Module03-10
C++ 面向对象编程：继承

面向对象编程 - 继承

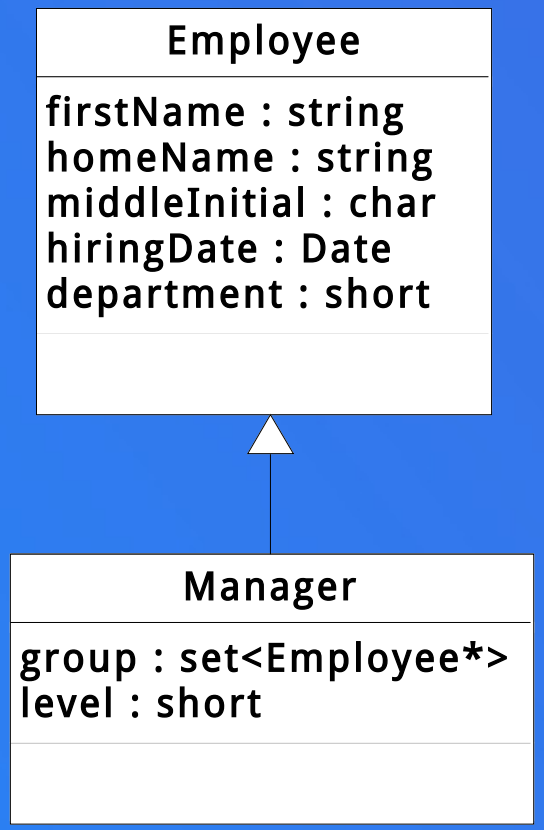
- 继承 (Inheritance)：
  - 派生类 (Derived Classes)
  - 抽象类 (Abstract Classes)

继承 - 派生类

- 从 Employee 派生 Manager
  - Employee 为 Manager 的基类
  - 派生类 Manager 从基类 Employee 继承所有的成员，且包含自己的成员： group 、 level

```
class Employee {
    string firstName;
    string homeName;
    char middleInitial;
    Date hiringDate;
    short department;
};

class Manager : public Employee {
    set<Employee*> group;
    short level;
};
```



继承 - 派生类

- 从 Empoyee 派生 Manager（续）
  - 一个 Manager（子类）也是一个 Employee（基类），但反过来说就不一定正确
  - 一个 Manager 的指针 Manager* 可以直接赋值给其基类 Manager 的指针 Employee*，不需作显式转换；反过来将一个 Employee* 赋值给 Manager* 则需作显式类型转换

```
Employee e;
Manager m;

Employee* ep = &m;   // OK, 一个Manager也是一个Employee
Manager* mp = &e;    // Error, 不一定所有的Employee都是Manager
```

继承 - 派生类

- 成员函数
  - 派生类可以使用基类的 public 和 protected 成员，但是不能访问基类的 private 成员

```
class Employee {
    string firstName;
    string homeName;
    // ...
public:
    Employee(const string& name, const short& d);
    void print();
    // ...
};
class Manager : public Employee {
    set<Employee*> group;
    short level;
public:
    Manager(const string& name, const short& d, const short&
lev);
    void print();
    // ...
};
```

- 成员函数（续）
  - 示例：

```
void Manager::print() {
    Employee::print(); // OK
    cout << "First name: " << firstName << endl; // Error: firstName
是类Employee的private成员
    // ...
    cout << "Level: " << level << endl;
    print(); // 危险，无穷递归调用
}
```

继承 - 派生类

- 构造函数和析构函数
  - 如果派生类要定义构造函数：
    - 如果其基类有构造函数，则必须调用基类的某个构造函数
    - 基类的默认构造函数可以隐式的被调用
    - 如果基类的所有构造函数都有参数，则需显式的调用
    - 派生类的构造，只能显式初始化自身的数据成员，基类的数据成员不能出现在初始化列表中
  - 类对象的构造、析构：
    - 类对象的构造是自下而上的，即：首先是基类，而后是成员，接下来才是派生类的本身
    - 类对象的析构恰好以相反的次序进行：首先是派生类本身，而后是成员，之后是基类
    - 成员和基类的构造严格按照在类中声明的顺序，它们的销毁则按相反的次序进行

- 构造函数和析构函数（续）
  - 示例：

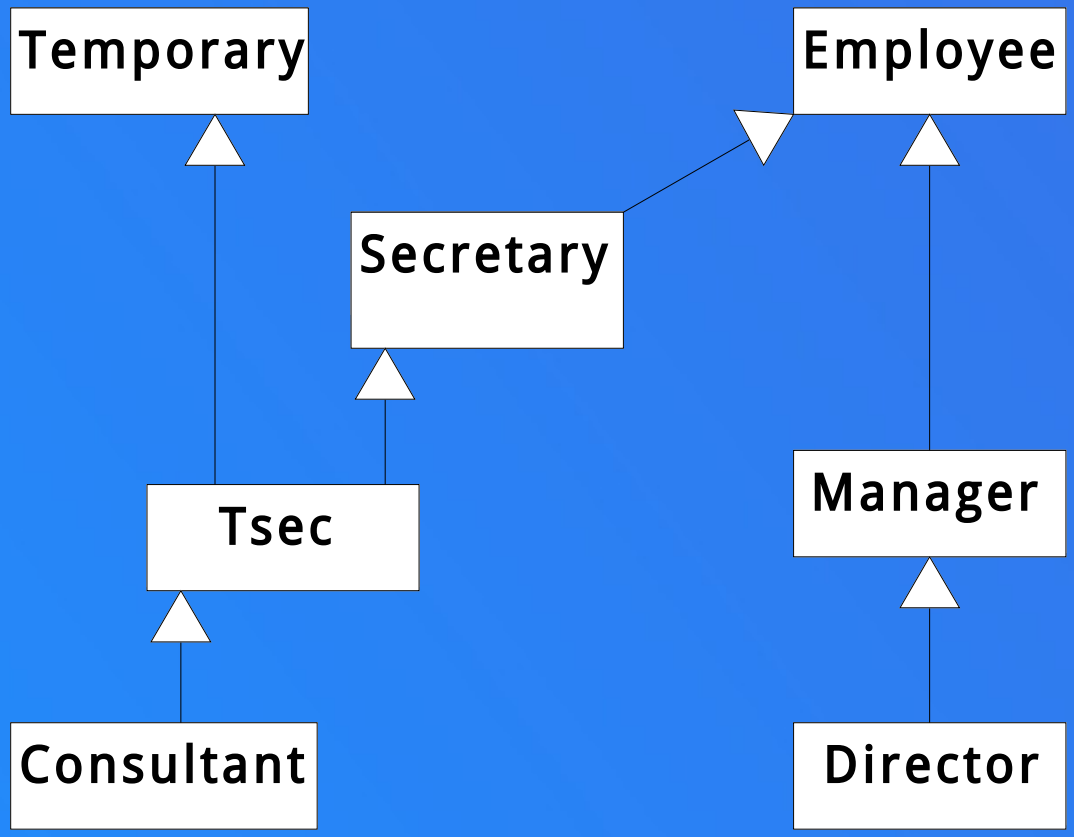
```
Employee::Employee(const string& name, const short& d) :
    firstName(name), department(d) {
    // ...
}

Manager::Manager(const string& name, const short& d, const short&
lev) :
    firstName(name), department(d), level(lev) { //
Error, firstName和department不是Manager的成员
    // ...
}

Manager::Manager(const string& name, const short& d, const short&
lev) :
    Employee(name, d), level(lev) { // OK
    // ...
}
```

继承 - 派生类

- 类层次
  - 派生类也可以作为基类，如下图： Employee 的派生类 Manager 就是 Director 的基类



- 虚函数
  - 派生类 Manager 成员函数 print() 的访问问题，如下面的情形：

```
void Employee::print() const {
    cout << "Employee::print()\n";
}

void Manager::print() const {
    cout << "Manager::print()\n";
}

void f1(Employee* e) {
    e->print();
}

int main() {
    Employee e("Joy", 200);
    Manager m("Stallman", 120, 2);

    f1(&e);  // Employee::print()
    f1(&m);  // Employee::print() → ???
}
```

- 虚函数（续）
  - 引入虚函数机制，将基类 Employee 的类成员函数 print() 声明为 virtual

```
class Employee {
    string firstName;
    string homeName;
    // ...
public:
    Employee(const string& name, const short& d);
    virtual void print();
    // ...
};
class Manager : public Employee {
    set<Employee*> group;
    short level;
public:
    Manager(const string& name, const short& d, const short&
lev);
    virtual void print();  // 子类的virtual关键字可写可不写
    // ...
};
```

- 虚函数（续）
  - 再看子类 Manager 成员函数 print() 的访问：

```
void Employee::print() const {
    cout << "Employee::print()\n";
}

void Manager::print() const {
    cout << "Manager::print()\n";
}

void f1(Employee* e) {
    e->print();
}

int main() {
    Employee e("Joy", 200);
    Manager m("Stallman", 120, 2);

    f1(&e);  // Employee::print()
    f1(&m);  // Manager::print() → OK
}
```

继承 - 派生类

- 虚函数（续）
  - 关于虚函数：
    - 基类的虚函数（除纯虚函数外）必须完整定义
    - 派生类要使用自己的版本的函数，可以覆盖 (Overriding) 基类的虚函数，但必须符合：
      - 派生类的函数名、参数列表、 const 限定等必须一致
      - 返回类型可以允许有些微差别（如基类的函数返回基类 B 的指针，子类的函数返回 B 的派生类 D 的指针）
    - 派生类可以不覆盖基类的虚函数，而直接使用之
    - 即使派生类覆盖了基类的虚函数，还是可以访问基类的虚函数，如在 Manager 类的函数中可以这么调用 Employee 的 print() 函数： Employee::print()

继承 - 多态类型

- 关于多态类型
  - 象前例中，从 Employee 的虚函数 print() 中取得正确的行为，而且不依赖于实际使用的到底是哪种具体的 Employee，这就是多态 (Polymorphism)
  - 一个带有虚函数的类型被称为“多态类型”
  - 要体现多态行为：
    - 被调用的必须是虚函数
    - 对象必须是通过指针或引用进行操作

继承 - 虚函数表

- 虚函数表 (Virtual Table)
  - 在多态行为中，为了保证调用正确的函数（如是基类、或某个派生类的函数），编译器必须在基类的各个子类的每个对象中存储某种信息，在选择正确函数时使用。
  - 在典型实现中，这些信息所需的空间只是一个指针
  - 只有包含了虚函数的类型才需要这个虚函数表
  - (DEMO - 虚函数表占用的空间)

继承 - 虚析构函数

- 虚析构函数
  - 如果一个类中有虚函数，则有可能成为其它类的基类
  - 基类并不知道其派生类的对象在销毁之前是否需要做一些清理工作，但将基类的析构函数声明成 virtual ，就能保证调用正确类型的（派生类）的析构函数
  - 所以：如果一个类中有虚函数，它就应该拥有一个虚析构函数
  - (DEMO)

继承 - 抽象类

- 纯虚函数 (Pure Virtual Function)
  - 一个类中，如果某个函数被声明成类似如下形式：

    ```
    virtual void f() = 0;
    ```

    后面带了 = 0，则该函数是纯虚函数
  - 一个类的纯虚函数一般不需要实现（当然也可以实现）

继承 - 抽象类

- 抽象类 (Abstract Classes)
  - 如果一个类中存在一个或多个纯虚函数，这个类就是抽象类
  - 抽象类不能创建对象
  - 抽象类一般作为接口，或作为其它类的基类
  - 如果一个抽象类的派生类没有覆盖抽象类的全部纯虚函数，则该派生类也是一个抽象类
- 抽象类的重要性
  - 抽象类是将接口与实现解耦的一个有效的机制

继承 - 抽象类

- 接口 (Interface)
  - 如果一个抽象类中：
    - 没有任何构造、复制构造、赋值操作符函数
    - 所有函数都是 public 的纯虚函数
    - 除 public 的 static const 成员外，无其它数据成员
    - 通常有一个虚析构函数

    我们通常将这样的抽象类称为“接口”

继承 - Bjarne's Advices

- Bjarne's Advices
  - 用指针和引用避免切割问题
  - 用抽象类将设计的中心集中到提供清晰的接口方面
  - 用抽象类使接口最小化
  - 用抽象类从接口中排除实现细节
  - 用虚函数使新的实现能够添加进来，同时也不影响用户代码
  - 用抽象类去尽可能减少用户代码的重新编译
  - 用抽象类使不同的实现能共存
  - 一个有虚函数的类应该有一个虚析构函数
  - 抽象类通常不需要构造函数