计的类层次结构则趋于将功能设置在较低层，即基类指定一个或多个应为类继承层次中的每个类定义的函数（即它们有相同的原型），但是各个派生类提供自己对于这些函数的实现。

12 Creating Abstract Base Class Employee

- Class Employee
  - 提供 *get* 和 *set* 函数
  - 提供 earnings 和 print 函数
    - ✓earnings 依赖于 employee 类型，所以声明为纯虚函数
    - ✓print 是虚函数，但不是纯虚函数

12 Creating Abstract Base Class Employee

- Class Employee
  - 例子中维护一个 Employee 指针 vector
    - ✓多态地调用正确的 earnings 和 print 函数

	earnings	print
Employee	= 0	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* (40 * wage) + ((hours - 40) * wage * 1.5)	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage*; hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	baseSalary + (commissionRate * grossSales)	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

```
1  // Fig. 13.13: Employee.h
2  // Employee abstract base class.
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class Employee
10 {
11 public:
12    Employee( const string &, const string &, const string & );
13    // First Name , Last Name , SSN
14    void setFirstName( const string & ); // set first name
15    string getFirstName() const; // return first name
16
17    void setLastName( const string & ); // set last name
18    string getLastName() const; // return last name
19
20    void setSocialSecurityNumber( const string & ); // set SSN
21    string getSocialSecurityNumber() const; // return SSN
```

```
22
23    // pure virtual function makes Employee abstract base class
24    virtual double earnings() const = 0; // pure virtual
25    virtual void print() const; // virtual
26 private:
27    string firstName;
28    string lastName;
29    string socialSecurityNumber;
30 }; // end class Employee
31
32 #endif // EMPLOYEE_H
```

Function **earnings** is pure **virtual**, not enough data to provide a default, concrete implementation

Function **print** is **virtual**, default implementation provided but derived-classes may override

```cpp
// Fig. 13.14: Employee.cpp
// Abstract-base-class Employee member-function definitions.
// Note: No definitions are given for pure virtual functions.
#include <iostream>
using std::cout;

#include "Employee.h" // Employee class definition

// constructor
Employee::Employee( const string &first, const string &last,
   const string &ssn )
   : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
{
   // empty body
} // end Employee constructor

// set first name
void Employee::setFirstName( const string &first )
{
   firstName = first;
} // end function setFirstName

// return first name
string Employee::getFirstName() const
{
   return firstName;
} // end function getFirstName

```

```
29 // set last name
30 void Employee::setLastName( const string &last )
31 {
32    lastName = last;
33 } // end function setLastName
34
35 // return last name
36 string Employee::getLastName() const
37 {
38    return lastName;
39 } // end function getLastName
40
41 // set social security number
42 void Employee::setSocialSecurityNumber( const string &ssn )
43 {
44    socialSecurityNumber = ssn; // should validate
45 } // end function setSocialSecurityNumber
46
47 // return social security number
48 string Employee::getSocialSecurityNumber() const
49 {
50    return socialSecurityNumber;
51 } // end function getSocialSecurityNumber
52
53 // print Employee's information (virtual, but not pure virtual)
54 void Employee::print() const
55 {
56    cout << getFirstName() << ' ' << getLastName()
57       << "\nsocial security number: " << getSocialSecurityNumber();
58 } // end function print
```

13 Creating Concrete Derived Class SalariedEmployee

- SalariedEmployee 继承自 Employee
  - 包含 weekly salary
    - ✓覆盖 earnings 函数以包含 weekly salary
    - ✓覆盖 print 函数以包含 weekly salary
  - 是一个具体类（实现了抽象基类中所有的纯虚函数）

```
1  // Fig. 13.15: SalariedEmployee.h
2  // SalariedEmployee class derived from Employee.
3  #ifndef SALARIED_H
4  #define SALARIED_H
5
6  #include "Employee.h" // Employee class definition
7
8  class SalariedEmployee : public Employee
9  {
10 public:
11    SalariedEmployee( const string &, const string &,
12       const string &, double = 0.0 );
13
14    void setWeeklySalary( double ); // set weekly salary
15    double getWeeklySalary() const; // return weekly salary
16
17    // keyword virtual signals intent to override
18    virtual double earnings() const; // calculate earnings
19    virtual void print() const; // print SalariedEmployee object
20 private:
21    double weeklySalary; // salary per week
22 }; // end class SalariedEmployee
23
24 #endif // SALARIED_H
```

SalariedEmployee inherits from **Employee**, must override **earnings** to be concrete

Functions will be overridden (or defined for the first time)

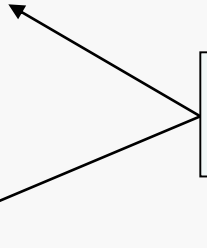
```cpp
// Fig. 13.16: SalariedEmployee.cpp
// SalariedEmployee class member-function definitions.
#include <iostream>
using std::cout;

#include "SalariedEmployee.h" // SalariedEmployee class definition

// constructor
SalariedEmployee::SalariedEmployee( const string &first,
   const string &last, const string &ssn, double salary )
   : Employee( first, last, ssn )
{
   setWeeklySalary( salary );
} // end SalariedEmployee constructor

// set salary
void SalariedEmployee::setWeeklySalary( double salary )
{
   weeklySalary = ( salary < 0.0 ) ? 0.0 : salary;
} // end function setWeeklySalary

// return salary
double SalariedEmployee::getWeeklySalary() const
{
   return weeklySalary;
} // end function getWeeklySalary
```



Maintain new data member **weeklySalary**

```
27
28 // calculate earnings;
29 // override pure virtual function earnings in Employee
30 double SalariedEmployee::earnings() const
31 {
32    return getWeeklySalary();
33 } // end function earnings
34
35 // print SalariedEmployee's information
36 void SalariedEmployee::print() const
37 {
38    cout << "salaried employee: ";
39    Employee::print(); // reuse abstract base-class print function
40    cout << "\nweekly salary: " << getWeeklySalary();
41 } // end function print
```

Overridden `earnings` and `print` functions incorporate weekly salary

14 Creating Concrete Derived Class HourlyEmployee

- HourlyEmployee 继承自 Employee
  - 包含工资和工作的小时数
    - ✓ 覆盖 `earnings` 函数以包含雇员的工资乘以小时数
    - ✓ 覆盖 `print` 函数以包含工资和工作的小时数
  - 是一个具体类（实现了抽象基类中所有的纯虚函数）

```
1  // Fig. 13.17: HourlyEmployee.h
2  // HourlyEmployee class definition.
3  #ifndef HOURLY_H
4  #define HOURLY_H
5
6  #include "Employee.h" // Employee class definition
7
8  class HourlyEmployee : public Employee
9  {
10 public:
11    HourlyEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0 );
13
14    void setWage( double ); // set hourly wage
15    double getWage() const; // return hourly wage
16
17    void setHours( double ); // set hours worked
18    double getHours() const; // return hours worked
19
20    // keyword virtual signals intent to override
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print HourlyEmployee object
23 private:
24    double wage; // wage per hour
25    double hours; // hours worked for week
26 }; // end class HourlyEmployee
27
28 #endif // HOURLY_H
```

HourlyEmployee inherits from **Employee**, must override **earnings** to be concrete

Functions will be overridden (or defined for first time)

```cpp
// Fig. 13.18: HourlyEmployee.cpp
// HourlyEmployee class member-function definitions.
#include <iostream>
using std::cout;

#include "HourlyEmployee.h" // HourlyEmployee class definition

// constructor
HourlyEmployee::HourlyEmployee( const string &first, const string &last,
   const string &ssn, double hourlyWage, double hoursWorked )
   : Employee( first, last, ssn )
{
   setWage( hourlyWage ); // validate hourly wage
   setHours( hoursWorked ); // validate hours worked
} // end HourlyEmployee constructor

// set wage
void HourlyEmployee::setWage( double hourlyWage )
{
   wage = ( hourlyWage < 0.0 ? 0.0 : hourlyWage );
} // end function setWage

// return wage
double HourlyEmployee::getWage() const
{
   return wage;
} // end function getWage
```

Maintain new data member, **hourlyWage**

```
28
29 // set hours worked
30 void HourlyEmployee::setHours( double hoursWorked )
31 {
32    hours = ( ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
33       hoursWorked : 0.0 );
34 } // end function setHours
35
36 // return hours worked
37 double HourlyEmployee::getHours() const
38 {
39    return hours;
40 } // end function getHours
41
42 // calculate earnings;
43 // override pure virtual function earnings in Employee
44 double HourlyEmployee::earnings() const
45 {
46    if ( getHours() <= 40 ) // no overtime
47       return getWage() * getHours();
48    else
49       return 40 * getWage() + ( ( getHours() - 40 ) * getWage() * 1.5 );
50 } // end function earnings
51
52 // print HourlyEmployee's information
53 void HourlyEmployee::print() const
54 {
55    cout << "hourly employee: ";
56    Employee::print(); // code reuse
57    cout << "\nhourly wage: " << getWage() <<
58       "; hours worked: " << getHours();
59 } // end function print
```

Maintain new data member, **hoursWorked**

Overridden **earnings** and **print** functions incorporate wage and hours

15 Creating Concrete Derived Class CommissionEmployee

- CommissionEmployee 继承自 Employee
  - 包含总销售额和提成比率
    - ✓ 覆盖 earnings 函数以包含 gross sales 和 commission rate
    - ✓ 覆盖 print 函数以包含 gross sales 和 commission rate
  - 具体类（实现了抽象基类的所有纯虚函数）

```
1  // Fig. 13.19: CommissionEmployee.h
2  // CommissionEmployee class derived from Employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include "Employee.h" // Employee class definition
7
8  class CommissionEmployee : public Employee
9  {
10 public:
11    CommissionEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0 );
13
14    void setCommissionRate( double ); // set commission rate
15    double getCommissionRate() const; // return commission rate
16
17    void setGrossSales( double ); // set gross sales amount
18    double getGrossSales() const; // return gross sales amount
19
20    // keyword virtual signals intent to override
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print CommissionEmployee object
23 private:
24    double grossSales; // gross weekly sales
25    double commissionRate; // commission percentage
26 }; // end class CommissionEmployee
27
28 #endif // COMMISSION_H
```

CommissionEmployee inherits from **Employee**, must override **earnings** to be concrete

Functions will be overridden (or defined for first time)

```cpp
// Fig. 13.20: CommissionEmployee.cpp
// CommissionEmployee class member-function definitions.
#include <iostream>
using std::cout;

#include "CommissionEmployee.h" // CommissionEmployee class definition

// constructor
CommissionEmployee::CommissionEmployee( const string &first,
   const string &last, const string &ssn, double sales, double rate )
   : Employee( first, last, ssn )
{
   setGrossSales( sales );
   setCommissionRate( rate );
} // end CommissionEmployee constructor

// set commission rate
void CommissionEmployee::setCommissionRate( double rate )
{
   commissionRate = ( ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0 );
} // end function setCommissionRate

// return commission rate
double CommissionEmployee::getCommissionRate() const
{
    return commissionRate;
} // end function getCommissionRate
```

Maintain new data member, **commissionRate**

```
28
29 // set gross sales amount
30 void CommissionEmployee::setGrossSales( double sales )
31 {
32    grossSales = ( ( sales < 0.0 ) ? 0.0 : sales );
33 } // end function setGrossSales
34
35 // return gross sales amount
36 double CommissionEmployee::getGrossSales() const
37 {
38     return grossSales;
39 } // end function getGrossSales
40
41 // calculate earnings;
42 // override pure virtual function earnings in Employee
43 double CommissionEmployee::earnings() const
44 {
45     return getCommissionRate() * getGrossSales();
46 } // end function earnings
47
48 // print CommissionEmployee's information
49 void CommissionEmployee::print() const
50 {
51    cout << "commission employee: ";
52    Employee::print(); // code reuse
53    cout << "\ngross sales: " << getGrossSales()
54       << "; commission rate: " << getCommissionRate();
55 } // end function print
```

Maintain new data member, **grossSales**

Overridden **earnings** and **print** functions incorporate commission rate and gross sales

16 Creating Indirect Concrete Derived Class BasePlusCommissionEmployee

- BasePlusCommissionEmployee 继承自 CommissionEmployee
  - 包含 base salary
    - ✓覆盖 earnings 函数以包含 base salary
    - ✓覆盖 print 函数以包含 base salary
  - 具体类，因为其直接基类为具体类

```
1  // Fig. 13.21: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from Employee.
3  #ifndef BASEPLUS_H
4  #define BASEPLUS_H
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  class BasePlusCommissionEmployee : public CommissionEmployee
9  {
10 public:
11    BasePlusCommissionEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0, double = 0.0 );
13
14    void setBaseSalary( double ); // set base salary
15    double getBaseSalary() const; // return base salary
16
17    // keyword virtual signals intent to override
18    virtual double earnings() const; // calculate earnings
19    virtual void print() const; // print BasePlusCommissionEmployee object
20 private:
21    double baseSalary; // base salary per week
22 }; // end class BasePlusCommissionEmployee
23
24 #endif // BASEPLUS_H
```

BasePlusCommissionEmployee inherits from **CommissionEmployee**, already concrete

Functions will be overridden

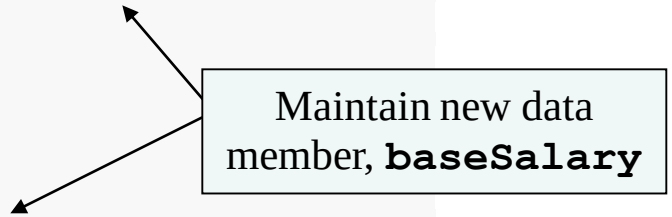
```cpp
// Fig. 13.22: BasePlusCommissionEmployee.cpp
// BasePlusCommissionEmployee member-function definitions.
#include <iostream>
using std::cout;

// BasePlusCommissionEmployee class definition
#include "BasePlusCommissionEmployee.h"

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string &first, const string &last, const string &ssn,
   double sales, double rate, double salary )
   : CommissionEmployee( first, last, ssn, sales, rate )
{
   setBaseSalary( salary ); // validate and store base salary
} // end BasePlusCommissionEmployee constructor

// set base salary
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
   baseSalary = ( ( salary < 0.0 ) ? 0.0 : salary );
} // end function setBaseSalary

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const
{
    return baseSalary;
} // end function getBaseSalary
```



```
29
30 // calculate earnings;
31 // override pure virtual function earnings in Employee
32 double BasePlusCommissionEmployee::earnings() const
33 {
34    return getBaseSalary() + CommissionEmployee::earnings();
35 } // end function earnings
36
37 // print BasePlusCommissionEmployee's information
38 void BasePlusCommissionEmployee::print() const
39 {
40    cout << "base-salaried ";
41    CommissionEmployee::print(); // code reuse
42    cout << "; base salary: " << getBaseSalary();
43 } // end function print
```

Overridden **`earnings`** and **`print`** functions incorporate base salary

17 Demonstrating Polymorphic Processing

- 创建 SalariedEmployee，HourlyEmployee，CommissionEmployeeBase 和PlusCommissionEmployee 对象
  - 演示用静态绑定来处理对象
    - ✓ 使用名字句柄而不是指针和引用
    - ✓ 编译器标识对象类型并决定调用哪个 print 和 earning 函数
  - 演示用多态来处理对象
    - ✓ 使用 Employee 指针向量
    - ✓ 使用指针或引用来调用虚拟函数

```
1  // Fig. 13.23: fig13_23.cpp
2  // Processing Employee derived-class objects individually
3  // and polymorphically using dynamic binding.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10 using std::setprecision;
11
12 #include <vector>
13 using std::vector;
14
15 // include definitions of classes in Employee hierarchy
16 #include "Employee.h"
17 #include "SalariedEmployee.h"
18 #include "HourlyEmployee.h"
19 #include "CommissionEmployee.h"
20 #include "BasePlusCommissionEmployee.h"
21
22 void virtualViaPointer( const Employee * const ); // prototype
23 void virtualViaReference( const Employee & ); // prototype
```

Vectors 是常用的顺序容器，它将单一类型的元素聚集起来成为容器，然后根据位置来存储和访问这些元素。

Vector容器可以看成是动态数组，可以动态增加元素，元素也是存放在一块连续的内存空间

```
24
25 int main()
26 {
27    // set floating-point output formatting
28    cout << fixed << setprecision( 2 );
29
30    // create derived-class objects        实例化派生类的对象
31    SalariedEmployee salariedEmployee(
32       "John", "Smith", "111-11-1111", 800 );
33    HourlyEmployee hourlyEmployee(
34       "Karen", "Price", "222-22-2222", 16.75, 40 );
35    CommissionEmployee commissionEmployee(
36       "Sue", "Jones", "333-33-3333", 10000, .06 );
37    BasePlusCommissionEmployee basePlusCommissionEmployee(
38       "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
39
40    cout << "Employees processed individually using static binding:\n\n";
41
42    // output each Employee's information and earnings using static binding
43    salariedEmployee.print();
44    cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
45    hourlyEmployee.print();
46    cout << "\nearned $" << hourlyEmployee.earnings() << "\n\n";
47    commissionEmployee.print();
48    cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
49    basePlusCommissionEmployee.print();
50    cout << "\nearned $" << basePlusCommissionEmployee.earnings()
51       << "\n\n";
```

Using objects (rather than pointers or references) to demonstrate static binding

静态绑定

```
52
53    // create vector of four base-class pointers
54    vector < Employee * > employees( 4 );
55
56    // initialize vector with Employees
57    employees[ 0 ] = &salariedEmployee;
58    employees[ 1 ] = &hourlyEmployee;
59    employees[ 2 ] = &commissionEmployee;
60    employees[ 3 ] = &basePlusCommissionEmployee;
61
62    cout << "Employees processed polymorphically via dynamic binding:\n\n";
63
64    // call virtualViaPointer to print each Employee's information
65    // and earnings using dynamic binding
66    cout << "Virtual function calls made off base-class pointers:\n\n";
67
68    for ( size_t i = 0; i < employees.size(); i++ )
69       virtualViaPointer( employees[ i ] );
70
71    // call virtualViaReference to print each Employee's information
72    // and earnings using dynamic binding
73    cout << "Virtual function calls made off base-class references:\n\n";
74
75    for ( size_t i = 0; i < employees.size(); i++ )
76       virtualViaReference( *employees[ i ] ); // note dereferencing
77
78    return 0;
79 } // end main
```

创建vector对象employees（抽象基类指针类型）

将各个具体派生类对象的地址赋值给抽象基类指针

Demonstrate dynamic binding using first pointers, then references

```
80
81 // call Employee virtual functions print and earnings off a
82 // base-class pointer using dynamic binding
83 void virtualViaPointer( const Employee * const baseClassPtr )
84 {
85    baseClassPtr->print();
86    cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
87 } // end function virtualViaPointer
88
89 // call Employee virtual functions print and earnings off a
90 // base-class reference using dynamic binding
91 void virtualViaReference( const Employee &baseClassRef )
92 {
93    baseClassRef.print();
94    cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
95 } // end function virtualViaReference
```

Using references and pointers cause **virtual** functions to be invoked polymorphically

动态绑定

- 编译时，编译器并不知道通过抽象基类指针调用哪个具体类的函数
- 执行时，每次virtual函数调用时，都会调用基类指针在那时指向的对象的函数
- 基类引用同样产生了多态性的行为
- 每次virtual函数的调用都会调用基类引用对象在执行时所引用对象的函数
- 使用基类引用和使用基类指针产生了同样的输出结果

- 几种初始化vector对象的方式
  - vector<T> v1;
    - ✓ vector 保存类型为T的对象。默认构造函数v1为空
  - vector<T> v2(v1);
    - ✓ v2 是v1的一个副本。
  - vector<T> v3(n, i);
    - ✓ v3 包含n个值为i的元素。
  - vector<T> v4(n);
    - ✓ v4 含有值初始化的元素的n个副本。

```
54      vector < Employee * > employees( 4 );
```

```
Employees processed individually using static binding:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

(Continued at top of next slide...)

(...continued from bottom of previous slide)

```
Employees processed polymorphically using dynamic binding:

Virtual function calls made off base-class pointers:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

(Continued at the top of next slide...)

(...Continued from bottom of previous page)

```
Virtual function calls made off base-class references:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00


commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

17 Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

- C++ 如何实现多态、虚拟函数和动态绑定
  - 三级指针（三级间接取值）
  - 当 C++ 编译一个带有一个或多个虚拟函数的类时，创建虚拟函数表 (*vtable*)
    - ✓ 一级指针
    - ✓ 包含指向虚拟函数的函数指针

17 Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood”

- ✓每次一个类的一个虚拟函数被调用时，用来选择正确的函数实现
- ✓如果为纯虚函数，函数指针被置 0
- ✓一个类的 *vtable* 中有一个或多个 null 指针，该类为抽象类

17 Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

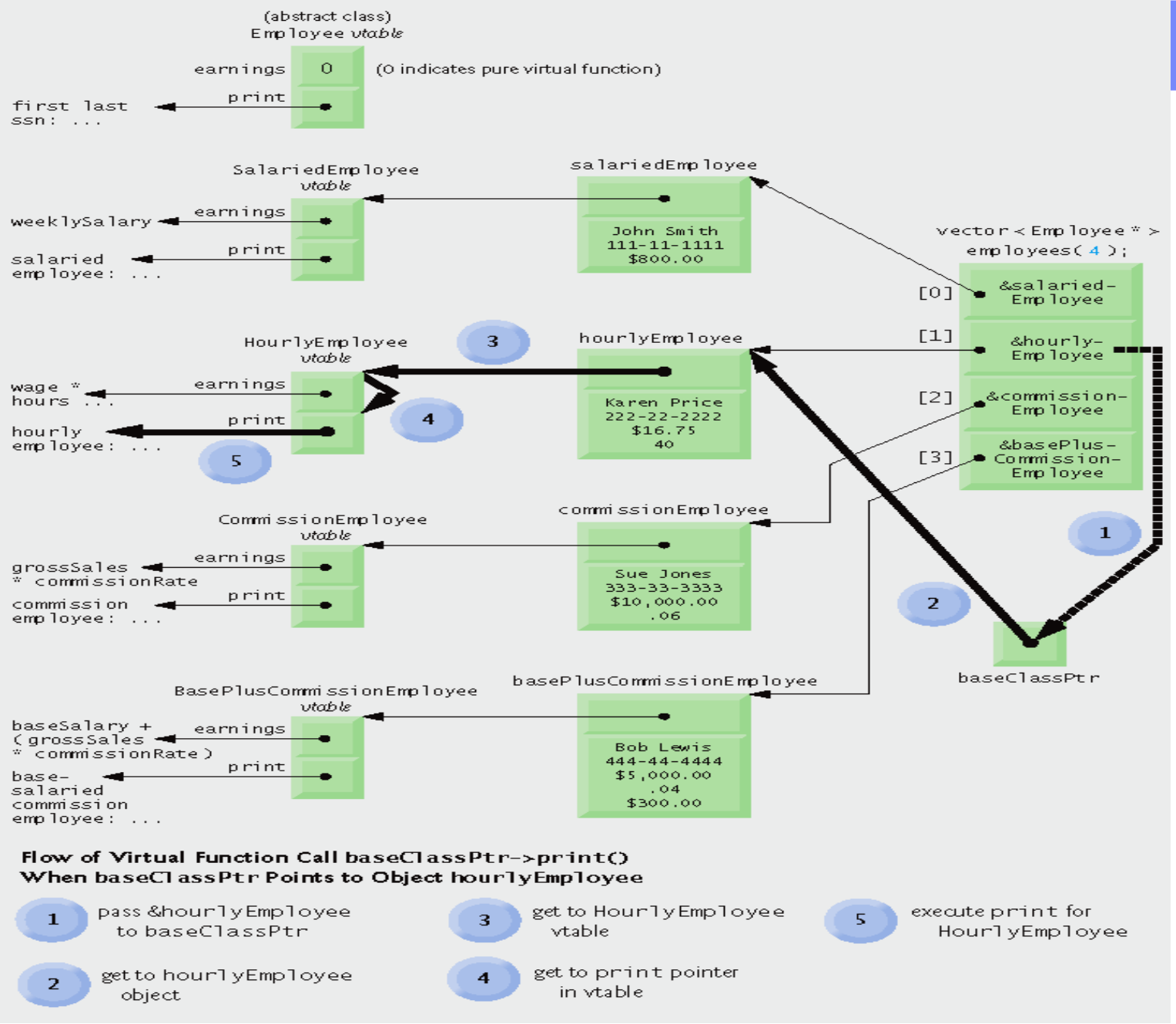
- 如果一个非纯虚函数没有被派生类所覆盖
  - ✓ 该派生类 vtable 中相应的函数指针应指向基类的虚拟函数
- 二级指针
  - ✓ 当具有一个或多个虚拟函数的类被实例化时，编译器为该对象添加一个指向该类 vtable 的指针
- 三级指针
  - ✓ 接收虚拟函数调用的对象句柄

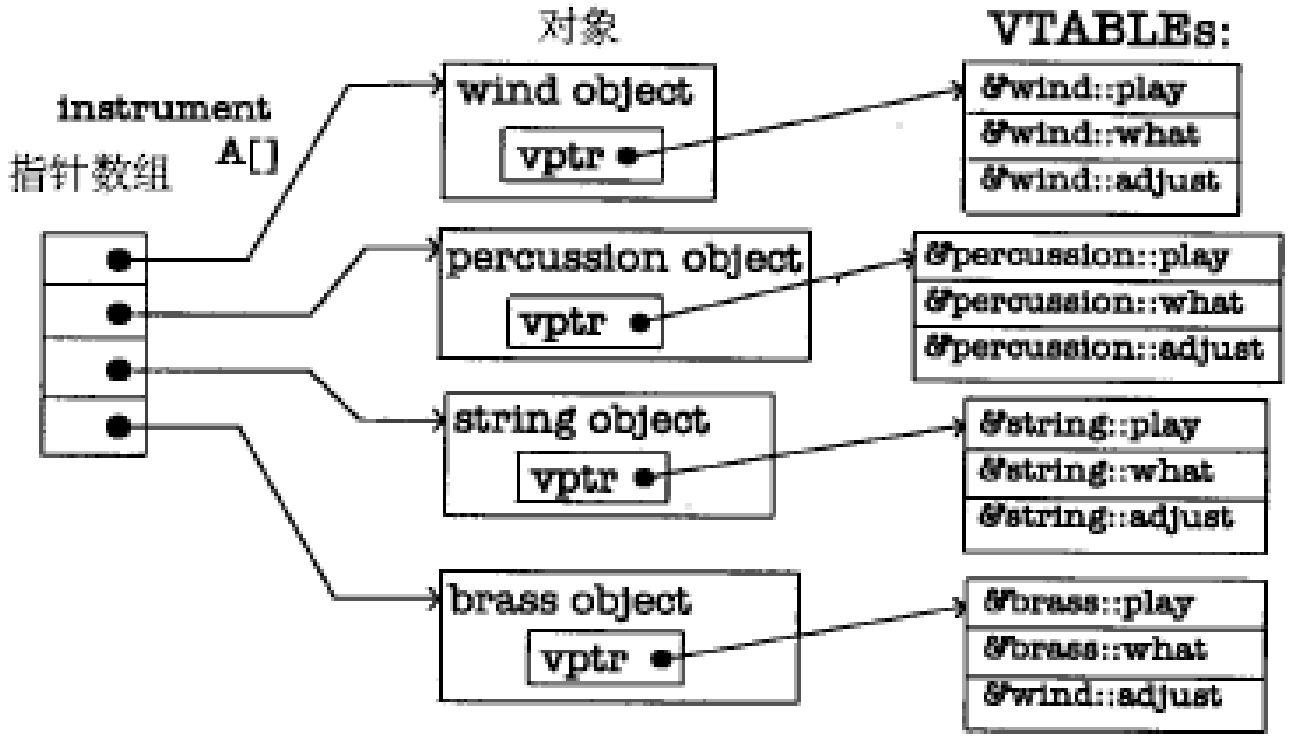
17 Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

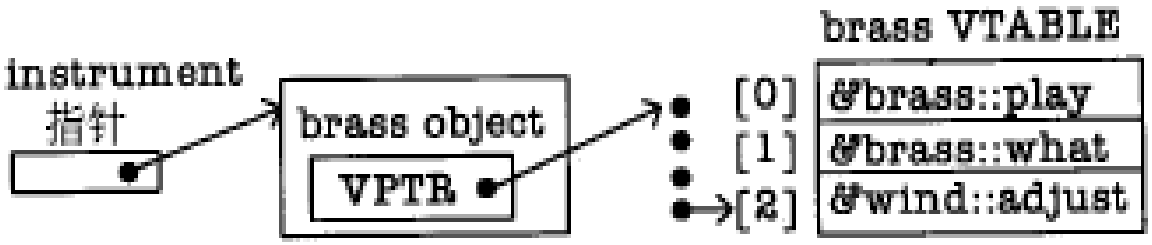
- 虚拟函数典型的调用过程
  - 编译器判断调用是否通过一个基类指针并且该函数为一个虚拟函数
  - 利用偏移量来定位 *vtable* 中的元素

17 Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

- 编译器生成执行下列操作的代码：
  - ✓选取函数调用中的指针
  - ✓解引用该指针得到相关对象
  - ✓解引用对象的 *vtable* 指针得到 *vtable*
  - ✓略过偏移量选择正确的函数指针
  - ✓解引用函数指针执行函数调用




对象
VTABLEs:
instrument
A[]
指针数组
wind object
vptr
&wind::play
&wind::what
&wind::adjust
percussion object
vptr
&percussion::play
&percussion::what
&percussion::adjust
string object
vptr
&string::play
&string::what
&string::adjust
brass object
vptr
&brass::play
&brass::what
&wind::adjust


brass VTABLE
instrument
指针
brass object
VPTR
[0]
[1]
[2]
&brass::play
&brass::what
&wind::adjust

18 Case Study: Payroll System Using Polymorphism and Run-Time Type Information with Downcasting, dynamic_cast, typeid and type_info

- 例子：奖励 BasePlusCommissionEmployees 将其 base salaries 增加 10%
- 使用运行时类型信息 (RTTI) 和动态类型转换
  - 一些编译器需要在使用 RTTI 前进行设置

18 Case Study: Payroll System Using Polymorphism and Run-Time Type Information with Downcasting, dynamic_cast, typeid and type_info

- dynamic_cast 运算符
  - 向下类型转换（Downcast ）运算
    - ✓ 基类指针转换为派生类指针
  - 如果指向的为派生类类型，转换被执行
    - ✓ 否则，赋值为 0
  - 如果不使用 dynamic_cast 并且尝试将派生类指针赋值给基类指针
    - ✓ 将会出现编译错误

18 Case Study: Payroll System Using Polymorphism and Run-Time Type Information with Downcasting, dynamic_cast, typeid and type_info

- typeid 运算符
  - 返回 type_info 类的对象的引用
    - ✓包含操作数的类型信息
    - ✓type_info 中的 name 成员函数
      - ◈返回包含类型名称的基于指针的字符串
  - 必须包含头文件 <typeinfo>

```
1  // Fig. 13.25: fig13_25.cpp
2  // Demonstrating downcasting and run-time type information.
3  // NOTE: For this example to run in Visual C++ .NET,
4  // you need to enable RTTI (Run-Time Type Info) for the project.
5  #include <iostream>
6  using std::cout;
7  using std::endl;
8  using std::fixed;
9
10 #include <iomanip>
11 using std::setprecision;
12
13 #include <vector>
14 using std::vector;
15
16 #include <typeinfo>
17
18 // include definitions of classes in Employee hierarchy
19 #include "Employee.h"
20 #include "SalariedEmployee.h"
21 #include "HourlyEmployee.h"
22 #include "CommissionEmployee.h"
23 #include "BasePlusCommissionEmployee.h"
24
25 int main()
26 {
27    // set floating-point output formatting
28    cout << fixed << setprecision( 2 );
```

```
29
30    // create vector of four base-class pointers
31    vector < Employee * > employees( 4 );
32
33    // initialize vector with various kinds of Employees
34    employees[ 0 ] = new SalariedEmployee(
35       "John", "Smith", "111-11-1111", 800 );
36    employees[ 1 ] = new HourlyEmployee(
37       "Karen", "Price", "222-22-2222", 16.75, 40 );
38    employees[ 2 ] = new CommissionEmployee(
39       "Sue", "Jones", "333-33-3333", 10000, .06 );
40    employees[ 3 ] = new BasePlusCommissionEmployee(
41       "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
42
43    // polymorphically process each element in vector employees
44    for ( size_t i = 0; i < employees.size(); i++ )
45    {
46       employees[ i ]->print(); // output employee information
47       cout << endl;
48
49       // downcast pointer
50       BasePlusCommissionEmployee *derivedPtr =
51          dynamic_cast < BasePlusCommissionEmployee * >
52             ( employees[ i ] );
```

Create employee objects, only one of type **BasePlusCommissionEmployee**

Downcast the **Employee** pointer to a **BasePlusCommissionEmployee** pointer

```
53
54       // determine whether element points to base-salaried
55       // commission employee
56       if ( derivedPtr != 0 ) // 0 if not a BasePlusCommissionEmployee
57       {
58          double oldBaseSalary = derivedPtr->getBaseSalary();
59          cout << "old base salary: $" << oldBaseSalary << endl;
60          derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
61          cout << "new base salary with 10% increase is: $"
62             << derivedPtr->getBaseSalary() << endl;
63       } // end if
64
65       cout << "earned $" << employees[ i ]->earnings() << "\n\n";
66    } // end for
67
68    // release objects pointed to by vector's elements
69    for ( size_t j = 0; j < employees.size(); j++ )
70    {
71       // output class name
72       cout << "deleting object of "
73          << typeid( *employees[ j ] ).name() << endl;
74
75       delete employees[ j ];
76    } // end for
77
78    return 0;
79 } // end main
```

Determine if cast was successful

If cast was successful, modify base salary

Use **typeid** and function **name** to display object types

```
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

deleting object of class SalariedEmployee
deleting object of class HourlyEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
```

19 Virtual Destructors

- 在C++中，不能声明虚构造函数，但可以声明虚析构函数。
- 虚函数作为运行过程中多态的基础，主要是针对对象的，而构造函数是在对象产生之前运行的，因此，虚构造函数是没有意义的。

19 Virtual Destructors

- 非虚拟析构函数
  - 不以关键字 virtual 声明的析构函数
  - 如果基类指针指向派生类对象，当使用 delete 运算符销毁该指针指向的对象时，其行为未定义

19 Virtual Destructors

- 虚拟析构函数
  - 用关键字 virtual 声明
  - 如果基类指针指向派生类对象，当使用 delete 运算符销毁该指针指向的对象时，相应的派生类的析构函数被调用
    - ✓然后基类的析构函数被执行

●如果外部程序使用new运算符定义了动态对象，则当外部程序结束时，要使用delete运算符删除该动态对象。

●如果外部程序利用赋值兼容原则，把动态申请的派生类对象地址赋给了基类对象指针，由于delete运算符隐含有对析构函数的自动调用，因此此时系统自动调用的必定是基类的析构函数，这就有可能引起内存泄漏问题。

- 解决这个问题的方法是：在存在虚函数的类层次中，在析构函数定义前也添加关键字virtual，即把析构函数也设计成虚析构函数。
- 如果外部程序利用赋值兼容原则，把动态申请的派生类对象地址赋给了基类对象指针，则delete运算符就能根据当前对象指针指向的是基类对象还是派生类对象，来自动调用基类的析构函数或是派生类的析构函数，从而避免出现内存泄漏问题。

虚析构函数的声明语法为：

virtual ～类名( );

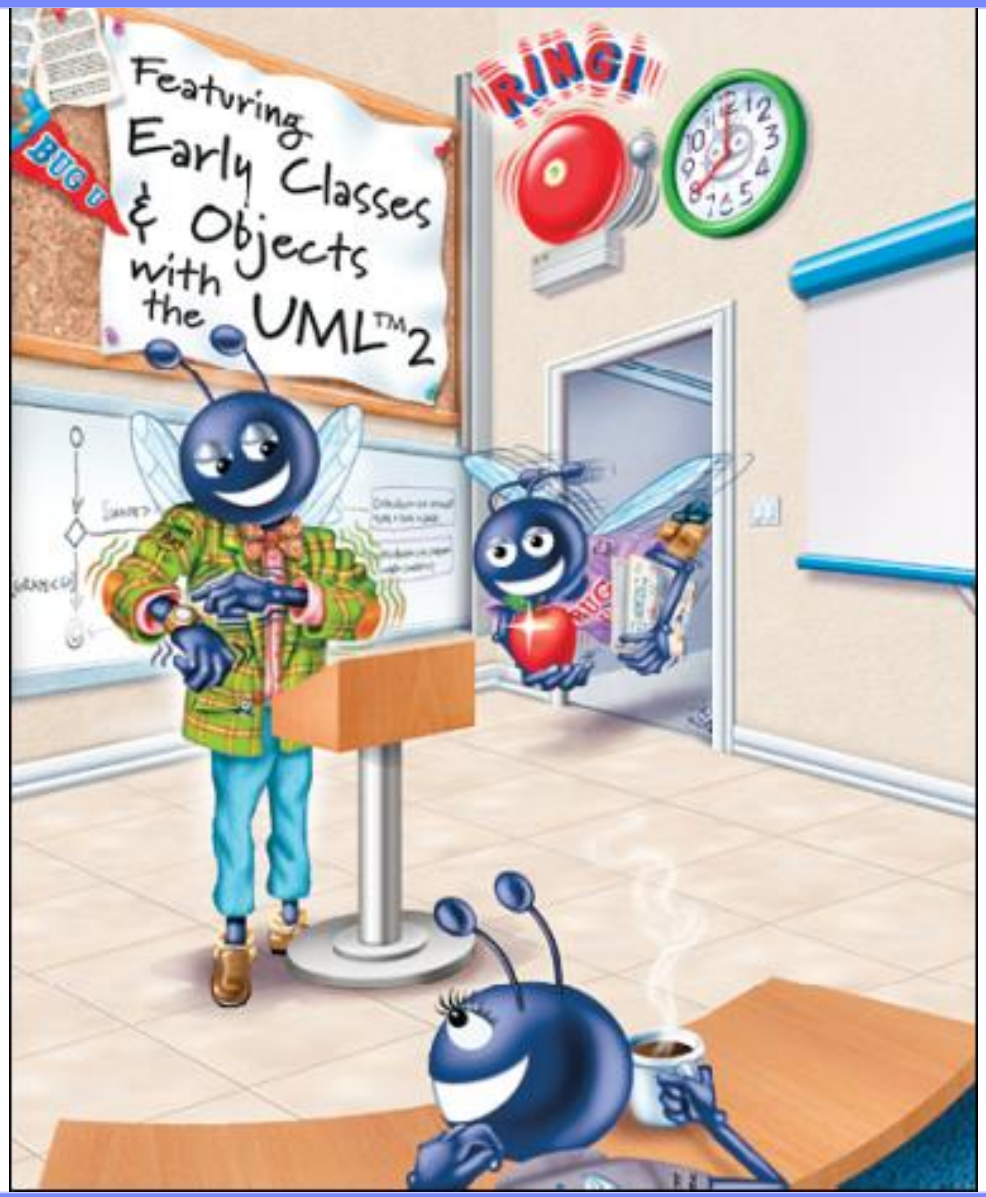
如果一个类的析构函数是虚函数，那么由它派生而来的所有子类的析构函数也是虚函数。析构函数设置为虚函数之后，在使用指针或引用时可以进行动态联编，实现运行时的多态，保证使用基类类型的指针或引用能够调用适当的析构函数对不同的对象进行清理工作。

虚析构函数的工作过程与普通虚函数不同，普通虚函数只是调用相应层上的函数，而虚析构函数是先调用相应层上的析构函数，然后逐层向上调用基类的析构函数.

19 Virtual Destructors

良好编程习惯：如果一个类有虚拟函数，该类就会提供一个虚析构函数，尽管该析构函数并不一定是该类需要的。从这个类派生出的类可以包含析构函数，但是必须正确调用。

常见编程错误：构造函数不能是虚拟函数，声明一个构造函数为虚拟函数是一个语法错误。



Thank you!