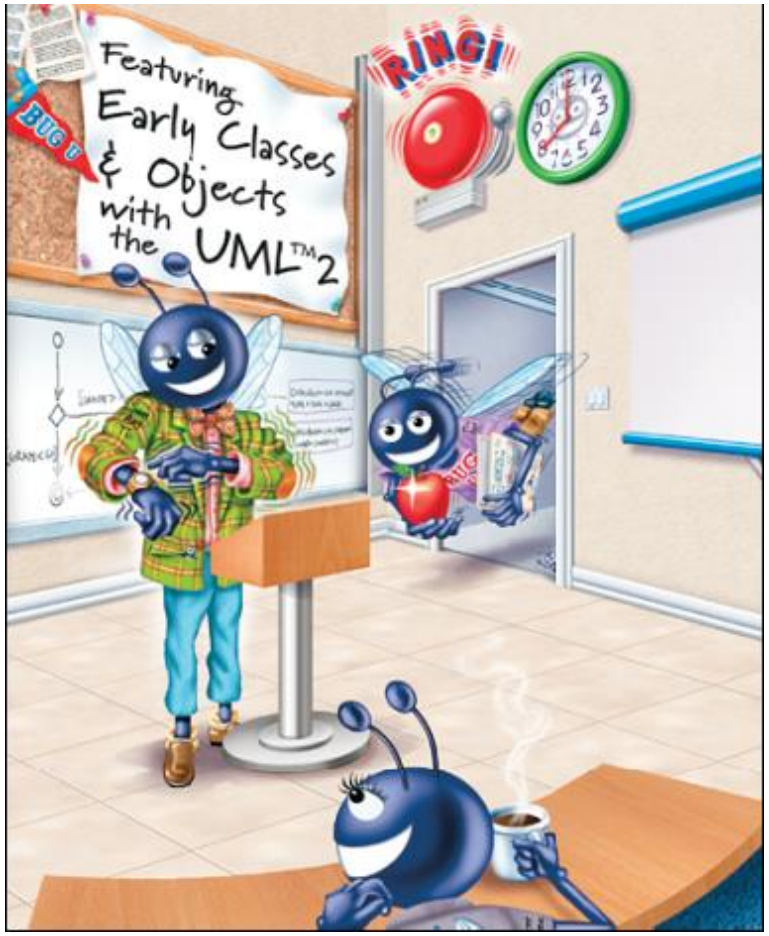
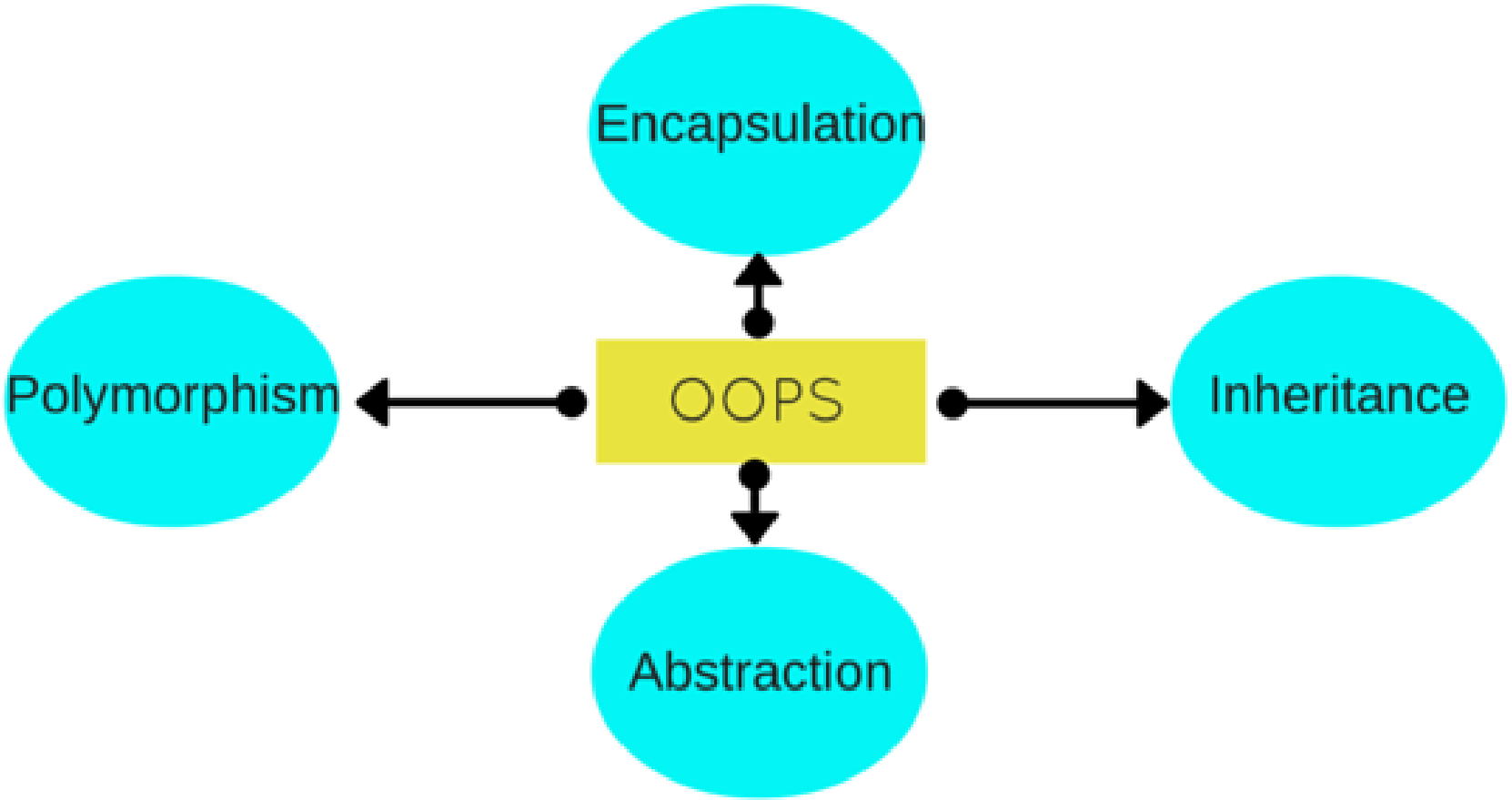
C++程序设计



上节课内容回顾

1. 通过继承现有类来创建新类
2. 基类和派生类之间的关系
3. protected 成员的访问
4. 继承层次中构造函数和析构函数的调用



第十二讲 面向对象编程：多态

学习目标：

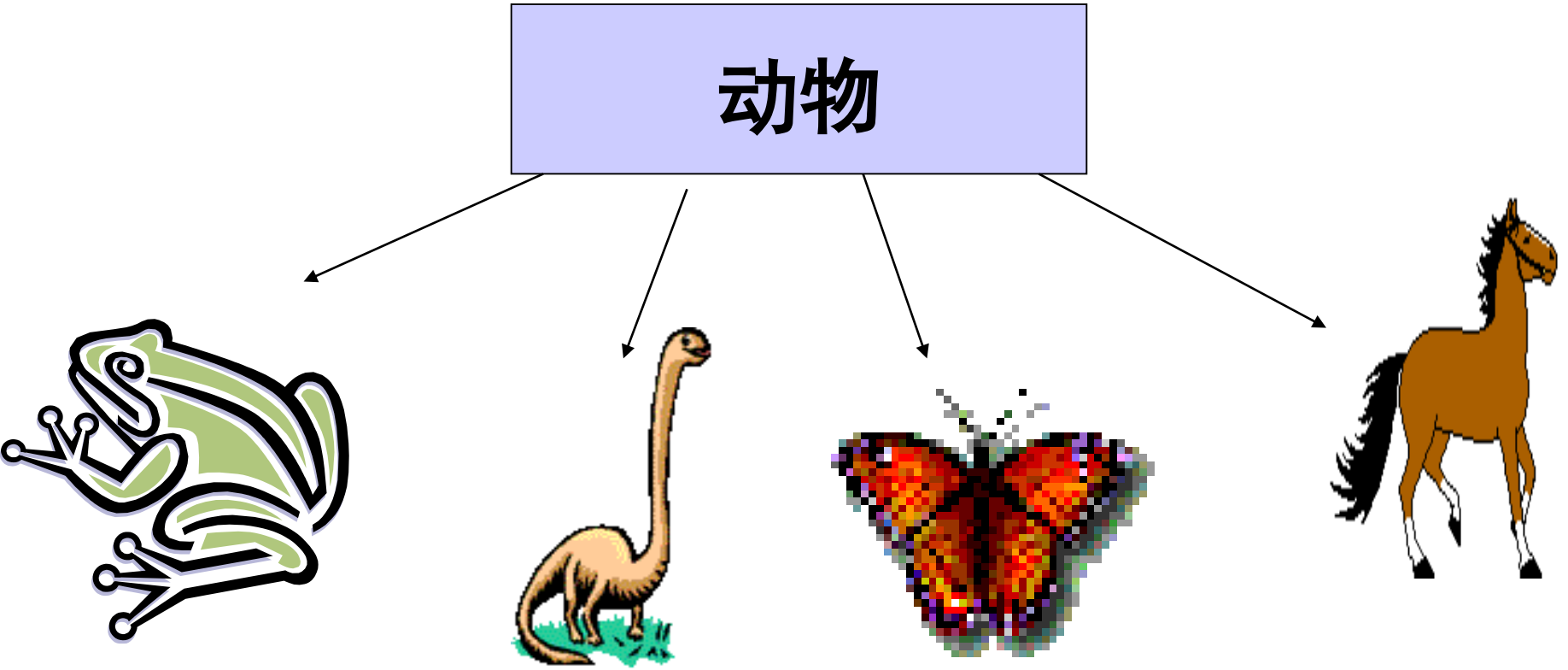
- 理解多态性的概念
- 理解如何声明和利用虚拟函数来实现多态性
- 理解多态性如何扩展和维护系统
- 理解C++如何实现虚拟函数和动态绑定
- 理解运行时类型信息（RTTI）和运算符typeid和dynamic_cast的用法

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



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

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**

4. Invoking Base-Class Functions from Derived-Class Objects

- 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

4. Invoking Base-Class Functions from Derived-Class Objects

- 基类指针指向派生类对象
  - 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

```
double CommissionEmployee::earnings() const
{
   return getCommissionRate() * getGrossSales();
} // end function earnings

void CommissionEmployee::print() const
{
   cout << "commission employee: "
      << getFirstName() << ' ' << getLastName()
      << "\nsocial security number: " << getSocialSecurityNumber()
      << "\ngross sales: " << getGrossSales()
      << "\ncommission rate: " << getCommissionRate();
} // end function print
```

```
double BasePlusCommissionEmployee::earnings() const
{
   return getBaseSalary() + CommissionEmployee::earnings();
} // end function earnings

void BasePlusCommissionEmployee::print() const
{
   cout << "base-salaried ";

   // invoke CommissionEmployee's print function
   CommissionEmployee::print();

   cout << "\nbase salary: " << getBaseSalary();
} // end function print
```

```
int main( )
{
   // create base-class pointer
   CommissionEmployee *commissionEmployeePtr = 0;
   ......
   // aim base-class pointer at base-class object and print
   commissionEmployeePtr = &commissionEmployee; // perfectly natural
   cout << "\n\n\nCalling print with base-class pointer to "
      << "\nbase-class object invokes base-class print function:\n\n";
   commissionEmployeePtr->print(); // invokes base-class print
```

```
   basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\n\nCalling print with derived-class pointer to "
      << "\nderived-class object invokes derived-class "
      << "print function:\n\n";
   basePlusCommissionEmployeePtr->print();
   // aim base-class pointer at derived-class object and print
   commissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\n\nCalling print with base-class pointer to "
      << "derived-class object\ninvokes base-class print "
      << "function on that derived-class object:\n\n";
   commissionEmployeePtr->print(); // invokes base-class print
   cout << endl;
   return 0;
} // end main
```

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

```
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "222-22-2222", 10000, .06 );
   BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

   // aim derived-class pointer at base-class object
   basePlusCommissionEmployeePtr = &commissionEmployee;
   return 0;
} // end main
```

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
    - 可以通过 downcasting 来完成

```
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;

   string firstName = commissionEmployeePtr->getFirstName();
   string lastName = commissionEmployeePtr->getLastName();
   ......
   double baseSalary = commissionEmployeePtr->getBaseSalary();
   commissionEmployeePtr->setBaseSalary( 500 );
   return 0;
} // end main
```

7. Virtual Functions

- 哪个函数被调用？
  - 通常句柄决定能够调用的函数
  - 对于虚拟函数（virtual functions）
    - 句柄所指向的对象类型，而不是句柄的类型决定了虚拟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
  - 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

```
class CommissionEmployee
{
public:
   CommissionEmployee( const string &, const string &, const string &,
      double = 0.0, double = 0.0 );
   ......
    virtual double earnings() const; // calculate earnings
    virtual void print() const; // print CommissionEmployee object
private:
   string firstName;
   ......
};
```

```
class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
   BasePlusCommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0, double = 0.0 );

   void setBaseSalary( double ); // set base salary
   double getBaseSalary() const; // return base salary

   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print BasePlusCommissionEmployee object
private:
   double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee
```

```
int main( )
{
   ......
  // aim base-class pointer at base-class object and print
  commissionEmployeePtr = &commissionEmployee;
  cout << "\n\nCalling virtual function print with base-class pointer"
     << "\nto base-class object invokes base-class "
     << "print function:\n\n";
  commissionEmployeePtr->print(); // invokes base-class print
```

```
   basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\nCalling virtual function print with derived-class "
      << "pointer\nto derived-class object invokes derived-class "
      << "print function:\n\n";
   basePlusCommissionEmployeePtr->print();

   commissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\nCalling virtual function print with base-class pointer"
      << "\nto derived-class object invokes derived-class "
      << "print function:\n\n";

   // polymorphism; invokes BasePlusCommissionEmployee's print;
   // base-class pointer to derived-class object
   commissionEmployeePtr->print();
   cout << endl;
   return 0;
} // end main
```

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

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
  - 基类指针指向派生类对象
    - ◈安全，但只能用来调用基类的成员函数
    - ◈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来决定对象类型
  - 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
    - 类型测试可能被忘记
    - 增加新的类型可能忘记增加 switch 语句

10. Abstract Classes and Pure virtual Functions

- 抽象类
  - 不能被实例化的类
    - 不完整—派生类来定义“缺失的部分”
  - 通常作为基类，称为抽象类
    - 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” in its declaration
      - 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

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拟函数。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

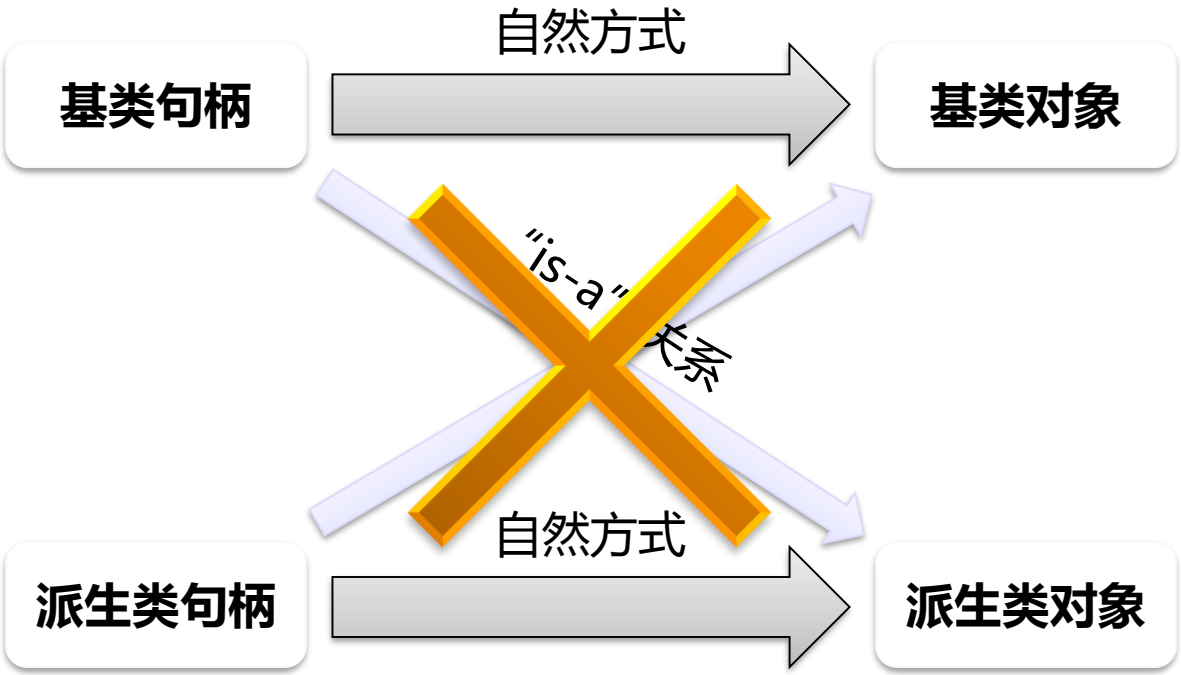
11. Case Study: Payroll System Using Polymorphism

- 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
  - Earnings 和 print 处理不同的派生类对象

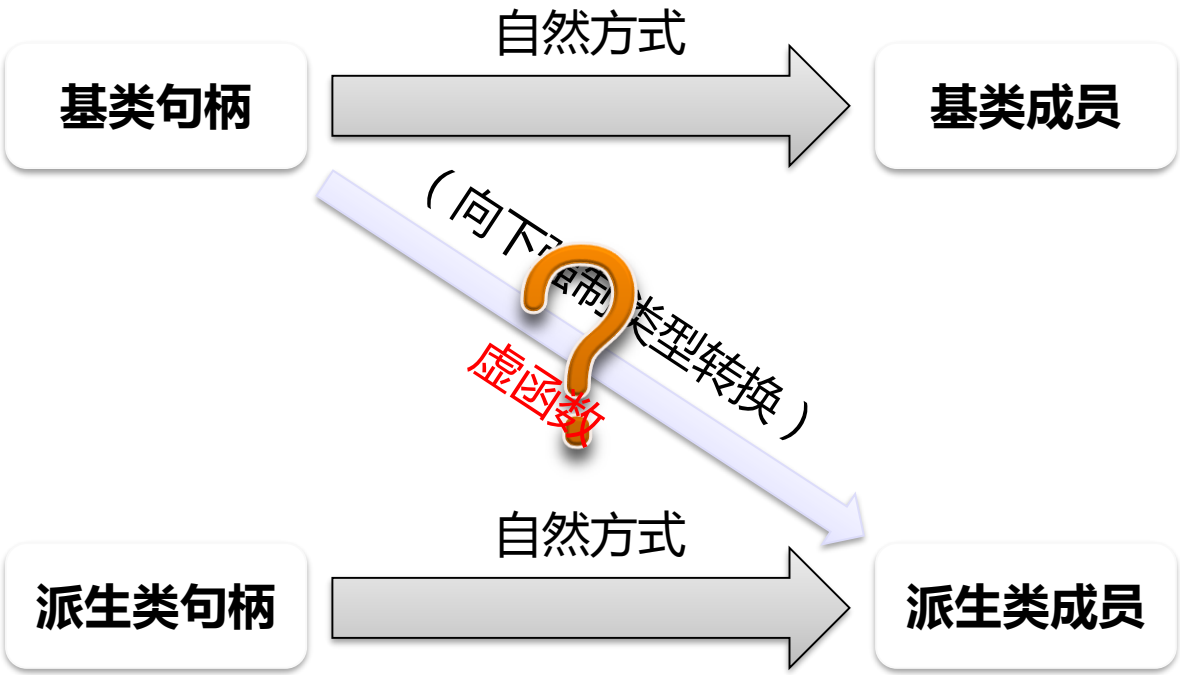
11. Case Study: Payroll System Using Polymorphism

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。

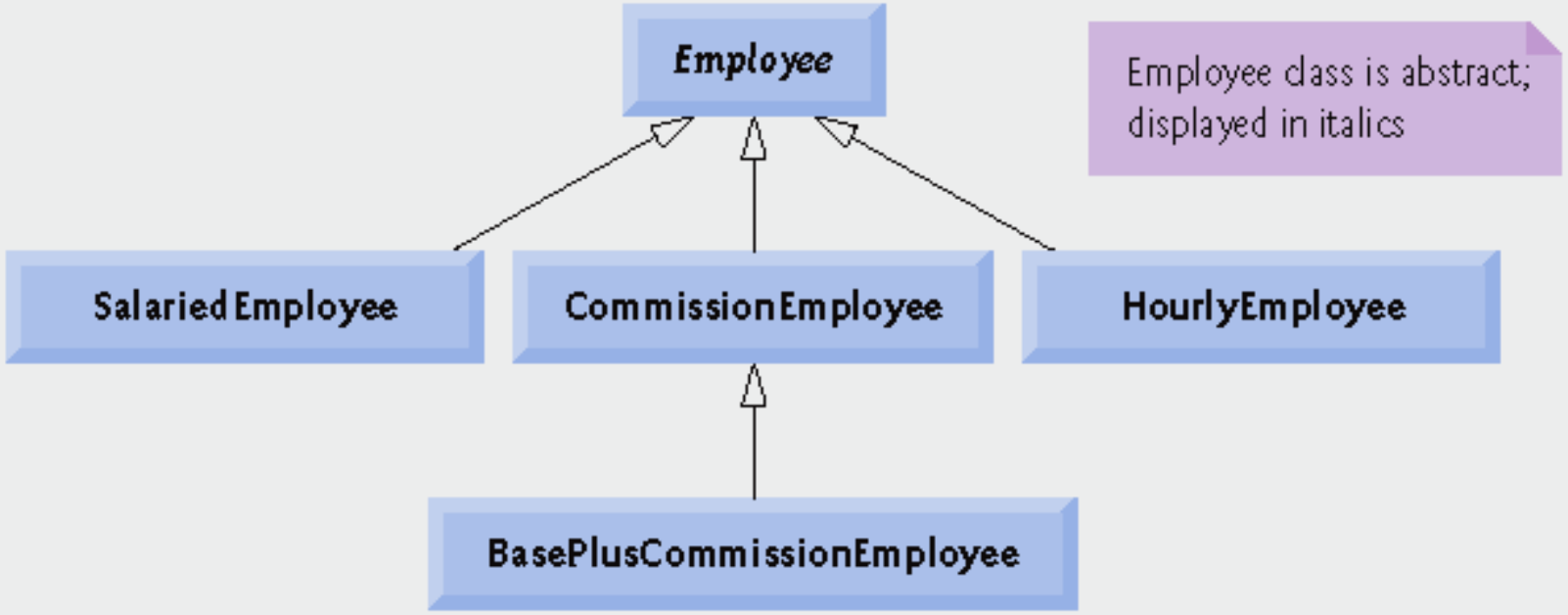
基类与派生类句柄间的赋值



通过句柄进行的基类与派生类间的相互访问



11. Case Study: Payroll System Using Polymorphism



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

```
class Employee
{
public:
   Employee( const string &, const string &, const string & );
   ......
   // pure virtual function makes Employee abstract base class
   virtual double earnings() const = 0; // pure virtual
   virtual void print() const; // virtual
private:
   string firstName;
   string lastName;
   string socialSecurityNumber;
}; // end class Employee
```

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）

```
class SalariedEmployee : public Employee
{
public:
   SalariedEmployee( const string &, const string &,
      const string &, double = 0.0 );

   void setWeeklySalary( double ); // set weekly salary
   double getWeeklySalary() const; // return weekly salary

   // keyword virtual signals intent to override
   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print SalariedEmployee object
private:
   double weeklySalary; // salary per week
}; // end class SalariedEmployee
```

```
// 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
```

```
// calculate earnings;
// override pure virtual function earnings in Employee
double SalariedEmployee::earnings() const
 {
   return getWeeklySalary();
} // end function earnings

// print SalariedEmployee's information
void SalariedEmployee::print() const
 {
   cout << "salaried employee: ";
   Employee::print(); // reuse abstract base-class print function
   cout << "\nweekly salary: " << getWeeklySalary();
} // end function print
```

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）

```
class HourlyEmployee : public Employee
{
public:
   HourlyEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0 );

   void setWage( double ); // set hourly wage
   double getWage() const; // return hourly wage

   void setHours( double ); // set hours worked
   double getHours() const; // return hours worked

   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print HourlyEmployee object
private:
   double wage; // wage per hour
   double hours; // hours worked for week
}; // end class HourlyEmployee
```

```
// override pure virtual function earnings in Employee
double HourlyEmployee::earnings() const
{
   if ( getHours() <= 40 ) // no overtime
      return getWage() * getHours();
   else
      return 40 * getWage() + ( ( getHours() - 40 ) * getWage() * 1.5 );
} // end function earnings

// print HourlyEmployee's information
void HourlyEmployee::print() const
{
   cout << "hourly employee: ";
   Employee::print(); // code reuse
   cout << "\nhourly wage: " << getWage() <<
      "; hours worked: " << getHours();
} // end function print
```

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）

```
class CommissionEmployee : public Employee
{
public:
   CommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0 );

   void setCommissionRate( double ); // set commission rate
   double getCommissionRate() const; // return commission rate

   void setGrossSales( double ); // set gross sales amount
   double getGrossSales() const; // return gross sales amount

   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print CommissionEmployee object
private:
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
}; // end class CommissionEmployee
```

```
double CommissionEmployee::earnings() const
{
   return getCommissionRate() * getGrossSales();
} // end function earnings

// print CommissionEmployee's information
void CommissionEmployee::print() const
{
   cout << "commission employee: ";
   Employee::print(); // code reuse
   cout << "\ngross sales: " << getGrossSales()
      << "; commission rate: " << getCommissionRate();
} // end function print
```

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

```
class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
   BasePlusCommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0, double = 0.0 );

   void setBaseSalary( double ); // set base salary
   double getBaseSalary() const; // return base salary

   // keyword virtual signals intent to override
   virtual double earnings() const; // calculate earnings
   virtual void print() const;
private:
   double baseSalary; // base salary per week
}; // end class BasePlusCommissionEmployee
```

```
// calculate earnings;
// override pure virtual function earnings in Employee
double BasePlusCommissionEmployee::earnings() const
{
   return getBaseSalary() + CommissionEmployee::earnings();
} // end function earnings

// print BasePlusCommissionEmployee's information
void BasePlusCommissionEmployee::print() const
{
  cout << "base-salaried ";
  CommissionEmployee::print(); // code reuse
  cout << "; base salary: " << getBaseSalary();
} // end function print
```

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

17. Demonstrating Polymorphic Processing

- 创建 SalariedEmployee，HourlyEmployee，CommissionEmployeeBase 和 PlusCommissionEmployee 对象
  - 演示多态的来处理对象
    - 使用 Employee 指针向量
    - 使用指针或应用来调用虚拟函数

```
int main()
{
  // set floating-point output formatting
  cout << fixed << setprecision( 2 );

  // create derived-class objects
  SalariedEmployee salariedEmployee(
    "John", "Smith", "111-11-1111", 800 );
  HourlyEmployee hourlyEmployee(
    "Karen", "Price", "222-22-2222", 16.75, 40 );
  CommissionEmployee commissionEmployee(
    "Sue", "Jones", "333-33-3333", 10000, .06 );
  BasePlusCommissionEmployee basePlusCommissionEmployee(
    "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
```

```
cout << "Employees processed individually using static binding:\n\n";


salariedEmployee.print();
cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
hourlyEmployee.print();
cout << "\nearned $" << hourlyEmployee.earnings() << "\n\n";
commissionEmployee.print();
cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
basePlusCommissionEmployee.print();
cout << "\nearned $" << basePlusCommissionEmployee.earnings()
   << "\n\n";
```

```
   // create vector of four base-class pointers
   vector < Employee * > employees( 4 );

   // initialize vector with Employees
   employees[ 0 ] = &salariedEmployee;
   employees[ 1 ] = &hourlyEmployee;
   employees[ 2 ] = &commissionEmployee;
   employees[ 3 ] = &basePlusCommissionEmployee;

   cout << "Employees processed polymorphically via dynamic
binding:\n\n";
```

```
   cout << "Virtual function calls made off base-class pointers:\n\n";

   for ( size_t i = 0; i < employees.size(); i++ )
      virtualViaPointer( employees[ i ] );

   // call virtualViaReference to print each Employee's information
   // and earnings using dynamic binding
   cout << "Virtual function calls made off base-class references:\n\n";

   for ( size_t i = 0; i < employees.size(); i++ )
      virtualViaReference( *employees[ i ] ); // note dereferencing

   return 0;
} // end main
```

```
// call Employee virtual functions print and earnings off a
// base-class pointer using dynamic binding
void virtualViaPointer( const Employee * const baseClassPtr )
{
   baseClassPtr->print();
   cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
} // end function virtualViaPointer

// call Employee virtual functions print and earnings off a
// base-class reference using dynamic binding
void virtualViaReference( const Employee &baseClassRef )
{
   baseClassRef.print();                                    '
   cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
} // end function virtualViaReference
```

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
  - 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”

- C++ 如何实现多态、虚拟函数和动态绑定
  - 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”

- C++ 如何实现多态、虚拟函数和动态绑定
  - 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

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
  - 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

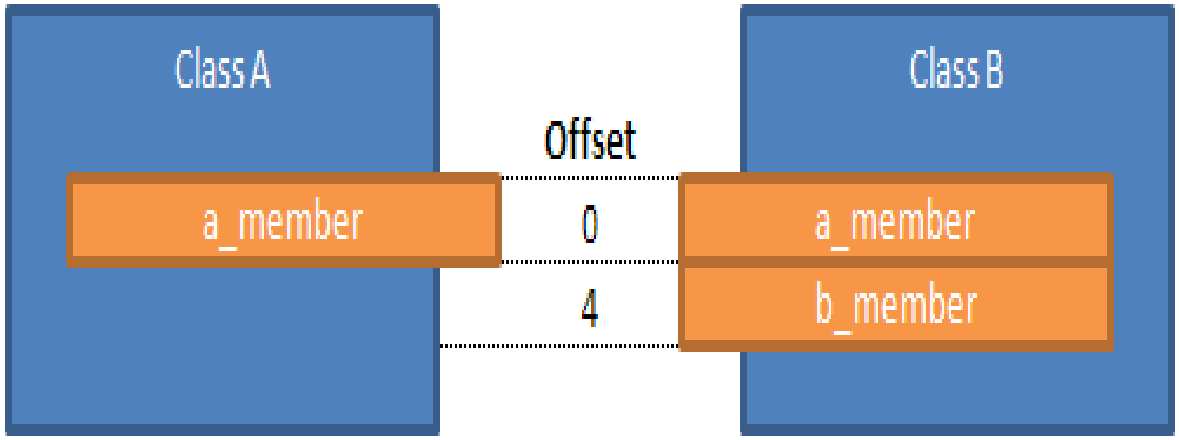
17. Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood”

- 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”

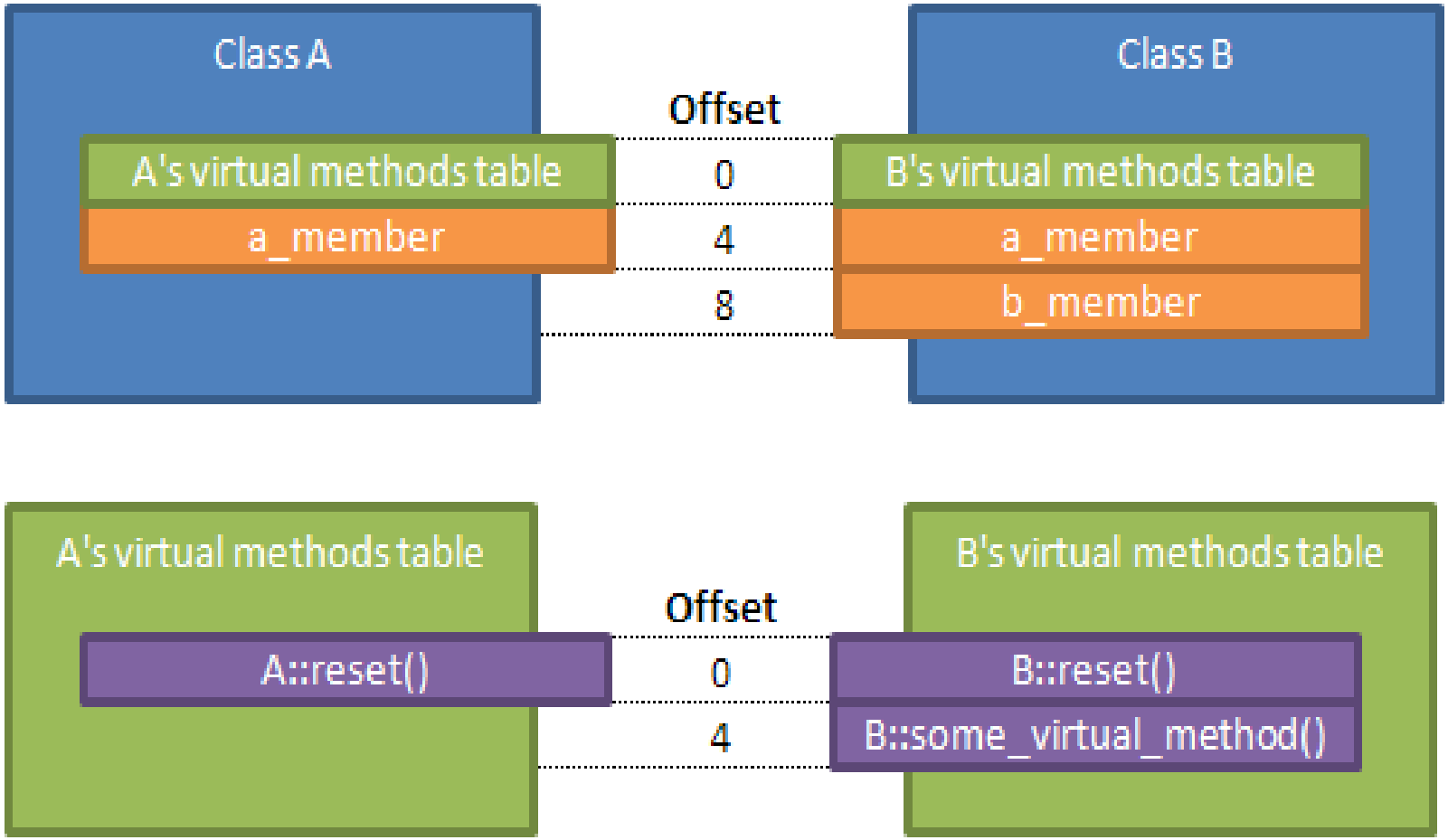
- 虚拟函数典型的调用过程
  - 编译器生成执行下列操作的代码：
    - 选取函数调用中的指针
    - 解引用该指针得到相关对象
    - 解引用对象的 *vtable* 指针得到 *vtable*
    - 略过偏移量选择正确的函数指针
    - 解引用函数指针执行函数调用

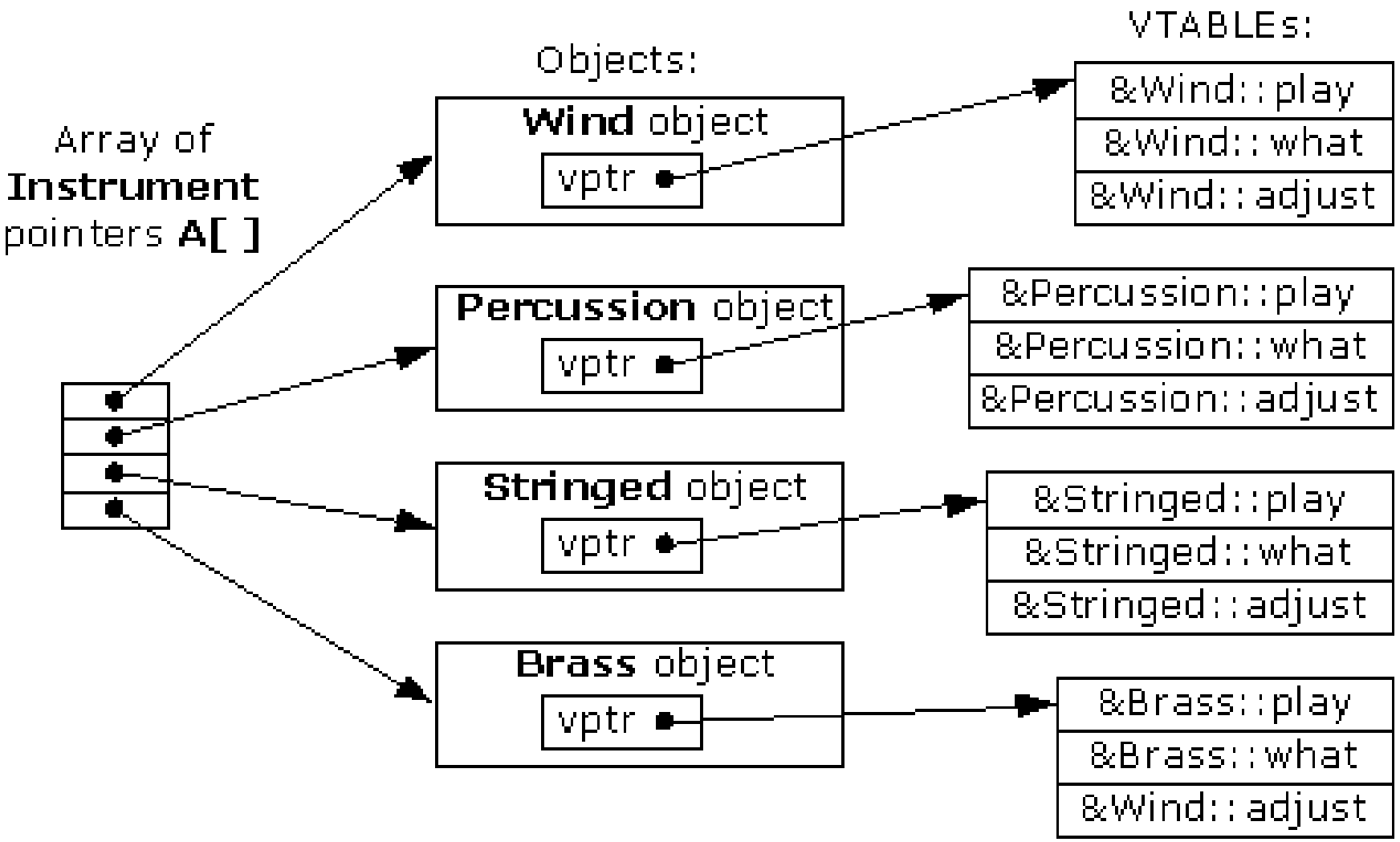
```
class A
{
 public:
      A() { a_member = 0; }
protected:
      int a_member;
};

class B : public A
{
public:
      B() : A() { b_member = 0; };
protected:
     int b_member;
};
```



```
class A
{
    public:
        A() { a_member = 0; }
        virtual void reset() { a_member = 0; }
        void set_a_member( int a ) { a_member = a; }
        int get_a_member() { return a_member; }
    protected:
        int a_member;
};
```

```
class B : public A
{
    public:
        B() : A() { b_member = 0; };
        virtual void reset() { a_member = b_member = 0; }
        virtual void some_virtual_method() { }
        void set_b_member(int b ) { b_member = b; }
        int get_b_member() { return b_member; }
    protected:
        int b_member;
};
```





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

- 书P$_{557}$页，图13.24

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

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
  - 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>

```
int main( )
{
   // create vector of four base-class pointers
   vector < Employee * > employees( 4 );

   // initialize vector with various kinds of Employees
   employees[ 0 ] = new SalariedEmployee(
      "John", "Smith", "111-11-1111", 800 );
   employees[ 1 ] = new HourlyEmployee(
      "Karen", "Price", "222-22-2222", 16.75, 40 );
   employees[ 2 ] = new CommissionEmployee(
      "Sue", "Jones", "333-33-3333", 10000, .06 );
   employees[ 3 ] = new BasePlusCommissionEmployee(
      "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
```

```
// polymorphically process each element in vector employees
for ( size_t i = 0; i < employees.size(); i++ )
{
   employees[ i ]->print(); // output employee information
   cout << endl;

   // downcast pointer
   BasePlusCommissionEmployee *derivedPtr =
      dynamic_cast < BasePlusCommissionEmployee * >
         ( employees[ i ] );
```

```
    // determine whether element points to base-salaried
    // commission employee
    if ( derivedPtr != 0 ) // 0 if not a BasePlusCommissionEmployee
    {
       double oldBaseSalary = derivedPtr->getBaseSalary();
       cout << "old base salary: $" << oldBaseSalary << endl;
       derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
       cout << "new base salary with 10% increase is: $"
          << derivedPtr->getBaseSalary() << endl;
    } // end if

    cout << "earned $" << employees[ i ]->earnings() << "\n\n";
  } // end for
```

```
   // release objects pointed to by vector's elements
   for ( size_t j = 0; j < employees.size(); j++ )
   {
      // output class name
      cout << "deleting object of "
         << typeid( *employees[ j ] ).name() << endl;

      delete employees[ j ];
   } // end for

   return 0;
} // end main
```

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

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。