

Assignment Project Exam Help

C++ - Object Oriented Paradigm

<https://powcoder.com>

December 4, 2017

Add WeChat powcoder

Assignment Project Exam Help

1 Polymorphism and Dynamic Binding

2 Pure Virtual Functions, Abstract Classes

<https://powcoder.com>

3 Member Access Control

Add WeChat powcoder

4 Abstract Data Types (ADT) and Templates

- covered in previous lectures: classes and objects (instances of classes)
- a derived class is defined by adding/modifying features to/of an existing class without reprogramming (no removing of features possible)
- derived classes inherit characteristics of their base classes and code is reused
- this results in a common interface for several related, but not identical classes

objects of these classes may behave differently but may be manipulated identically by other parts of the program

Example: Table Lamp, Adjustable Table Lamp



A table lamp may be switched on or off by pressing a switch.

<https://powcoder.com>

Add WeChat powcoder

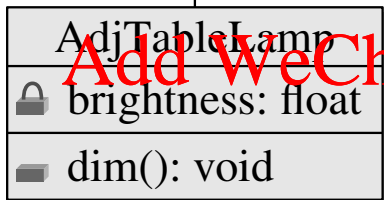
An adjustable table lamp inherits all properties of a table lamp; in addition it can be dimmed.



base class / superclass / parent class

<https://powcoder.com>

derives / inherits from



derived class / subclass / child class

```
class TableLamp {  
    enum {ON, OFF} state;  
public:  
    TableLamp() { state = ON; }  
    void pressSwitch() {  
        state = ( state == ON ? OFF : ON );  
        // dim(); // illegal!  
        // brightness = 0.4; // illegal!  
    }  
    friend ostream& operator<<(ostream& o,  
        const TableLamp& t) {  
        return o << (t.state == TableLamp::ON ?  
            " is on" : " is off" );  
    }  
};
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
class AdjTableLamp : public TableLamp {  
    float brightness;  
public:  
    AdjTableLamp() { brightness = 1.0; }  
    void dim() {  
        if (brightness > 0.1) brightness -= 0.1;  
    }  
    void print(ostream& o) const {  
        o << *this << " with brightness "  
          << brightness << endl;  
    }  
};
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
AdjTableLamp myLamp;
```

```
cout << "myLamp";
```

```
myLamp.print(cout);
```

```
// OUT: myLamp is on with brightness 1.0
```

```
myLamp.dim();
```

```
cout << "myLamp";
```

```
myLamp.print(cout);
```

```
// OUT: myLamp is on with brightness 0.9
```

```
myLamp.pressSwitch();
```

```
cout << "myLamp" << myLamp;
```

```
// OUT: myLamp is off
```

```
TableLamp yourLamp;
```

```
// yourLamp.dim();           illegal!
```

```
// yourLamp.print(cout);     illegal!
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder


```
AdjTableLamp* hisLamp = new AdjTableLamp();
```

```
cout << "hisLamp"; hisLamp->print(cout);
```

```
// OUT: hisLamp is on with brightness 1.0
```

```
hisLamp->dim();
```

```
cout << "hisLamp";
```

```
hisLamp->print(cout);
```

```
// OUT: hisLamp is on with brightness 0.9
```

```
hisLamp->pressSwitch();
```

```
cout << "hisLamp" << *hisLamp;
```

```
// OUT: hisLamp is off
```

```
TableLamp* herLamp = new TableLamp();
```

```
// herLamp->dim();          illegal!
```

```
// herLamp->print(cout);    illegal!
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

- objects of a derived class inherit all the members of the base class

e.g. `myLamp.pressSwitch()`, `hisLamp->pressSwitch()`

- but objects of the base class do not have access to the features of the derived class

- objects of a derived class may have additional features
e.g. `myLamp.dim()`, `hisLamp->print(cout)`

modification of existing features (overriding, redefining) will be discussed shortly

- objects of a derived class may be used where an object of a base class is expected (common interface)

e.g. `cout << myLamp`, `cout << *hisLamp`

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
AdjTableLamp* hisLamp = new AdjTableLamp();
```

```
TableLamp* herLamp = hisLamp; // OK  
// hisLamp = herLamp;         illegal!
```

```
AdjTableLamp myLamp;
```

```
TableLamp theirLamp;
```

```
herLamp = &myLamp; // OK  
// hisLamp = &theirLamp; illegal!
```

```
/* If we allowed the illegal assignments above  
what would happen if we then did:
```

```
    hisLamp->dim();
```

```
    hisLamp->print(cout)
```

```
*/
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Assignment Project Exam Help

- pointers to a derived class may be implicitly converted to pointers to a base class
e.g. `herLamp = hisLamp`, `herLamp = &myLamp`
- but not vice-versa

Add WeChat powcoder

Assignment and Inheritance: Objects

```
AdjTableLamp myLamp; TableLamp yourLamp;
```

```
myLamp.pressSwitch();
```

```
cout << "myLamp"; myLamp.print(-cout);
```

```
// OUT: myLamp is off with brightness 1.0
```

```
cout << "yourLamp" << yourLamp;
```

```
// OUT: yourLamp is on
```

```
yourLamp = myLamp;
```

```
cout << "yourLamp" << yourLamp;
```

```
// OUT: yourLamp is off
```

```
myLamp.pressSwitch();
```

```
cout << "myLamp" << myLamp;
```

```
// OUT: myLamp is on
```

```
cout << "yourLamp" << yourLamp;
```

```
// OUT: yourLamp is off
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Assignment and Inheritance: Pointers

```
AdjTableLamp* hisLamp = new AdjTableLamp();  
TableLamp* herLamp;
```

```
herLamp = hisLamp;
```

```
cout << "herLamp" << *herLamp;  
    // OUT: herLamp is on  
cout << "hisLamp"; hisLamp->print(cout);  
    // OUT: hisLamp is on with brightness 1.0
```

```
hisLamp->pressSwitch();
```

```
cout << "hisLamp"; hisLamp->print(cout);  
    // OUT: hisLamp is off with brightness 1.0  
cout << "herLamp" << *herLamp;  
    // OUT: herLamp is off
```

```
delete hisLamp;    // what's wrong with this?  
// delete herLamp    very bad!
```

- assignment to objects of derived class to objects of base class

```
yourLamp = myLamp
```

copies all data members defined in the base class

(`TableLamp`) from `myLamp` to `yourLamp` and does not change the class of the object assigned to (`yourLamp`)

- assignment of pointers

```
herLamp = hisLamp
```

Add WeChat powcoder

makes `herLamp` and `hisLamp` point to the same object, but still only features of `TableLamp` can be accessed on `herLamp`

Assignment and Inheritance: References

```
AdjTableLamp myLamp; TableLamp yourLamp;
```

```
AdjTableLamp& myLampRef = myLamp;
```

```
TableLamp& yourLampRef = yourLamp;
```

```
myLampRef.pressSwitch()
```

```
cout << "myLampRef"; myLampRef.print(cout);
```

```
// OUT: myLampRef is off with brightness 1.0
```

```
cout << "yourLampRef" << yourLampRef;
```

```
// OUT: yourLampRef is on
```

```
yourLampRef = myLampRef;
```

```
cout << "yourLampRef" << yourLampRef;
```

```
// OUT: yourLampRef is off
```

```
myLampRef.pressSwitch();
```

```
cout << "myLampRef"; myLampRef.print(cout);
```

```
// OUT: is on with brightness 1.0
```

```
cout << "yourLampRef" << yourLampRef;
```

```
// OUT: yourLampRef is off
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder


```
AdjTableLamp myLamp, myOtherLamp;
```

```
AdjTableLamp& myLampRef = myLamp  
TableLamp& myOtherLampRef = myOtherLamp;
```

```
myLampRef.dim(); myLampRef.pressSwitch();  
cout << "myLampRef"; myLampRef.print(cout);  
// OUT: myLampRef is off with brightness 0.9  
cout << "myOtherLampRef" << myOtherLampRef;  
// OUT: myOtherLampRef is on
```

```
myOtherLampRef = myLampRef;
```

```
cout << "myOtherLampRef" << myOtherLampRef;  
// OUT: myOtherLampRef is off  
cout << "myOtherLamp"; myOtherLamp.print(cout);  
// OUT: myOtherLamp is off with brightness 1.0
```

- assignment of references

```
yourLampRef = myLampRef
```

behaves like assignment of objects, i.e. it copies all data members defined in TableLamp from myLampRef to yourLampRef and does not change the class of the object aliased by yourLampRef

- if a TableLamp reference actually aliases an AdjTableLamp object,

```
myOtherLampRef = myLampRef
```

then assignment of references *still* behaves like assignment of objects, i.e. the AdjTableLamp attributes are *not* copied.

```
// But don't forget references are aliases
AdjTableLamp myLamp;
```

```
AdjTableLamp& myLampRef = myLamp
TableLamp& yourLampRef = myLamp;
```

```
cout << "myLampRef"; myLampRef.print(cout);
// OUT: myLampRef is on with brightness 1.0
```

```
yourLampRef.pressSwitch();
cout << "yourLampRef" << yourLampRef;
```

```
// OUT: yourLampRef is off
```

```
cout << "myLampRef"; myLampRef.print(cout);
// OUT: myLampRef is off with brightness 1.0
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

class TableLamp {
    ...
    friend ostream& operator<<(ostream& o,
                               TableLamp& t) {
        // If TableLamp& t not const
        ...
    }
};

```

```

class AdjTableLamp : public TableLamp {
    ...
    void print(ostream& o) const {
        o << *this ...
        // compile error: no match for operator<<
    }
};

```

Assignment Project Exam Help

<https://powcoder.com>

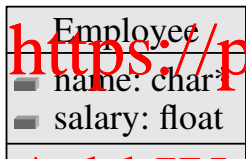
Add WeChat powcoder

Constructors and Inheritance

Base Class Initializers

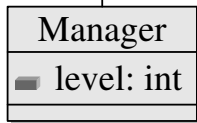
- the base class constructor must be called through a base class initializer

- if the base class constructor has arguments, then these arguments must be provided in the base class initializer



Employees have a name, and earn a salary.

Add WeChat powcoder



Managers are employees; thus they inherit all employee properties. They are paid according to their level.

```
class Employee {
```

```
protected:
```

```
    char* name;
```

```
    float salary;
```

```
public:
```

```
    Employee(float s, char* n) {
```

```
        salary = s;
```

```
        name = n;
```

```
    }
```

```
friend ostream& operator<< (ostream& o,
```

```
const Employee& e) {
```

```
    return o << e.name << " earns " << e.salary;
```

```
}
```

```
};
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
class Manager : public Employee {
```

```
private:  
    int level;
```

```
public:
```

```
    Manager(int l, char* n) : Employee(10000.0 * l, n) {  
        level = l;  
    }
```

```
    ostream& operator>>(ostream& o) const {  
        return o << *this << "at level " << level;  
    }  
};
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
int main() {  
  
    Manager Scrooge(5, "Scrooge MacDuck");  
    Employee Donald(13456.5, "Donald Duck");  
    cout << Donald << endl;  
        // OUT: Donald Duck earns 13456.5  
  
    cout << Scrooge << endl;  
        // OUT: Scrooge MacDuck earns 50000.0  
  
    Scrooge >> cout << endl;  
        // OUT: Scrooge MacDuck earns 50000.0 at level 5  
  
    return 0;  
}
```


Base class initializers are implicitly introduced by the compiler; this is why the following produces a compile time error:

```
class WrongManager : public Employee {  
private:  
    int level;  
public:  
    WrongManager(int l, char* n) {  
        level = l; name = n;  
        salary = 10000.0 * l;  
    }  
// Error: no matching function call to  
// 'Employee::Employee()'   
};
```

Question: Why was there no corresponding error in the constructor for AdjTableLamp?

- Objects are constructed from top to bottom: first the base class and then the derived class (first members then constructor)
- They are destroyed in the opposite order: first the derived class and then the base class (first destructor then members)

Example:

Employees are given a desk, and share offices. Bosses are employees, but they are also given PCs. On their first day at work, Bosses turn their PCs on; when they are fired they switch their PC off.

```
class Desk {  
public:  
    Desk() { cout << "Desk::Desk() \n"; }  
    ~Desk() { cout << "Desk::~~Desk() \n"; }  
};
```

```
class Office {  
public:  
    Office() { cout << "Office::Office() \n"; }  
    ~Office() { cout << "Office::~~Office() \n"; }  
};
```

```
class PC {  
public:  
    PC() { cout << "PC::PC() \n"; }  
    ~PC() { cout << "PC::~~PC() \n"; }  
  
    void turnOn() { cout << "turns PC on \n"; }  
    void turnOff() { cout << "turns PC off \n"; }  
};
```

```

class Empl {
    Desk myDesk;
    Office* myOffice;
public:
    Empl(Office* o) {
        myOffice = o;
        cout << "Empl::Empl() \n";
    }
    ~Empl() { cout << "Empl::~~Empl() \n"; }
};

class Boss : public Empl {
    PC myPC;
public:
    Boss(Office* o) : Empl(o) {
        myPC.turnOn(); cout << "Boss::Boss() \n";
    }
    ~Boss() {
        myPC.turnOff(); cout << "Boss::~~Boss() \n";
    }
};

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
int main() {  
  
    Office* pOff;  
    pOff = new Office();  
    // OUT: Office::Office()  
    Empl* pEmpl = new Empl(pOff);  
  
    /* OUT: Desk::Desk()  
       Empl::~Empl() */  
    delete pEmpl;  
  
    /* OUT: Empl::~Empl()  
       Desk::~Desk() */  
}
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Notice

The destructor for employees does not automatically destroy the office (nor should it - why?).

```
Boss* pBoss = new Boss(pOff);
```

```
/* OUT: Desk::Desk()
```

```
Empl::Empl()
```

```
PC::PC()
```

```
turns PC on
```

```
Boss::Boss() */
```

```
delete pBoss;
```

```
/* OUT: turns PC off
```

```
Boss::~Boss()
```

```
Empl::~Empl()
```

```
PC::~PC()
```

```
Desk::~Desk() */
```

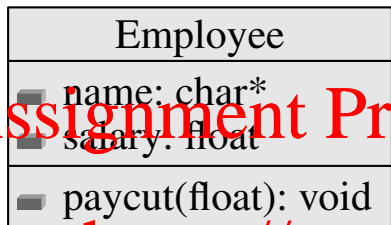
Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

So far, we only know half the truth about inheritance: its real power and usefulness lies in *virtual functions* (adding vs. modifying features)

- Method binding is the process of determining which method to execute for a given call.
- In C++ we have both static and dynamic method binding.
- When a virtual member function is called, the class of the receiver determines which function will be executed.
- The keyword `virtual` indicates that a function is virtual.



Employees have a name, and earn a salary. When they receive a pay cut, their salary is reduced by the specified amount.

Managers are employees; When they receive a pay cut their salary is incremented by the specified amount multiplied by their level.

The function `paycut(float)` will be implemented as a virtual member function.


```
class Employee {
```

```
protected:
```

```
    char* name;
```

```
    float salary;
```

```
public:
```

```
    Employee(float s, char* n) { salary = s; name = n;}
```

```
    friend ostream& operator<<(ostream& o,
```

```
        const Employee& e) {
```

```
        return o << e.name << " earns " << e.salary;
```

```
    }
```

```
    virtual void paycut(float amount) {
```

```
        salary -= amount;
```

```
    }
```

```
};
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
class Manager : public Employee {
```

```
private:
```

```
    int level;
```

```
public:
```

```
    Manager(int l, char* n) : Employee(10000.0 * l, n){  
        level = l;
```

```
    }
```

```
friend ostream& operator<< (ostream& o,
```

```
    const Manager& m) {
```

```
    return o << (Employee) m << " at level "
```

```
    << m.level;  
}
```

```
virtual void paycut(float amount) {
```

```
    salary += amount * level;
```

```
}
```

```
};
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

The function

```
virtual void paycut(float amount)
```

is virtual but the operators

```
ostream& operator<<(ostream& o, const Employee& e)
```

```
ostream& operator<<(ostream& o, const Manager& m)
```

are overloaded - not virtual.

- The construction (Employee) m is called a type cast. It requires the compiler to consider m as being of type Employee (also see static_cast<> and dynamic_cast<>)
- Note: downward type casts can be dangerous

```
int main()
{
    Manager Scrooge(5, "SMD");
    Employee Donald(13456.5, "DD");
    cout << Donald << endl;
    // OUT:    DD earns 13456.5
    Donald.paycut(300);
    cout << Donald << endl;
    // OUT:    DD earns 13156.5
    cout << Scrooge << endl;
    // OUT:    SMD earns 50000.0 at level 5
    Scrooge.paycut(300);
    cout << Scrooge << endl;
    // OUT:    SMD earns 51500.0 at level 5
    Scrooge.Employee::paycut(300);
    cout << Scrooge << endl;
    // OUT:    SMD earns 51200.0 at level 5
    return 0;
}
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

- We distinguish between static and dynamic binding for functions.

- Static binding: function to be executed is determined at compile-time

- Dynamic binding: function to be executed can only be determined at run-time

- In C++, virtual functions are bound dynamically if the receiver is a pointer, i.e. `pointer->f(...)` is bound dynamically if `f` is virtual

- All other functions (virtual or non-virtual) are bound statically according to the class of the object executing the function

- The most powerful effect is produced by the combination of virtual functions and pointers.

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Assignment Project Exam Help

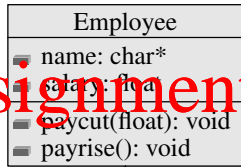
Language Design Philosophy

In other OO languages, e.g. C# or Java, there is only dynamic binding.

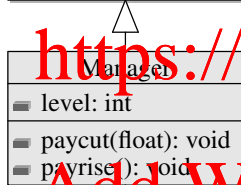
<https://powcoder.com>

- Which mode is more important for OO?
- Why are there two modes of binding in C++?

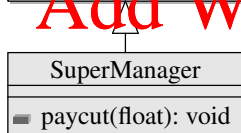
Add WeChat powcoder



Employees have a name, and earn a salary. `paycut`, reduces their salary by the specified amount; `payrise` increases salary by 800.



Managers are employees. When they receive a `paycut` their salary is incremented by the specified amount multiplied by their level. A `payrise` increases salary by 100.



SuperManagers are Managers. They double their salary when they get a `paycut`.

In order to demonstrate the issues around virtual functions, we declare:

```
Employee::paycut(float) as a virtual function
```

```
Employee::payrise() as a non-virtual function
```

In addition, we say that the function

```
Manager::payrise()
```

redefines

```
Employee::payrise()
```

and

```
Manager::paycut()
```

overrides

```
Employee::paycut()
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder


```
class Employee {
```

```
... // same as before
```

```
friend ostream& operator<<(ostream& o,  
    const Employee& e) {  
    return o << e.name << " earns " << e.salary;
```

```
}  
  
virtual void paycut(float amount) {  
    salary -= amount;
```

```
}  
  
void payrise() { salary += 800; }
```

```
};
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
class Manager : public Employee {
```

```
... // same as before
```

```
friend ostream& operator<<(ostream& o,  
const Manager& m) {  
    return o << (Employee) m << " at level "  
        << m.level;  
}
```

```
virtual void paycut(float amount) {  
    salary += amount * level;  
}
```

```
void payrise() { salary += 100; }
```

```
};
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Assignment Project Exam Help

```
class SuperManager : public Manager {  
public:  
    SuperManager(char* n) : Manager(10, n) {}  
  
    virtual void payout(float amount) {  
        salary *= 2;  
    }  
};
```

<https://powcoder.com>

Add WeChat powcoder

```
int main() {  
  
    Manager* M1 = new Manager(5, "ScrMcDuck");  
    Employee* E1 = new Employee(13456.5, "DonDuck");  
    Employee* E2 = new SuperManager("ValtDisn");  
  
    cout << "E1: " << *E1 << endl;  
    // OUT: E1: DonDuck earns 13456.5  
  
    E1->paycut(300);  
    cout << "E1: " << *E1 << endl;  
    // OUT: E1: DonDuck earns 13156.5  
  
    E1->payrise();  
    cout << "E1: " << *E1 << endl;  
    // OUT: E1: DonDuck earns 13956.5  
}
```

```
cout << "M1: " << *M1 << endl;  
// OUT: M1: ScrMcDuck earns 50000 at level 5
```

```
M1->paycut(300);
```

```
cout << "M1: " << *M1 << endl;  
// OUT: M1: ScrMcDuck earns 45500 at level 5
```

```
M1->payrise();
```

```
cout << "M1: " << *M1 << endl;  
// OUT: M1: ScrMcDuck earns 51600 at level 5
```

```
cout << "E2: " << *E2 << endl;  
// OUT: E2: WaltDisn earns 100000
```

```
E2->paycut(300);
```

```
cout << "E2: " << *E2 << endl;  
// OUT: E2: WaltDisn earns 200000
```

```
E2->payrise();
```

```
cout << "E2: " << *E2 << endl;  
// OUT: E2: WaltDisn earns 200800
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Assignment Project Exam Help

```
E2 = E1;  
cout << "E2: " << *E2 << endl;  
// OUT: E2: DonDuck earns 13956.5
```

```
E2->paycut(300);  
cout << "E2: " << *E2 << endl;  
// OUT: E2: DonDuck earns 13656.5
```

```
E2->payrise();  
cout << "E2: " << *E2 << endl;  
// OUT: E2: DonDuck earns 14456.5
```

```
Employee Donald(30000.0, "Donald Duck");  
SuperManager Walter("Walter Disney");
```

```
cout << "Donald: " << Donald << endl;  
// OUT:   Donald: Donald Duck earns 30000
```

```
Donald.paycut(100);  
cout << "Donald: " << Donald << endl;  
// OUT:   Donald: Donald Duck earns 29700
```

```
Donald.payrise(5);  
cout << "Donald: " << Donald << endl;  
// OUT:   Donald: Donald Duck earns 30500
```

Static Binding for Objects Cont.

```
cout << "Walter: " << Walter << endl;
// OUT:   Walter: Walter Disney earns 100000 at
//        level 10
```

```
Donald = Walter;
// Object assignment (copying of fields)
```

```
cout << "Donald: " << Donald << endl;
// OUT:   Donald: Walter Disney earns 100000
```

```
Donald.paycut(300);
```

```
cout << "Donald: " << Donald << endl;
// OUT:   Donald: Walter Disney earns $9700
```

```
Donald.payrise();
```

```
cout << "Donald: " << Donald << endl;
// OUT:   Donald: Walter Disney earns 100500
```

```
return 0;
```


Assignment Project Exam Help

- `E2->paycut(300)` which results in calling
 - `Employee::paycut(float)` if E2 points to an object of class Employee
 - `Manager::paycut(float)` if E2 points to an object of class Manager
 - `SuperManager::paycut(float)` if E2 points to an object of class SuperManager
- `paycut(float)` is a virtual function

- `cout << *E2` which always results in calling the `ostream& operator<<(ostream&, Employee&)` function even if E2 points to an object of class Manager or class SuperManager
- `E2->payrise()`, which always results in calling `Employee::payrise()` even if E2 points to an object of class Manager or class SuperManager.
- `Donald.paycut(300)`, which always results in calling `Employee::paycut(float)` even after the assignment `Donald = Walter`

- the class of the object executing the member function determines which function will be executed
 - for objects, the class of the object is known at compile time, therefore static binding
 - for pointers (and references), the class of object is unknown at compile time, therefore dynamic binding (but only if the function is virtual).
- The difference between virtual functions (overriding) and non-virtual (redefining) functions, e.g.
`Employee::paycut(int)` vs `Employee::payrise()` is subtle
- in general: if a function should behave differently in subclasses, then it should be declared virtual

- static binding results in faster programs; dynamic binding allows for flexibility at run-time
- programmers should use dynamic binding only when necessary
- C++ aims for:

- as much static binding as possible (i.e. for non-virtual functions, for non pointer receivers)
- dynamic binding only when necessary (i.e. only for calls of virtual function if the receiver is a pointer)

The type system (compiler) guarantees that when you access a member variable or member function, then the receiver will always have such a member, i.e. invoking methods and accessing fields always leads to well-defined behaviour.

What makes a function virtual?

- virtual functions are preceded by the keyword `virtual`
- a function with same identifier and arguments in a subclass is virtual as well (also see `override` specifier since C++11)

```
class Food {  
public:  
    virtual void print() { cout << " food \n"; }  
};
```

```
class FastFood: public Food {  
public:  
    void print() { cout << "fast food \n"; }  
};
```

```
class Pizza: public FastFood {  
public:  
    void print() { cout << " salami, pepperoni \n"; }  
};
```

The functions `Food::print()`, `FastFood::print()` and `Pizza::print()` are all virtual, even though only the function `Food::print()` contains the keyword `virtual` in its declaration.

In other words, the keyword `virtual` in the declaration of the overriding functions can be omitted.

```
int main() {  
    Food* f; f = new Food; f->print();  
    // OUT: food  
    f = new FastFood; f->print();  
    // OUT: fast food  
    f = new Pizza; f->print();  
    // OUT: salami, pepperoni  
    FastFood* ff; ff = new FastFood; ff->print();  
    // OUT: fast food  
    ff = new Pizza; ff->print();  
    // OUT: salami, pepperoni  
}
```

Assignment Project Exam Help

- for a member function `f`, inside code of the containing class, the function call `f(...)` corresponds to `this->f`
- thus local function calls can be bound dynamically

Add WeChat powcoder

Assignment Project Exam Help

```
class Human {  
public:  
    void holiday() {  
        cout << "spends holidays ";  
        enjoying();  
    }  
};
```

<https://powcoder.com>

```
    virtual void enjoying() {  
        cout << "relaxing\n";  
    }  
};
```

Add WeChat powcoder


```
class Italian: public Human {  
public:  
    void enjoying() override {  
        cout << " on the beach\n";  
    }  
};
```

```
class Swede: public Human {  
public:
```

```
    void enjoying() override {  
        cout << "in the sauna\n";  
    }  
};
```

```
int main() {  
    Italian Giuseppe; Swede Stefan;  
    cout << "Giuseppe "; Giuseppe.holiday();  
    // OUT: Giuseppe spends holidays on the beach  
    cout << "Stefan "; Stefan.holiday();  
    // OUT: Stefan spends holidays in the sauna  
}
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

This example demonstrates the Template Method Design Pattern

When the behaviour of objects of different classes bears some similarities, but differences in some aspects, then one should extract the common behaviour into a member function of a superclass, and express the differing aspects through the call of virtual functions.

<https://powcoder.com>

Such an approach:

- supports reuse of code
- more maintainable
- clarifies similarities, stresses differences

Assignment Project Exam Help

- Destructors are bound according to the same rules as any other member function - in particular, they can be virtual.
- There are no virtual constructors. Cloning operators and factory design patterns play this role.

<https://powcoder.com>

Add WeChat powcoder

```
class Empl {  
public:  
    ~Empl() { cout << "Empl::~Empl() \n"; }  
};
```

```
class Boss : public Empl {  
public:  
    ~Boss() { cout << "Boss::~Boss() \n"; }  
};
```

```
class EmplV {  
public:  
    virtual ~EmplV() { cout << "EmplV::~EmplV() \n"; }  
};
```

```
class BossV : public EmplV {  
public:  
    ~BossV() { cout << "BossV::~BossV() \n"; }  
};
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

int main() {

    Empl* pEmpl = new Boss();
    delete pEmpl;
    // OUT: Empl::Empl()
    EmplV* pEmplV = new BossV();
    delete pEmplV;
    /* OUT: BossV::~~BossV()
       EmplV::~~EmplV() */

    return 0;
}

```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

- it is a good policy to always make destructors virtual
- also solves the issue of undefined behaviour if deleting objects of subclasses through base class pointers (see TableLamp example)

- Overloading:

- if several function declarations share the same name
 - appropriate function body is selected by comparing the types of actual arguments with the types of formal parameters (function signature)

- inheritance is not required for overloading to occur

- Overriding: a virtual function f defined in a derived class D overrides a virtual function f defined in a base class B , if the two functions share the same parameter types. ($D::f$ is allowed to return a subtype of the return type of $B::f$.)
Calling f on an object of class D invokes $D::f$.

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Types are known at compile time, therefore:

- Overloading is resolved statically according to the compile time type of the arguments.

Example of overloading:

Humans chat with one another; when one human meets another, then they (invariably) talk about the weather. If a person meets someone that they know is a computer scientist, then they talk about how computer illiterate they are.

Assignment Project Exam Help

```
class Human {  
public:  
    void chatsWith(Human h) {  
        cout << "about the weather";  
    }  
};
```

```
void chatsWith(ComputerScientist c) {  
    cout << "about their computer illiteracy";  
}  
};
```

Add WeChat powcoder

```
class ComputerScientist : public Human {};
```



```
int main() {  
    Human* someone;  
    Human* someone_else;  
    Human john;  
    ComputerScientist julia;  
    someone = new Human;
```

```
    john.chatsWith(*someone);  
    // OUT: about the weather  
    john.chatsWith(julia);  
    // OUT: about their computer illiteracy
```

```
    someone = &julia;  
    john.chatsWith(*someone);  
    // OUT: about the weather  
    someone_else->chatsWith(john);  
    // OUT: about the weather  
    // why does an uninitialized pointer  
    // not cause errors?  
}
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Classes are known only at run time, therefore:

- Overriding is resolved according to the run time class of the receiver, if possible statically, otherwise dynamically.
- Notice that functions may be involved in both overloading and overriding.

Example of overriding:

When a computer scientist meets someone else, they talk about computer games; when they meet another computer scientist, then they chat about other people's computer illiteracy!

```

class Human {
public:
    virtual void chatsWith(Human* h) {
        cout << "about the weather";
    }
    virtual void chatsWith(ComputerScientist* c) {
        cout << "about their own computer illiteracy";
    }
};

class ComputerScientist : public Human {
public:
    virtual void chatsWith(Human* h) {
        cout << "about computer games";
    }

    virtual void chatsWith(ComputerScientist* c) {
        cout << "about others' computer illiteracy";
    }
};

```

```
int main() {
```

```
    Human John, *Paul;
```

```
    ComputerScientist Julia, *Paola;
```

```
    John.chatsWith(Paul);
```

```
        // OUT: about the weather
```

```
    John.chatsWith(Paola);
```

```
        // OUT: about their own computer illiteracy
```

```
    Julia.chatsWith(Paul);
```

```
        // OUT: about computer games
```

```
    Julia.chatsWith(Paola);
```

```
        // OUT: about others' computer illiteracy
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
Human* Han = new Human;
```

```
Han->chatsWith(Paul);  
// OUT: about the weather
```

```
Han->chatsWith(Paula);  
// OUT: about their own computer illiteracy
```

```
Han = new ComputerScientist;  
Han->chatsWith(Paul);  
// OUT: about computer games
```

```
Han->chatsWith(Paula);  
// OUT: about others' computer illiteracy
```

```
Paul->chatsWith(Han);  
// Segmentation fault!      Why?
```

```
}
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

Assignment Project Exam Help

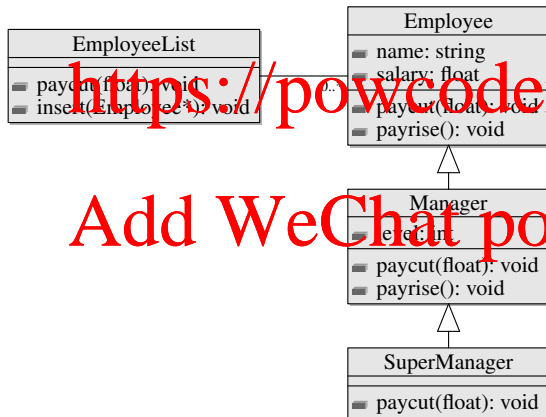
- `Human::chatsWith(Human*)` overloads `Human::chatsWith(ComputerScientist*)`
- `ComputerScientist::chatsWith(Human*)` overloads `ComputerScientist::chatsWith(ComputerScientist*)`
- `ComputerScientist::chatsWith(Human*)` overrides `Human::chatsWith(Human*)`
- `ComputerScientist::chatsWith(ComputerScientist*)` overrides `Human::chatsWith(ComputerScientist*)`

<https://powcoder.com>

Add WeChat powcoder

Polymorphism and Dynamic Binding

Polymorphism allows us to uniformly define and manipulate structures consisting of objects which share some characteristics, but still differ in some details. Consider a list containing employees and/or managers and/or supermanagers:



```
// classes Employee, Manager, SuperManager as before
```

```
class EmployeeList {
```

```
    EmployeeList* next;
```

```
    Employee* theEmployee;
```

```
public:
```

```
    EmployeeList(Employee* e) : theEmployee(e),  
        next(nullptr) {}
```

```
void insert(Employee* e) {
```

```
    EmployeeList* newList = new EmployeeList(e);  
    newList->next = next;  
    next = newList;
```

```
}
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder


```
friend ostream& operator<<(ostream& o,  
    const EmployeeList& l){
```

```
    o << *(l.theEmployee) << endl;
```

```
    return (l.next == nullptr) ? o << endl  
                                   : o << *(l.next);
```

```
}
```

```
void paycut(float a) {
```

```
    theEmployee->paycut(a);
```

```
    if (next != nullptr) next->paycut(a);
```

```
}
```

```
};
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
int main() {  
  
    EmployeeList disneyList(  
        new Manager(5, "Scrooge Mac Duck")  
    );  
    disneyList.insert(  
        new Employee(13456.5, "Donald Duck")  
    );  
    disneyList.insert(  
        new SuperManager("Walter Disney")  
    );  
    disneyList.insert(  
        new Employee(45.7, "Louie Duck")  
    );  
  
    ...  
}
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
cout << disneyList;
/* OUT: Scrooge Mac Duck earns 50000
      Louie Duck earns 45.7
      Walter Disney earns 100000
      Donald Duck earns 13456.5 */
```

```
disneyList.payout(40);
cout << disneyList;
/* OUT: Scrooge Mac Duck earns 50200
      Louie Duck earns 5.7
      Walter Disney earns 200000
      Donald Duck earns 13416.5 */
```

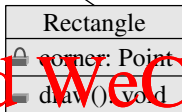
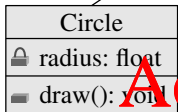
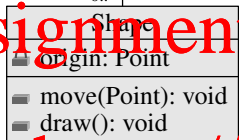
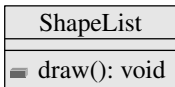
```
return 0;
}
```

Each element in the list reacts differently to the payout, according to its (dynamically determined) class.

- When we only need a function to define an interface, we can leave its implementation unspecified

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

- This is called a pure virtual function.
- A class that contains at least one pure virtual function is called an abstract class.
- No objects of an abstract class may be created.
- Client code knows that all objects of derived classes provide this function.
- Classes that inherit pure virtual functions and do not override them are also abstract.



```

class Shape{
    Point origin;
public:
    void move(Point p)
        { origin += p; };
    virtual void draw()=0;
}
  
```

```

class ShapeList{
    Shape* theShape;
    ShapeList* next;
public:
    void draw()
        { theShape->draw();
          ... next->draw(); }
};
  
```

```

class Circle: public Shape{
    float radius;
public:
    void draw(){    }};
  
```

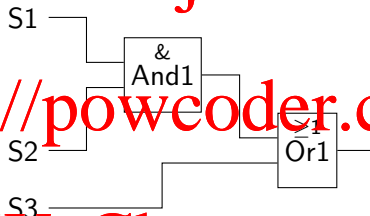
The pure virtual function draw() is indispensable!

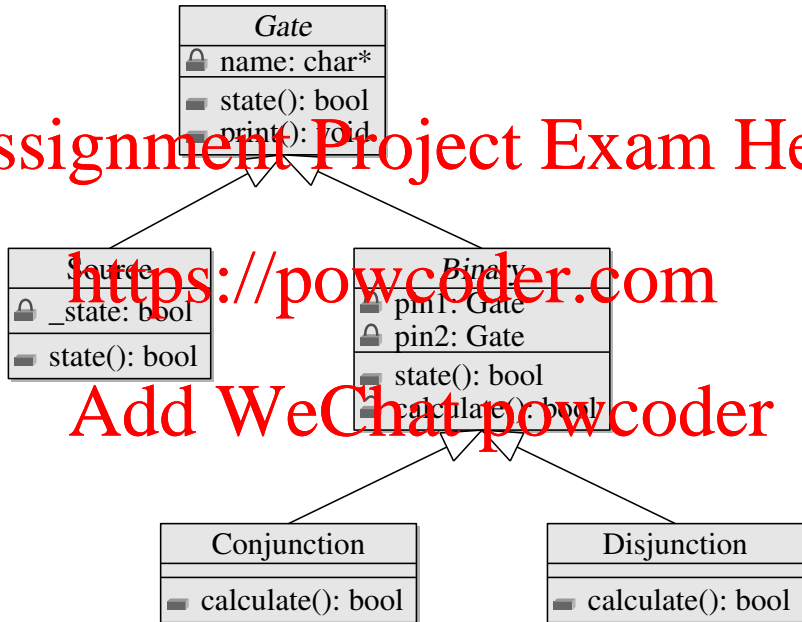
- The clients of the abstract class may safely expect all the objects of subclasses of this class to implement all the public functions.
- All non-abstract subclasses of the abstract class are under the obligation to provide an implementation of the pure virtual functions
- If ClassA and ClassB are similar, but none is necessarily more general than the other, then they probably should both be subclasses of a new, common abstract superclass, ClassC.

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder





Assignment Project Exam Help

In this example, we will make use of Member Access Control.

Class members can be:

- `public` - may be used anywhere.
- `protected` - may be used by any member function of the class and by any member function of any subclass.
- `private` - may be used by member functions of the class only.

Furthermore, a friend of a class can access everything the class has access to.

```
class Gate {
```

```
    const char* name; // private is the default
```

```
public:
```

```
    Gate(const char* name) : name(name) {}
```

```
    virtual bool state() = 0; // Pure virtual function
```

```
    virtual void print() { cout << name; }
```

```
};
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
class Source : public Gate {
```

```
    bool _state;
```

```
public:
```

```
    Source(const char* name, bool state)
        : Gate(name), _state(state) {}
```

```
    bool state() override { return _state; }
```

```
    void print() override {
```

```
        Gate::print();
```

```
        cout << (_state ? "(1)" : "(0)"),
```

```
    }
```

```
};
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
class Binary : public Gate {  
  
    Gate &pin1, &pin2;  
  
protected:  
    virtual bool calculate(bool state1, bool state2)=0;  
public:  
    Binary(const char* name, Gate& pin1, Gate& pin2)  
        : Gate(name), pin1(pin1), pin2(pin2) {}  
    bool state() override {  
        return calculate(pin1.state(), pin2.state());  
    }  
    void print() override {  
        Gate::print();  
        cout << "["; pin1.print();   cout << ",";  
        pin2.print(); cout << "]"  
    }  
};
```

```
class Conjunction : public Binary {
protected:
    bool calculate(bool state1, bool state2) override {
        return state1 && state2;
    }
public:
    And(const char* name, Gate& pin1, Gate& pin2)
        : Binary(name, pin1, pin2) {}
};

class Disjunction : public Binary {
protected:
    bool calculate(bool state1, bool state2) override {
        return state1 || state2;
    }
public:
    Or(const char* name, Gate& pin1, Gate& pin2)
        : Binary(name, pin1, pin2) {}
};
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

int main()
{
    Gate g1("S1");
    // Compile error: cannot instantiate abstract class

    Source s1("S1", true);
    Source s2("S2", false);
    Source s3("S3", true);

    Binary* o1 = new Binary("And", s1, s2);
    // Compile error: cannot instantiate abstract class

    Binary* a1 = new Conjunction("And1", s1, s2);
    Gate* o1 = new Disjunction("Or", a1, s3);

    // Calling pure virtual function on abstract class
    bool state = o1->state();

    ...
}

```

Assignment Project Exam Help

```

cout << "State of ";
o1->print();
cout << (state ? " is 1 \n" : " is 0 \n");
// Gate::State o1 Or1[And1[S1(-1),S2(0)],S3(1)] is 1
return 0;

```

}

The example above also demonstrates how superclass implementation can be called from an overriding function (Gate::print)

Add WeChat powcoder

Assignment Project Exam Help

Member functions and friends of a class:

- can access protected members of its superclass directly
- cannot access protected non-static members of its superclass through a variable (obj, ptr, ref) of the superclass type

<https://powcoder.com>

Add WeChat powcoder


```
class Employee {
protected:
    float salary;
    static const float SALARY_STEP;
};

const float Employee::SALARY_STEP = 800.0;

class SuperManager;

class Manager : public Employee {
public:
    void blame(Employee* e);
    void think(SuperManager* s);
    friend void punish(Manager* m);
};

class SuperManager : public Manager {};
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
void Manager::blame(Employee* e) {  
    salary += Employee::SALARY_STEP;  
  
    e->salary -= Employee::SALARY_STEP;  
    // Compile error: cannot access protected member  
    // 'salary' from Employee *  
}
```

```
void Manager::thank(SuperManager* s) {  
    s->salary += SuperManager::SALARY_STEP;  
}
```

```
void punish(Manager* m) {  
    m->salary -= Employee::SALARY_STEP;  
    // ((SuperManager*) m)->salary is accessible  
    // if m is also a SuperManager  
    // ((Employee*) m)->salary is inaccessible  
}
```

The base class of a derived class may be specified public, protected or private. The access specifier affects the extent to which the derived class may inherit from the superclass, and whether clients may treat objects of a subclass as if they belonged to the superclass.

```
class Goldfish : public Animal {
```

- private members of Animal inaccessible in Goldfish
- protected members of Animal become protected members of Goldfish
- public members of Animal become public members of Goldfish
- any function may implicitly transform a Goldfish* to an Animal*

Access Specifiers for Base Classes - Cont.

```
class Stack : protected List /*.... */
```

- private members of List inaccessible in Stack
- protected and public members of List become protected members of Stack
- only friends and members of Stack and friends and members of Stack's derived classes may implicitly transform a Stack* to a List*

```
class AlarmedDoor : private Alarm /*... */
```

- private members of Alarm inaccessible in AlarmedDoor
- protected and public members of Alarm become private members of AlarmedDoor
- only friends and members of AlarmedDoor may implicitly transform an AlarmedDoor* to an Alarm*

Assignment Project Exam Help

The interplay of access modifiers is quite sophisticated. For our course, we concentrate on the use of access modifiers for class members, distinguish between private and public derivation, but do not worry about the distinction between private and protected derivation.

<https://powcoder.com>

Add WeChat powcoder

- An Alarm is active or inactive, and has the ability to call the police. It is activated through the function call `set()` and deactivated through `reset()`. One can query its status by calling `isActive()`.
- <https://powcoder.com> AlarmedDoors have a code which controls the door when entering the correct code one can deactivate/activate the door alarm. When one opens the door (`open()`), if the alarm is activated the police is called.
- We want to use the features of an Alarm to implement AlarmedDoor ... but is an AlarmedDoor a type of Alarm?

```
class Alarm {  
  
    bool state;  
  
public:  
    Alarm() { set(); }  
  
    void set() { state = true; }  
    void reset() { state = false; }  
  
    bool isActive() const { return state; }  
    void callPolice() {  
        cout << "Police are on the way!" << endl;  
    }  
  
};
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```

class AlarmedDoor : private Alarm {
    int code;
public:
    AlarmedDoor(int code) : Alarm(), code(code) {}

    void enterCode(int codeEntered) {
        if (code == codeEntered) {
            cout << "Code correct.";
            if (isActive()) {
                reset();
                cout << "Door alarm is now deactivated."
                    << endl;
            } else {
                set();
                cout << "Door alarm is now activated."
                    << endl;
            }
        }
        else { cout << "Code incorrect." << endl; }
    }
}

```



```
using Alarm::isActive;
```

```
void open() {  
    if (isActive())  
        callPolice();  
    else  
        cout << "Access granted " << endl;  
}  
};
```

```
int main() {  
    AlarmedDoor ad(1357);  
  
    if (ad.isActive())  
        cout << "Door alarm is active." << endl;  
    // OUT: Door alarm is active.
```

```
ad.enterCode(1357);  
// OUT: Code correct. Door alarm is now deactivated.  
ad.enterCode(2468);  
// OUT: Code incorrect.  
ad.open();  
// OUT: Access granted.  
ad.enterCode(1357);  
// OUT: Code correct. Door alarm is now activated.  
ad.open();  
// OUT: Police are on the way!
```

```
Alarm* a = &ad;  
//Compile error: Conversion to inaccessible base class  
//If we could do this, we can then do the following  
//a->reset();  
//ad.open();  
  
return 0;  
}
```

So what is private inheritance for?

Assignment Project Exam Help

- Private derivation corresponds to an is-implemented-in-terms-of relationship (does not exist in the real world but implementation domain; compare to *is-a*).
- One can almost always use composition instead.
- The only benefit of private derivation is that
 - one less level of indirection (slightly less code to write)
 - you can access protected members of the base class

<https://powcoder.com>
Add WeChat powcoder

- Static binding results in faster programs.
- Dynamic binding allows for flexibility at run-time.
- Access modifiers restrict who knows what.

C++ allows you to program so that

- there is as much static binding as possible
- dynamic binding is used only when necessary
- code and objects operate on a *need to know* basis

The compiler guarantees that at run time

- objects always know how to handle method calls
- non-existent fields are not accessed
- variables contain objects of class, or subclass of their definition.

Assignment Project Exam Help

- derived classes inherit characteristics from base class
- derived class objects may be used wherever base class objects expected
- derived classes may override virtual functions
- virtual functions are bound according to class of receiver
- virtual functions are bound dynamically for pointers/references
- polymorphic structures may be programmed using pointers/references

<https://powcoder.com>

Add WeChat powcoder

Assignment Project Exam Help

- pure virtual functions declare but do not define a function
- abstract classes provide interface for their subclasses
- member access control supports encapsulation

<https://powcoder.com>

Add WeChat powcoder

Assignment Project Exam Help

- use different object types (classes) to reflect different logical entities
- let each object do the work it is responsible for
- distinguish between *is a*, *has a*, and *behaves as a*
- reuse code (via inheritance) as much as possible

<https://powcoder.com>

Add WeChat powcoder