

The Object Oriented Paradigm in C++

Object Oriented = Object Based + Inheritance

Derived Classes

WS, ch 15

- a derived class is defined by adding (or modifying) features to (of) an existing class without reprogramming (no removing of features possible)

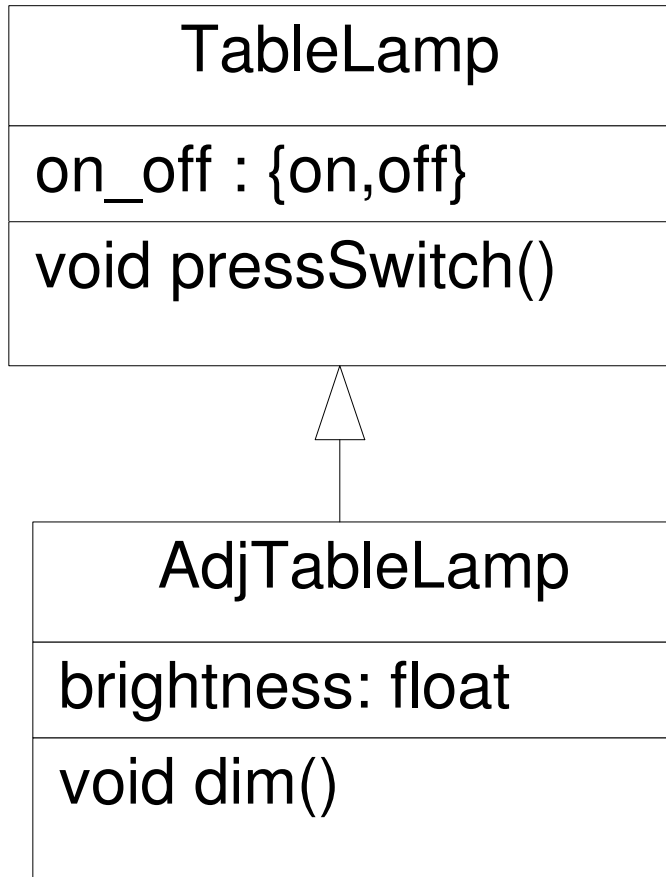
Thus

- common interface for several related, but not identical classes; objects of these classes may be manipulated identically by other parts of the program
- we obtain a taxonomy, and a better understanding of the situation
- derived classes inherit characteristics, and code is reused

taxonomy: principles of classification, esp. in biology

(Concise Oxford Dictionary)

Example: Table Lamp, Adjustable Table Lamp WS p 816-821



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

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

`TableLamp` is the *base class* of `AdjTableLamp`, and `AdjTableLamp` is *derived from* `TableLamp`.

Also, TableLamp is a *superclass* of AdjTableLamp, and AdjTableLamp is a *subclass* of TableLamp.

The above is broken into five files:

1. TableLamp.h
2. TableLamp.cpp
3. AdjTableLamp.h
4. AdjTableLamp.cpp
5. LampsMain.cpp

```
/* in file TableLamp.h */
#include <iostream>
class TableLamp {
    enum state {ON, OFF} on_off;
public:
    TableLamp();
    void pressSwitch();
    friend ostream& operator
        <<(ostream&, const TableLamp& t); };
```

```

/* in file AdjTableLamp.h */
#include "TableLamp.h"
class AdjTableLamp: public TableLamp{
    float brightness;
public:
    AdjTableLamp();
    void dim();
    void print(ostream&);
};

```

```

// in file TableLamp.cpp
#include "TableLamp.h"
TableLamp::TableLamp() { on_off = ON; }

void TableLamp::pressSwitch()
{ // dim();           illegal!
  // brightness = 0.4;   illegal!
  on_off = (( on_off == ON) ? OFF : ON); }

```

```

ostream& operator <<(ostream& o, const TableLamp& t)
    { return
      ((t.on_off==0)?o<< " is on" : o<<" is off");}

// in file AdjTableLamp.cpp
#include "AdjTableLamp.h"

AdjTableLamp::AdjTableLamp() { brightness = 1.0; }

void AdjTableLamp::dim()
    { if (brightness>0.1) brightness -= 0.1;}

void AdjTableLamp::print(ostream& o)
    { o<<*this<<" , with brightness"<<brightness<<endl;}

// in LampsMain.cpp
// simple demonstration of inheritance
#include "AdjTableLamp.h"
void main(void) {
    AdjTableLamp yourLamp;

```

```

cout << yourLamp << endl;
    // output          is on
yourLamp.print(cout);
    // output          is on with brightnes 1.0

yourLamp.pressSwitch(); yourLamp.dim();
yourLamp.print(cout);
    // output          is off with brightnes 0.9

// now, the same with pointers
AdjTableLamp *herLamp = new AdjTableLamp();
cout << *herLamp << endl;
    // output          is on
herLamp->print(cout);
    // output          is on with brightnes 1.0

herLamp->pressSwitch(); herLamp->dim();
herLamp->print(cout);
    // output          is off with brightnes 0.9
};

```

The previous example demonstrates:

- objects of a derived class inherit all members of the base class, e.g.

```
yourLamp.pressSwitch()
```

- objects of a derived class may have additional features, e.g.

```
yourLamp.dim(),
```

- objects of a derived class may appear where an object of base class expected.

```
cout << yourLamp
```

- the above observations also apply to objects obtained by dereferencing pointers of derived classes, e.g.

```
herLamp->pressSwitch()      or  
herLamp->dim()              or  
cout << *herLamp
```

- obviously, objects of the base class do *not* have access to the features of the derived class

Derived Classes and Type Compatibility - Objects

- Objects of a derived class may appear wherever an object of a base class is expected. WS, p 841-845

```
#include "AdjTableLamp.h"
void main(void) {
    TableLamp myTL;
    myTL.pressSwitch();
    cout << myTL << endl;
        // output          is off
// myTL.dim();           illegal!
// myTL.print(cout);     illegal!

    AdjTableLamp yourATL;
    yourATL.print(cout);
        // output          is on with brightness 1.0
    myTL = yourATL;
    cout << myTL;
        // output          is on
}
```

```
// myTL.print(cout);      illegal!  
// myTL.dim();           illegal!  
// yourATL = myTL;      illegal!  
}
```

The above example demonstrates

- an `AdjTableLamp` may be used where a `TableLamp` expected, but not the opposite.
- assignment of objects

```
obj1 = obj2;
```

copies all data members defined in `Class1` from `obj2` to `obj1`, where

`Class1` is the class of `obj1`, and

`Class2` is the class of `obj2`, and

`Class2` is a subclass of `Class1`

and does not change the class of `obj1`.

Derived Classes and Type Compatibility - Pointers

- Pointers to a derived class may be implicitly converted to pointers to a base class. WS, p 841-845

```
#include "AdjTableLamp.h"
void main(void) {
    TableLamp* myTLp = new TableLamp() ;
    myTLp->pressSwitch();
    // myTLp->dim();           illegal!

    AdjTableLamp* yourATLp = new AdjTableLamp;
    myTLp = yourATLp;
    // yourATLp = myTLp;     illegal!
};
```

The above example demonstrates

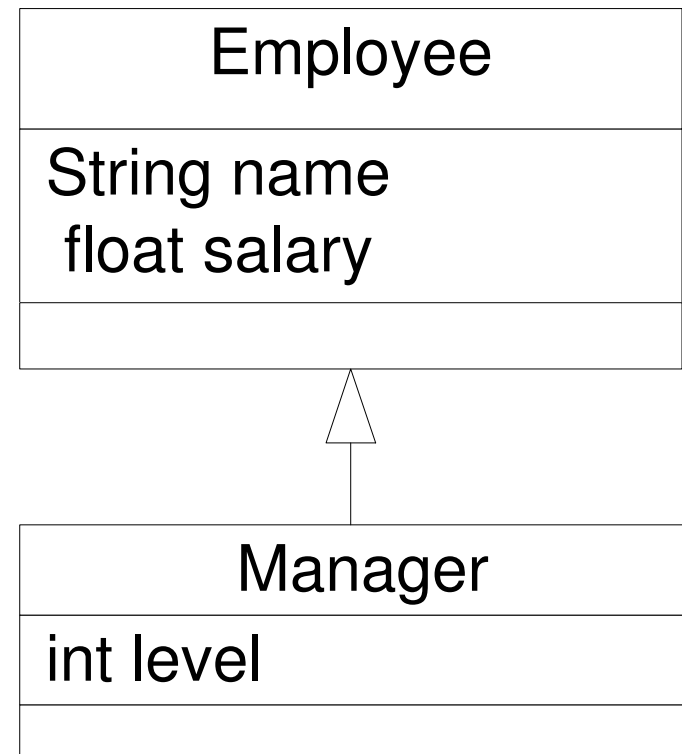
- a `AdjTableLamp*` may be used where a `TableLamp*` expected, but not the opposite.

Constructors and Inheritance – Base Class Initializers

- If the base class has a constructor, then that 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. WS, p 826-829

Employees have a name, and earn a salary.

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



```
#include <iostream>

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 << " paid "
                 << e.salary ); };};
```

```

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 << m <<" , at level"<<m.level);}; };

void main(){
    Manager Scrooge(5, "Scrooge MacDuck");
    Employee Donald(13456.5, "Donald Duck");

    cout << Donald << endl;
        // output    Donald Duck paid 13456.5
    cout << Scrooge << endl;
        // output    ScroogeMacDuck paid 50000.0
    (cout && Scrooge) << endl ;
        // output    Scrooge MacDuck paid 50000.0  at level 5
}

```

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

```
class Employee{
protected:
    float salary;    char* name;
public:
    Employee(float s){salary = s}; };

class Manager: public Employee{
private:
    int level;
public:
// Manager(char* n){ name=n; salary = 500000; level= 5;};
// Error: no appropriate default constructor available
};
```

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

So far, we only know half the truth about inheritance....

.... its real power and usefulness lies in

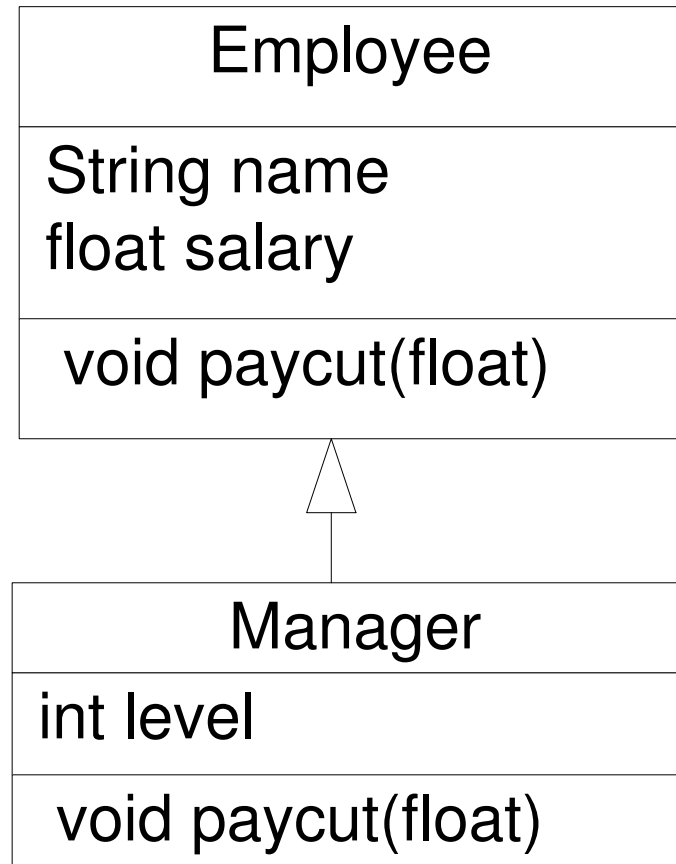
Virtual Functions

- Virtual Functions may have different implementations in the derived classes. The keyword **virtual** indicates that a function is virtual.
- When a virtual function is called for an object, the class of the object determines which function will be executed.
- Method binding is the process by which we determine which method to execute for a given call. In C++ we have static and dynamic method binding.
- The notation `Class::memFunction` ensures that the virtual mechanism is not used

For illustration we continue using the employees' example:

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

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



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

```

// in file EmployeesVirtualMain.cpp
#include <iostream>
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<< ", paid"<<e.salary<<endl); };

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

class Manager: public Employee{
private:
    int level;
public:

```

```

Manager(int l, char* n) : Employee(100000*l, n)
    {level=l;};

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

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

```

Note, that the function

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

is virtual, but the operators

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

and

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

are overloaded -- not virtual.

```

void main() {
    Manager Scrooge(5, "SMD");
    Employee Donald(13456.5, "Donald Duck");

    cout << Donald << endl;
        // output   Donal Duck, paid 13456.5
    Donald.paycut(300);
    cout << Donald << "\n " ;
        // output   Donal Duck, paid 13156.5
    cout << Scrooge ;
        // output   SMD, paid 500000.0, level 5
    Scrooge.paycut(300); cout << Scrooge ;
        // output   SMD, paid 501500.0, level 5
    Scrooge.Employee::paycut(300);
        // output   SMD, paid 501200.0, level 5
};

```

Virtual Functions, Static and Dynamic Binding

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

- non-virtual functions are bound statically
- virtual functions *may be* bound dynamically
- virtual functions are bound according to the class of the object executing the function
- when a virtual function is called for `pointer`, then the function is bound according to the *class of the object pointed to* by `pointer`, and *not* according to the *type* of `pointer`
- what about references?

In other words, we distinguish between static and dynamic binding for functions.

- *Dynamic binding*: function to be executed can only be determined at run-time
- *Static Binding*: function to be executed can be determined at compile-time

In C++

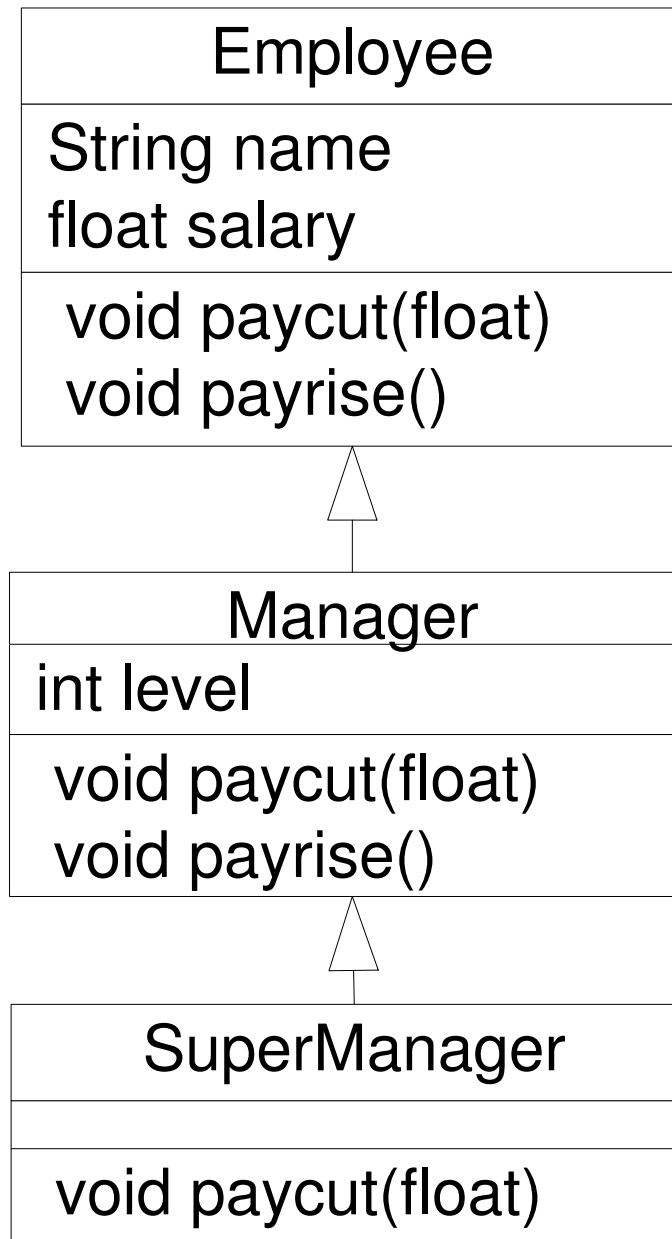
- Virtual functions are bound dynamically if the receiver is a pointer, ie `pointer->f(...)` is bound dynamically if `f` is virtual
- All other functions are bound statically

Language Design Philosophy: In other OO languages, eg Smalltalk, Java, there is only dynamic binding. Which mode is more important for OO? Why are there two modes of binding in C++?

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, and `Employee::payrise()` as a non-virtual function. The function `Manager::payrise()` **overrides** `Employee::payrise()`.

```
// file EmplsVirtsVirtsPntrsMain.cpp
#include <iostream>
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 << endl ); };
    virtual void paycut(float amount)
    {salary-=amount;};
```

```

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

```

```

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 << endl);};
}

```

/ The notation (typ) expr is a type cast; it requires the compiler to consider expr as of type typ. Type casts are not checked, and therefore they are dangerous. */*

```

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

```

```

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

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

void main(){
    Manager* M1= new Manager(5, "ScrMcDuck");
    Employee* E1= new Employee(13456.5, "DonDuck");
    Employee* E2= new SuperManager("WaltDisn");

    cout << "E1 : " << *E1 ;
        // output DonDuck earns 13456.5
    E1->paycut(300);
    cout << "E1: " << *E1;
        // output DonDuck earns 13156.5
}

```

```
E1->payrise();
cout << "E1: " << *E1;
    // output   DonDuck earns 13956.5

cout << "M1 : " << *M1 ;
    // output   ScrMcDuck earns 50000 at level 5
M1->paycut(300);
cout << "M1: " << *M1;
    // output   ScrMcDuck earns 51500 at level 5
M1->payrise()
cout << "M1: " << *M1;
    // output   ScrMcDuck earns 51600 at level 5

cout << "E2 :  " << *E2;
    // output   WaltDisn earns 100000
E2->paycut(300);
cout << "E2: " << *E2 << endl ;
    // output   WaltDisn earns 200000
E2->payrise()
```

```

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

E2 = E1;
cout << "E2 : " << *E2;
    // output  DonDuck earns 13956.5
E2->paycut(300);
cout << "E2: " << *E2 << endl ;
    // output  DonDuck earns 13656.5
E2->payrise()
cout << "E2: " << *E2 << endl ;
    // output  DonDuck earns 14456.5

// demonstrate static binding for objects
Employee Donald(30000.0, "Donald Duck");
SuperManager Walter("Walter Disney");

cout << "Donald: " << Donald;
    // output  Donald: Donald Duck earns 30000
Donald.paycut(300);

```

```

cout << "Donald: " << Donald;
    // output Donald: Donald Duck earns 29700
Donald.payrise();
cout << "Donald: " << Donald;
    // output Donald: Donald Duck earns 30500

cout << "Walter: " << Walter;
    // output Walter: Walter Disney earns 100000
    //          at level 10
Donald = Walter;
cout << "Donald: " << Donald;
    // output Donald: Walter Disney earns 100000
Donald.paycut(300);
cout << "Donald: " << Donald;
    // output Donald: Walter Disney earns 99700
Donald.paysrise();
cout << "Donald: " << Donald;
    // output Donald: Walter Disney earns 100500
};

```

The above example was about dynamic binding and static binding.

An example of *dynamic binding* is

- `E2->paycut(300)`, which results into 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`.

Examples of *static binding* are

- `Donald.paycut(300)`, which always results into calling `Employee::paycut(float)` even after the assignment `Donald = Walter`

- `E2->payrise()`, which always results into calling `Employee::payrise()` even if `E2` points to an object of class `Manager` or class `SuperManager`.
- `cout << *E2` which always results into calling the `ostream& operator <<(ostream&, Employee&)` even if `E2` points to an object of class `Manager` or class `SuperManager`

The difference between virtual functions and non-virtual overriding functions (eg `Employee::paycut(int)` and `Employee::payrise()`) is subtle. I suggest, that

- one should be aware of the difference,
- for our course, functions with different behaviours in subclasses should be declared virtual.

The above example demonstrates.

- life is not fair!
- it is possible and useful to have subclasses without additional data members (eg `SuperManager`).
- 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 , the class of object unknown at compile time, therefore dynamic binding, if the function is virtual.

Language Design Philosophy

Static binding results in faster programs. Dynamic binding allows for flexibility at run-time.

Programmers should use dynamic binding only when necessary - but *not* hard-code it!

In C++

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

Sound type systems guarantee that exception “object does not understand message” never occurs, that at run-time all variables contain objects of the class, or subclass of their definition. Soundness is a nontrivial, desirable property.

Dynamic Binding for local Function Calls

Remember that:

- virtual functions are bound dynamically for pointers
- for a member function f , inside that/another member function the call $f(\dots)$ corresponds to **this** $\rightarrow f$

and consider:

```
/* in file LocalFunctionsMain.cpp */  
#include <iostream>  
class Human{  
public:  
    void holiday()  
        { cout << "spends holidays ";  
          enjoying(); };  
    virtual void enjoying()  
        { cout << " watching TV\n";}; };
```

```
class Female: public Human{
public:
    virtual void enjoying()
        { cout << " cooking\n";};
};

void main()
    Human John;
    Female Gloria;

    cout << "John " ; John.holiday();
        // output John spends holidays watching TV
    cout << "Gloria " ; Gloria.holiday();
        // output Gloria spends holidays cooking
}
```

The previous example demonstrates

- Women are cleverer than men.
- 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.

For example, consider:

When a fox is woken up, it digests one unit of its undigested food. Afterwards, if it is not hungry it does nothing; if it is hungry and the next position is not occupied by another animal, it moves to the next position; if it is hungry and the next position is occupied by another animal, it attacks this animal.

When a chicken is woken up, it digests one unit of its undigested food. Afterwards, if it is not hungry it does nothing; if it is hungry and the

next position is not occupied by another animal, it moves to the next position; if it is hungry and the next position is occupied by another animal, it retreats.

The above should be expressed as

Such an approach

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

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.

```
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 virtual, even though only the function

Food::print() contains the keyword **virtual** in its declaration.

The keyword **virtual** in the declaration of the overriding functions (here FastFood::print() and Pizza::print()) is legal but superfluous.

```
void main() {
    Food* f; f = new Food; f->print();
// OUTPUT                                food
    f = new FastFood; f->print();
// OUTPUT                                fast food
    f = new Pizza; f->print();
// OUTPUT                                salami, pepperoni

    FastFood* ff; ff = new FastFood; ff->print();
// OUTPUT                                fast food
    ff = new Pizza; ff->print(); }
// OUTPUT                                salami, pepperoni
```

Clarification on Constructors, Destructors and Inheritance

Objects are constructed from the bottom up: first the base, then the members, and then the derived class. They are destroyed in the opposite order: first the derived class, then the members, and then the base.

For 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.

```
#include <iostream>

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:
    void turnOn(){ cout << "    turn_PC_on() \n"; };
    void turnOff(){ cout << "    turn_PC_off() \n";};
    PC(){ cout << " PC::PC() \n"; };
    ~PC(){ cout << " PC::~~PC() \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"; };};

void main(){
    Office* pOff = new Office();
    /* output
       Office::Office ()
    Boss* pBoss = new Boss(pOff);
    /* output
       Desk::Desk ()
       Empl::Empl ()
       PC::PC ()
       turn_PC_on ()
       Boss::Boss ()
    */
}

```

```

    delete pBoss;
/* output
    turn_PC_off()
    Boss::~Boss()
    PC::~PC()
    Empl::~Empl()
    Desk::~Desk()
*/

```

Notice: the destructor for employees does not automatically destroy the office.

Destructors are bound according to the same rules as any other functions.

```

    Empl* pEmpl = new Boss(pOff);
/* output
    Desk::Desk()
    Empl::Empl()

```

```

        PC::PC()
        turn_PC_on()
        Boss::Boss()
delete pEmpl;
/* output
        Empl::~~Empl()
        Desk::~~Desk()
return; }
*/
*/

```

There are, however,

Virtual destructors:

```

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

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

```

```

void main( ) {
    Emp1V* pEmp1V = new Emp1V();
    delete pEmp1V;
    /* output
        Emp1V::~~Emp1V()
    pEmp1V = new BossV();
    delete pEmp1V;
    /* output
        BossV::~~BossV()
        Emp1V::~~Emp1V()
    return; }

```

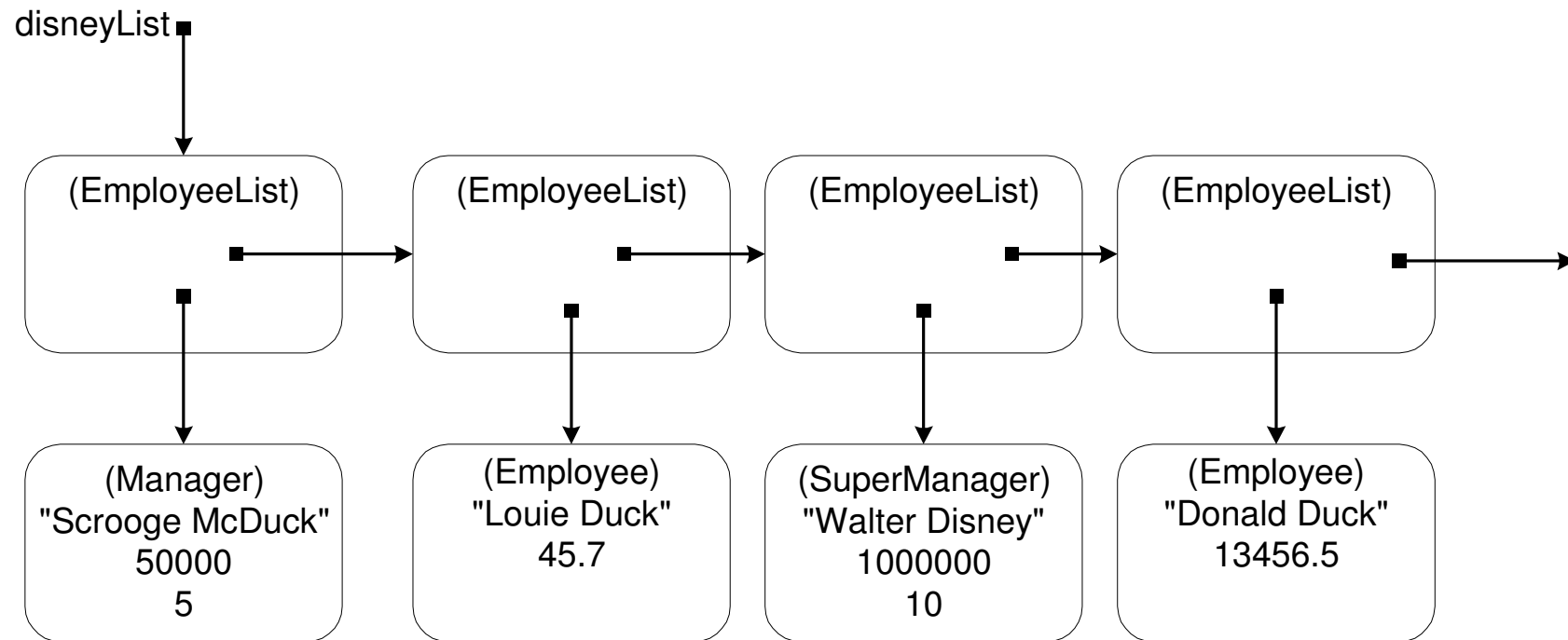
Destructors should be virtual if

There are *no* virtual constructors (what could they mean?) . .

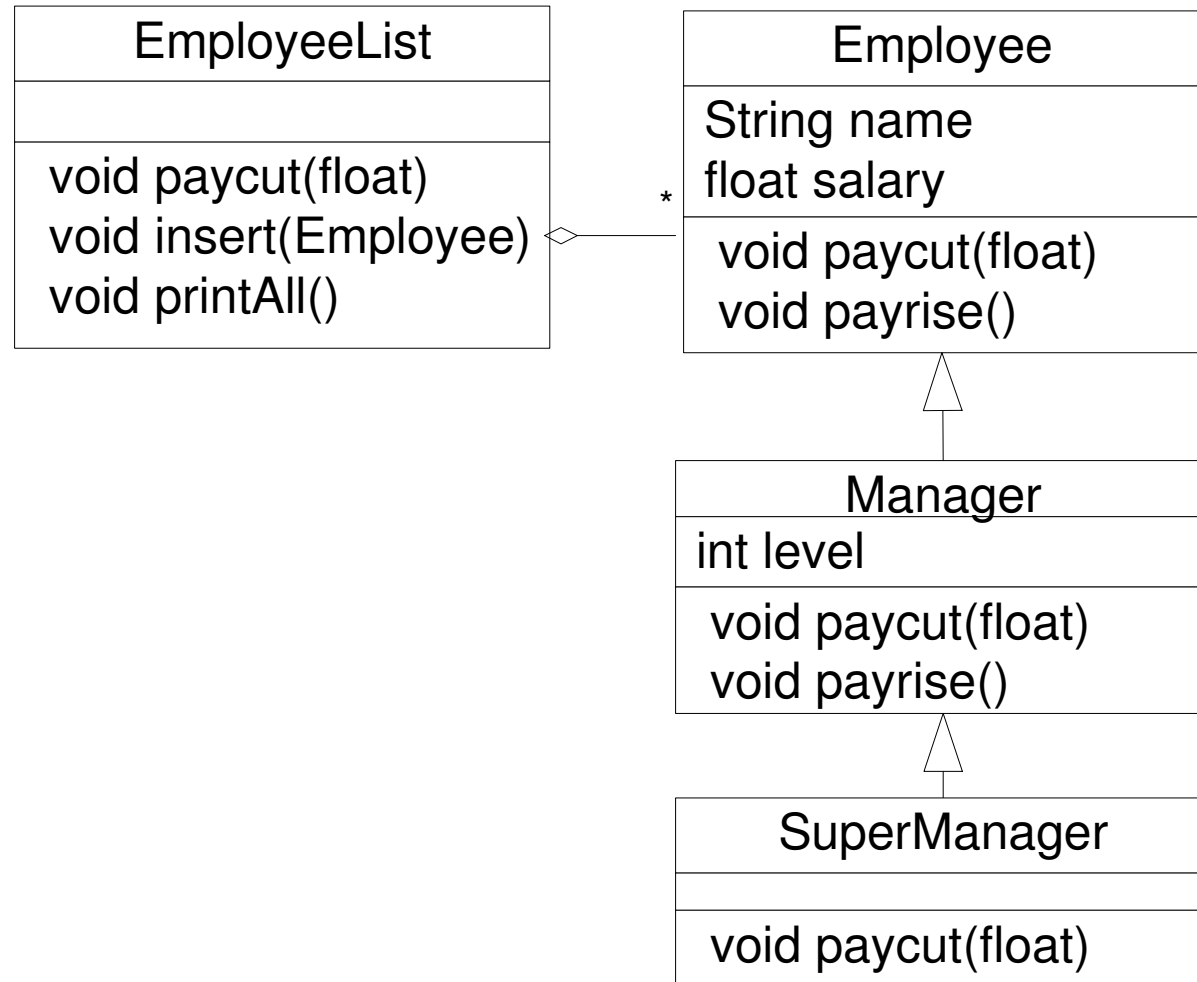
Polymorphism and Dynamic Binding

Polymorphism allows us to define and manipulate in a uniform way 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:



The
corresponding
UML class
diagram



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

```
class EmployeeList {
```

```

    EmployeeList* next;
    Employee* theEmployee;
public:
    EmployeeList(Employee* e)
        {theEmployee=e; next=nil;};
void insert(Employee* e)
    { EmployeeList* oldNext=next;
      EmployeeList* newList=new EmployeeList(e);
      newList ->next = oldNext;
      next = newList; };
friend ostream& operator<<
    (ostream& o, const EmployeeList& l)
    { o << "    " << *(l.theEmployee) ;
      return (l.next==nil) ? o : o << *(l.next);};
void paycut(float a)
    { theEmployee->paycut(a);
      if (next!=nil) next->paycut(a); };};

void 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") );
cout << disneyList;
/* output
    Scrooge Mac Duck with salary 50000
    Louie Duck with salary 45.7
    Walter Disney with salary 1000000
    Donald Duck with salary 13456.5          */

disneyList.paycut(40); cout << disneyList; }
/* output
    Scrooge Mac Duck with salary 50200
    Louie Duck with salary 5.7
    Walter Disney with salary 2000000

```

```
Donald Duck with salary 13416.5    */
```

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

Note, that if, after developing `EmployeeList`, we added further subclasses to `Employee`, the code for the class `EmployeeList` will still be valid!

Note also, that polymorphism in functional programming is achieved in a different way, and addresses a related, but different problem. The functional style polymorphism is similar to (and more powerful than) C++ templates.

Pure Virtual Functions, Abstract Classes

A virtual function may have no implementation. It only serves to define an interface provided by all objects of classes derived from its class. It is then *a pure virtual* function, denoted by the function body `=0`

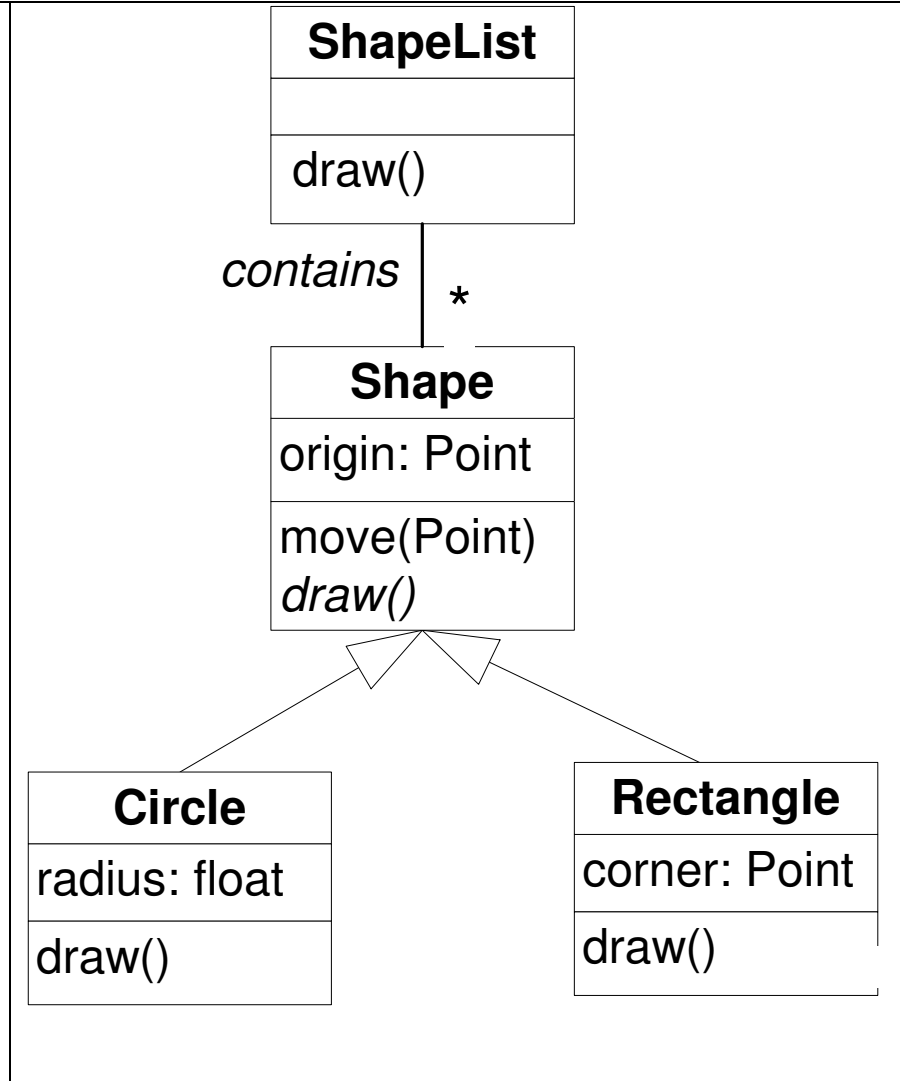
A class that contains at least one pure virtual function is an *abstract class*. No objects of that class may be created. (Why?)

Example of Use of Pure Virtual Functions

```
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() {...};
}
```

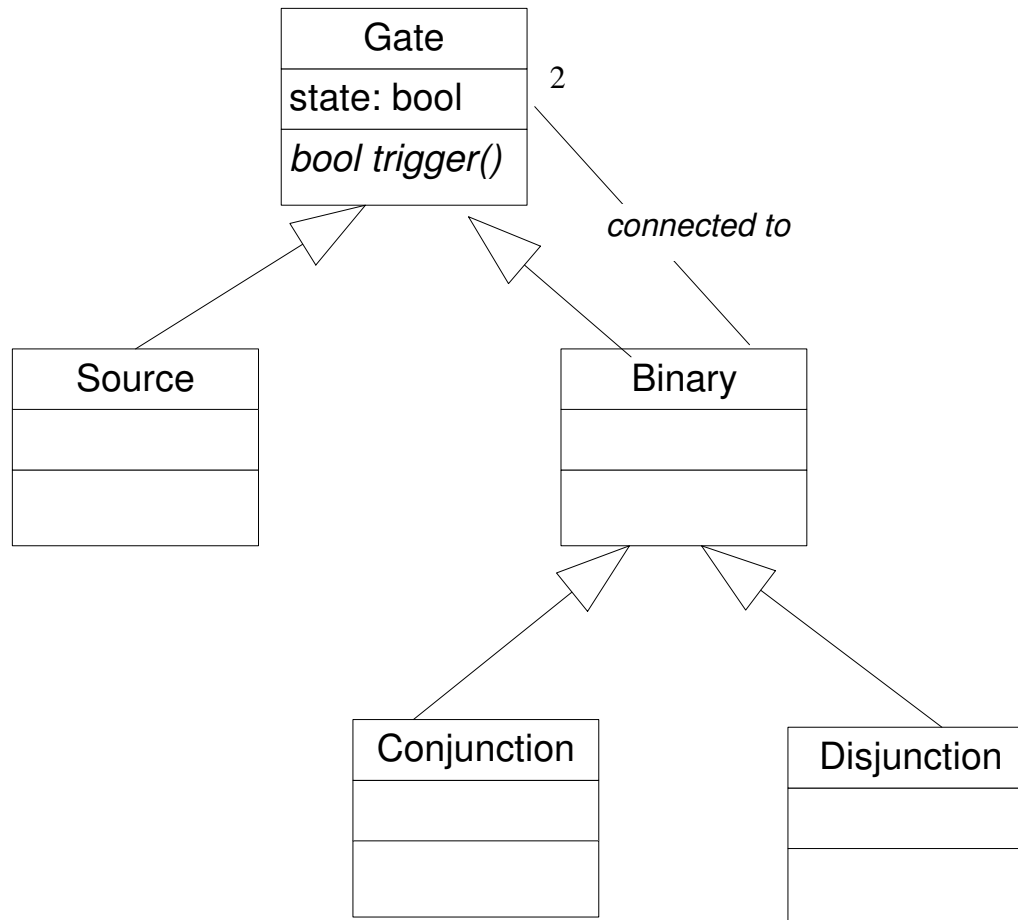


The pure virtual function `draw()` is indispensable!

Use of Abstract Classes

- The clients of the abstract class may safely expect all the objects of sub classes 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`.

Example: electronic gates



```

class Gate{
    ... . .
    bool state;
    virtual bool calculateState()=0; // pure virtual
    virtual bool trigger()
    { . . .waitFor(3);
      state = calculateState(); // using the
                                // pure virtual funct.
      return state; };
    Gate(char*) { ... }
};

```

```

class Source: public Gate{
    . . .
    bool calculateState(){ return state; };
    Source(char* n):Gate(n){};
};

```

```

class Binary: public Gate{
    . . .

```

```

virtual bool calculate(bool,bool)=0;
bool calculateState()
    { state =
        calculate( pin1->state, pin2->state );
      return state; };
Binary(char* n):Gate(n){pin1=null; pin2=null;};
};

```

```

class Conjunction: public Binary{
    . . .
    bool calculate(bool b1, bool b2)
        { return b1&&b2; };
    Conjunction(char* n):Binary(n); };

```

```

class Disjunction: public Binary{
    . . .
    bool calculate(bool b1, bool b2)
        { return b1 || b2; };
    Disjunction(char* n):Binary(n);
};

```

```
void main() {  
  
    // Gate G("G");  
    // 'Gate' : cannot instantiate abstract class ...  
  
    Gate *G1, *G2;  
    // G1 = new Gate("G1");  
    // 'Gate' : cannot instantiate abstract class ...  
    G1 = new Source("G1");  
    G1 = new Conjunction("G1");  
    // G1 = new Binary("G1");  
    // 'Binary' : cannot instantiate abstract class ..  
  
    G1->trigger(); } // call pure virtual function
```

Clarification on Overloading and Inheritance

- overloading: several function declarations for a single name; for a function call appropriate function body is selected by comparing the type of actual arguments with the type of formal parameters
- inheritance: if a base class B contains a virtual function f and a derived class D contains a function f with the same type as $B::f$, then a call of $f(\dots)$ for an object of class D invokes $D::f$

Types are known at compile time; classes are known at run time. Therefore:

- overloading is resolved statically according to the compile time type of the arguments.
- inheritance is resolved dynamically according to the run time class of the receiver.

Overloading is resolved statically.

In the following example, females are humans. Wolves love eating humans but hate eating females.

```
#include <iostream>
class Human{};

class Female: public Human{};

class Wolf{
public:
    void eats(Human h)
        { cout << " yummy!! \n ";};
    void eats(Female f)
        { cout << " yacc!! \n ";};
};

void main() {
    Human John, *aPerson; Female Julia;
```

```

Wolf aWolf;

aWolf.eats(John);
// OUTPUT          yummy!!
aWolf.eats(Julia);
// OUTPUT          yacc!!
aPerson = &John;
aWolf.eats(*aPerson);
// OUTPUT          yummy!!
aPerson = &Julia;
aWolf.eats(*aPerson);
// OUTPUT          yummy!!
}

```

The above did not involve dynamic binding. However, functions may be involved in both overloading and inheritance:

```

#include <iostream>    class Female;

class Human{
public:
    virtual void chatWith (Human* h)
        { cout << " of sports";};
    virtual void chatWith (Female* f)
        { cout << " of disco";};
};

class Female: public Human{
public:
    virtual void chatWith(Human* h)
        { cout << " about IC\n ";};
    virtual void chatWith(Female*)
        { cout << " about politics\n ";};
};

```

In the above example:

- `Human::chatWith(Human*)` **overloads**
`Human::chatWith(Female*)`
- `Female::chatWith(Human*)` **overloads**
`Female::chatWith(Female*)`
- `Female::chatWith(Human*)` **redefines**
`Human::chatWith(Human*)` **(virtual functions)**
- `Female::chatWith(Female*)` **redefines**
`Human::chatWith(Female*)` **(virtual functions)**

```

void main() {
    Human John, *Paul, *Han;   Female Julia, *Paola;

    John.chatWith( Paul );    // output      of sports
    John.chatWith( Paola );  // output      of disco
    Julia.chatWith( Paul);    // output      about IC
    Julia.chatWith( Paola);  // output      about politics
    // notice that the pointers were uninitialized
    // why did this work?

    Han = new Human;
    Han->chatWith( Han );     // output      of sports
    Han->chatWith( Paola );   // output      of disco

    Han = new Female;
    Han->chatWith( Han );     // output      about IC
    Han->chatWith( Paola );   // output      about politics
};

```

Member Access Control

Members may be:

- *public* may be used anywhere.
- *protected* may be used by member functions and friends of the class, by member functions and by friends of classes derived from the class in which declared.
- *private* may only be accessed by member functions of the class or by friends of the class.

```

class A {
public:
    char publica;
protected:
    char protecteda;
private:
    char privatea; };

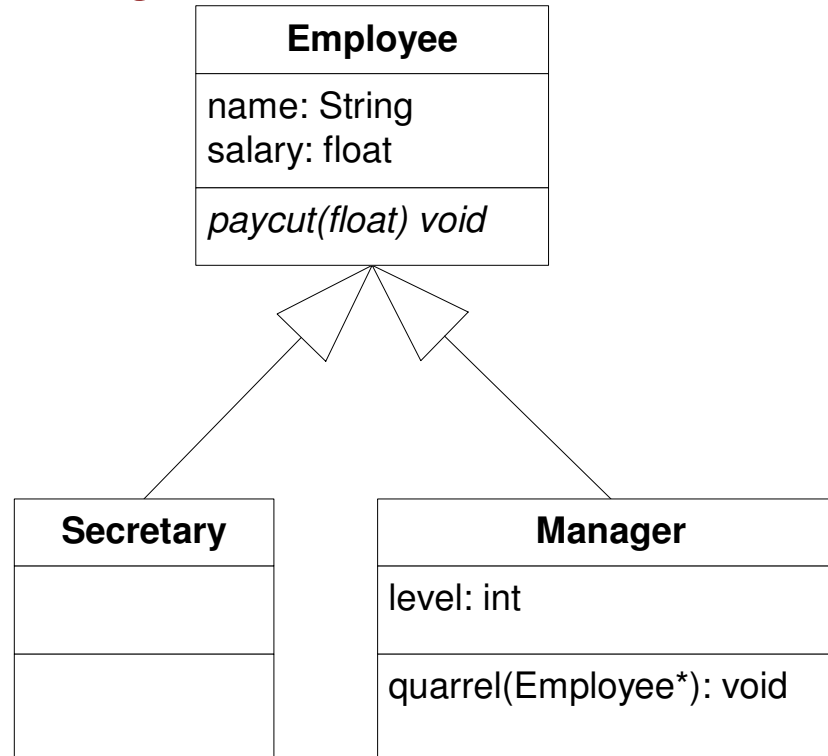
class B: public A{
    void test()
    {   publica = protecteda;
        /* publica = privatea; */   }; };

int main()
{   A anA; B aB;
    anA.publica = 'a';
    // anA.protecteda = 'a';
    // anA.privatea = 'a';
};

```

Member Access Control – a practical example

Consider the following, familiar situation



where the `name` is private, and the `salary` is protected

We obtain the following header files

```

/* in file Employee.h */    #include <iostream>
    class Employee{
    private:
        char* name;          // subclasses are not
                              // concerned with names

    protected:
        float salary;       // subclasses may access
                              // the salary

    public:
        Employee(float, char*);
        friend ostream&
            operator<<(ostream&, const Employee&);
        virtual void paycut(float amount)=0; };

/* in file Secretary.h */  #include "Employee.h"
    class Secretary: public Employee{
    public:
        Secretary(float, char*);
        void paycut(float); };

```

```

/* in file Manager.h */ #include "Employee.h"
class Manager: public Employee{
private:
    int level;
public:
    Manager(int, char*);
    void paycut(float);
    void quarrel(Employee*);
};

```

the bodies may be as follows:

```

/* in file Employee.cpp */ #include "Employee.h"
Employee::Employee(float s, char* n)
    {salary = s; name=n;};

```

```

ostream& operator<<(ostream& o, const Employee& e)
    {return (o<<e.name<< ", paid "<<e.salary<<endl);};

```

```

/* in file Secretary.cpp */ #include "Secretary.h"

Secretary::Secretary(float s, char* n) : Employee(s, n)
{ };

void Secretary::paycut(float amount)
{ // cout << *name << " is angry\n";
  // 'name' : cannot access private member ...
  cout << *this << " is angry\n";
  salary -= amount ;};

```

```

/* in file Manager.cpp */ #include "Manager.h"

Manager::Manager(int l, char* n)
    : Employee(1000.0*l, n) {level=l;};

void Manager::paycut(float amount)
{ salary += (level*amount);};

```

```

void Manager::quarrel(Employee* e)
{
    salary+=100;
    e->paycut(300);
    // salary = 300+ e->salary;
    // 'salary' : cannot access protected member
    // declared in class 'Employee'
    //
    //             note difference to private ...
};

```

The main function may be as follows:

```

/* in file Main.cpp */ #include "Manager.h"

void main() {
    Manager M(3, "Hilary");
    cout << M << endl;
        // OUTPUT           Hilary, paid 3000
    // M.salary = 300;
}

```

```
// 'salary' : cannot access protected member

Employee *E1 = new Secretary(356.5, "Bill");
cout << *E1 << endl;
    // OUTPUT    Bill, paid 365.5
M.quarrel(E1);
    // output    Bill, paid 365.5, is angry
cout << M << endl;
    // OUTPUT    Hilary, paid 3100
cout << *E1 << endl;
    // OUTPUT    Bill, paid 56.5

};
```

Access Specifiers for Base Classes

The base class of a derived class may be specified to be **public**, **protected** or **private**. The access specifier affects the extent to which the derived class may inherit from the derived class, 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*`

```
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 Stack : private List{ ..... }
```

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

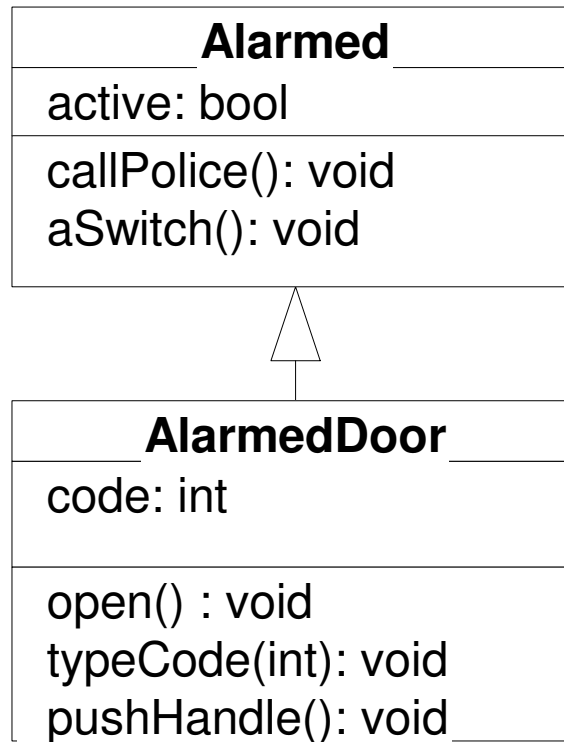
The interplay of access modifiers is quite sophisticated. For our course, we concentrate on the use of access modifiers for class members, distinguish private and public derivation, but do not worry about the distinction between private and protected derivation.

Private vs Public Derivation – a practical example

`Alarmed` entities are active or inactive, and have the ability to call the police. They switch from active to inactive, and from inactive to active through the function call `aSwitch()`.

`AlarmedDoors` have a code which controls the door: when typing the correct code one can deactivate/activate the door. If one pushes the handle (`pushHandle()`) and the door is inactive, then the door opens (`open()`), otherwise the police is called.

`AlarmedDoors` are implemented in terms of `Alarmed`, but may not be access as `Alarmed` entities.



```

/* file Alarmed.h */ #include <iostream>
class Alarmed{
private:
    bool active;
public:
    Alarmed();           // constructs active alarm
    bool isActive();
  
```

```

void aSwitch();
        // if alarm active, deactivate
        // if alarm inactive, activate
void callPolice();
        // notify the police
friend void repair(Alarmed* a);
        // notice, NOT a member
        // repairs a, and deactivates it
};

/* file AlarmedDoor.h */ #include "Alarmed.h"
// Note the use of private inheritance
class AlarmedDoor: private Alarmed{
    const int code;        // code for particular door
                          // you may enter only if you
                          // use the correct code

    void open();          // opens the door
public:
    AlarmedDoor(int);

```

```

void typeCode(int);
    // if argument matches code, then
    // if door is active, then deactivate
    // if door is inactive, then activate
void pushHandle();
    // if door active, then set alarm on
    // if door inactive, then open
};

```

```

/* file Alarmed.cpp */ #include "Alarmed.h"

```

```

Alarmed::Alarmed(){ active = true; };
void Alarmed::aSwitch( ){ active=!active; };
void Alarmed::callPolice(){ cout << "police, police!
\n"; };
bool Alarmed::isActive(){ return active; };

void repair(Alarmed* a){ a->active=false; }

```

```

/* file AlarmedDoor.cpp */ #include "AlarmedDoor.h"

```

```
void AlarmedDoor::open() { cout << "crrrh!\n"; };  
// door needs oiling
```

```
AlarmedDoor::AlarmedDoor(int n) : Alarmed(), code(n) {  
};
```

```
void AlarmedDoor::typeCode(int c)  
{ if (c==code) aSwitch();  
else cout << " nice try!\n"; }
```

```
void AlarmedDoor::pushHandle() {  
if (isActive()) callPolice(); else open(); }
```

Private inheritance ensures that the clients of `AlarmedDoor` cannot jeopardize its security:

```
/* file Main.cpp */
void main()
{   AlarmedDoor d(4455);

    d.typeCode(4455);    // deactivate
    d.pushHandle();
        // OUTPUT   crrrh!
    d.typeCode(4455);    // activate

    d.pushHandle();
        // OUTPUT   police, police!
    d.typeCode(6677);
        // OUTPUT   nice try!
    d.pushHandle();
        // OUTPUT   police, police!
    // Now trying to fool the security
```

```
// d.aSwitch();  
// 'aSwitch' : cannot access ....  
  
// repair(&d);  
// type cast from AlarmedDoor* to Door* exists,  
// but it is inaccessible
```

However, public inheritance would have exposed the whole security mechanism and broken the security. Consider, namely:

```
class BadAlarmedDoor: public Alarmed{  
    const int code;  
    void open();  
public:  
    BadAlarmedDoor(int);  
    void typeCode(int);  
    void pushHandle();  
  
};
```

where the bodies of `BadAlarmedDoor` are identical to those of `AlarmedDoor`, and in `main`

```
void main()
{   BadAlarmedDoor bd(3366);

    cout << " \n \n";
    bd.typeCode(3366);    // deactivate alarm
    bd.pushHandle();
        // OUTPUT   crrrh!
    bd.typeCode(3366);    // activate alarm

    bd.pushHandle();
        // OUTPUT   police, police!
    bd.typeCode(6677);
        // OUTPUT   nice try!
    bd.pushHandle();
        // OUTPUT   police, police!
    // So far, bd acted like an AlarmedDoor,
    // but we will now
```

```
// successfully fool the excurity
bd.aSwitch();
bd.pushHandle();
    // OUTPUT    crrrh!

bd.typeCode(3366);    // activate alarm
repair(&bd);
bd.pushHandle();
    // OUTPUT    crrrh!
```

Instead of private inheritance we could have used

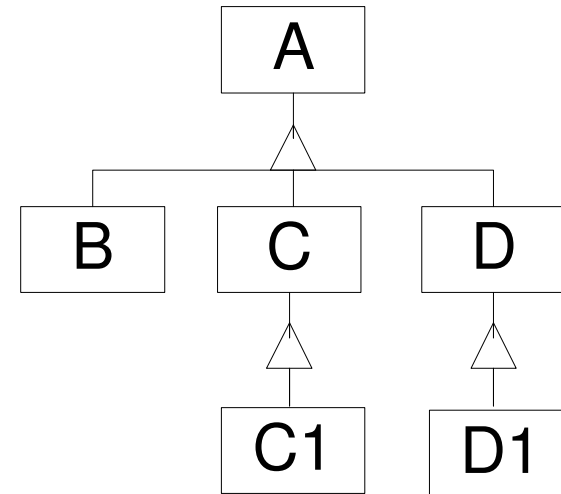
This solutions would have the drawback that

The following, contrived example demonstrates the interplay of access modifiers for members and for inheritance, and can serve as a reference. We do not need to study it in detail.

```
class A {  
    public:  
        char publica;  
    protected:  
        char protecteda;  
    private:  
        char privatea; };
```

```
class B: public A{  
    private:  
        void test()  
        { publica = protecteda;  
          /* publica= privatea; */};};
```

```
class C: protected A{
```



```
void test()  
{ publica = protecteda;  
  /* publica= privatea; */ };
```

```
class C1: public C{  
  void test()  
  { publica = protecteda;  
    /* publica= privatea; */};};
```

```
class D: private A{  
  void test()  
  { publica=protecteda;  
    /* publica=privatea; */};};
```

```
class D1: public D{  
  void test()  
  { /* publica=protecteda;  
    publica=privatea; */};};
```

```
int main()
{   A anA; B aB; C aC; D aD;
    anA = aB; aB.publica = 'a';

    // aB.protecteda = 'a';
    // aB.privatea = 'a';

    // anA = aC;

    // aC.publica = 'a';
    // aC.protecteda = 'a';
    // aC.privatea = 'a';

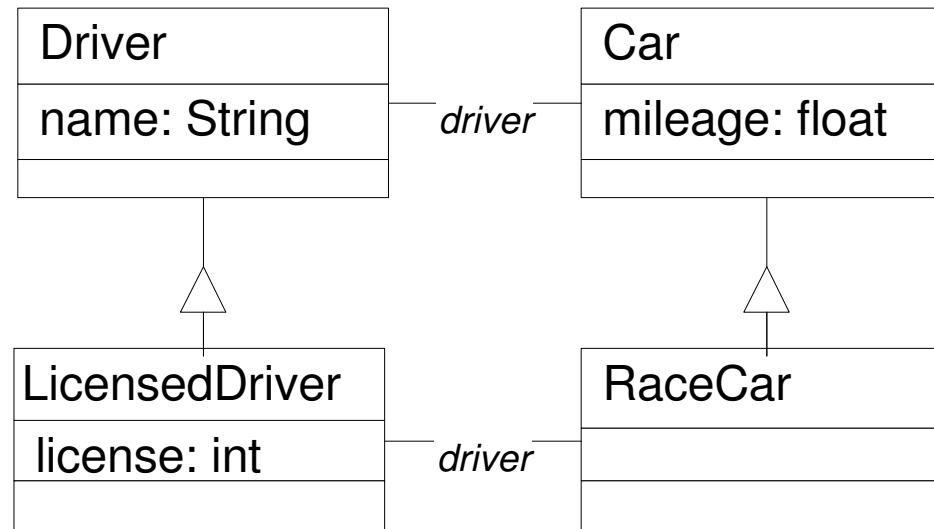
    // anA = aD;

    // aD.publica = 'a';
    // aD.protecteda = 'a';
    // aD.privatea = 'a';

    return; };
```

Virtual Members - more

Consider the situation where cars are driven by drivers, whereas race cars are driven by licensed drivers:



Could this be expressed in C++?

The answer is NO, because there are no virtual data members!

```

/*file CarsDriversMain.cpp - Inheritance&Types */
#include <iostream>
class Driver {
public:
    char* name;
    Driver(char* aName=0) {name=aName;};
    virtual void print()
        {cout << "driver called " << name << "\n";};};

Driver* defaultDriver= new Driver("Mickey Mouse");

class LicensedDriver: public Driver{
public:
    int license;
    LicensedDriver(char* aName, int licenseNr)
        {name=aName; license=licenseNr; };
    void print()
        {cout << "licensed driver called " << name <<
            ", license number " << license << "\n";}; };

```

```

class Car{
public:
    Driver* theDriver;
    int mileage;
    Car(Driver* aDriver=defaultDriver, int aMileage=0)
        {theDriver=aDriver; mileage=aMileage;};
    void setDriver(Driver* newDriver)
        {theDriver=newDriver; };
    virtual void print()
        {cout << "- a car with mileage " << mileage <<
            ", and " ; theDriver->print();}; };

```

```

class RaceCar: public Car{
public:
    LicensedDriver* theDriver;
    RaceCar(LicensedDriver* aDriver, int aMileage)
        {theDriver=aDriver; mileage=aMileage;};
    void setDriver(LicensedDriver* newDriver)
        {theDriver=newDriver; };

```

```

virtual void print()
    {cout << "- a racing car with mileage " <<
      mileage << ", and " ; theDriver->print();};};

```

Since there are no virtual data members, a RaceCar will have the following data members:

```

    Driver* theDriver;           // declared in Car
    int mileage;                 // declared in Car
    LicensedDriver* theDriver; // declared in RaceCar

```

```

void main(void)
{
    Driver* Noddy=new Driver("Noddy");
    Car* NoddysCar= new Car(Noddy,4000000);
    Noddy->print();
    // OUTPUT      driver called Noddy

    LicensedDriver* JamesBond =
        new LicensedDriver("James Bond",7);

```

```

JamesBond->print();
    // OUTPUT      licenced driver called James Bond,
    //              license number 7

Car* blueBird = new RaceCar(JamesBond,20);
blueBird -> print();
    // OUTPUT      a racing car with mileage 20, and
    //              licenced driver called James Bond,
    //              license number 7

blueBird->Car::print();
    // OUTPUT      a car with mileage 20, and
    //              driver called Mickey Mouse

blueBird->setDriver(Noddy);
blueBird -> print();
    // OUTPUT      a racing car with mileage 20, and
    //              licenced driver called James Bond,
    //              license number 7

blueBird -> Car::print();
    // OUTPUT      a car with mileage 20, and
    //              driver called Noddy

```

The above example demonstrates:

- stay at home when Noddy is driving the blueBird
- one cannot redefine data members
- but one might have used instead.
- one cannot override data members for reasons of type soundness, Namely, assume one could override data members, and consider:

```
Car* anyCar;  
RaceCar *bluebird  
blueBird = new RaceCar (JamesBond, 20) ;  
anyCar=bluebird;  
anyCar->setDriver (Noddy) ;  
blueBird->theDriver->license  
    // OUTPUT object does not understand message
```

Language Design Philosophy

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

- as much static binding as possible
- dynamic binding only when necessary
- operate on a “need to know” basis
- type system is (probably) sound.

Sound type systems guarantee that at run time

- exception “object does not understand message” never occurs,
- variables contain objects of class, or subclass of their definition.

Soundness is a desirable, nontrivial property. Eiffel was demonstrated to be unsound 4 years after its launch. Most of Java was demonstrated sound (by others and yours truly). C++ type soundness is a challenging open task. *Projects!*

C++ The Object Oriented Features - Summary

- derived classes inherit characteristics from base class
- derived class objects may appear wherever base class objects expected
- derived classes may override virtual functions
- virtual function bound according to class of receiver
- virtual functions bound dynamically for pointers and for `this` ->
- polymorphic structures may be programmed using pointers
- pure virtual functions may declare but not define function
- abstract classes provide interface for partially defined classes
- member access control supports encapsulation
- base class access specifiers distinguish “specification inheritance” from “implementation inheritance”

C++ Good Style

-
.....
-
.....
-
.....
- different objects to reflect different logical entities
- let the object do the work *it* is responsible for
- encapsulation
- distinguish "has a", "is a", "behaves as a"
- make the best of code reuse

Make the best of code reuse – example 1

```
// instead of
```

```
class B {  
public:  
    int a1;  
    int a2[20];  
    char* b;  
};
```

```
class C {  
public:  
    int a1;  
    int a2[20];  
    char* b;  
};
```

```
// DO
```

Make the best of code reuse – example 2

```
// instead of
```

```
class B {  
public:  
    void f(void) { /* do x, do y1, do z */}  
};
```

```
class C {  
public:  
    void f(void) { /* do x, do y2, do z */}  
};
```

```
// DO
```

Make the best of code reuse – example 3

```
// instead of
```

```
class B {  
private:  
    int isSomething;  
public:  
    void f(void)  
        { /* if (isSomething)  
                {caseSomething}  
            else  
                {caseNotSomething} */  
};  
  
// DO
```