

## Chapter 9

### The C++ Programming Language

#### Object Oriented Programming (inheritance)

### The Personnel Example

- Suppose we want to computerize our personnel records...
- We start by identifying the two main types of employees we have:

```

struct Engineer {
    Engineer *next;
    char *name;
    short year_born;
    short department;
    int salary;
    char *degrees;
    void give_raise(
        int how_much
    );
    // ...
};

```

The magic of MS

```

struct SalesPerson {
    SalesPerson *next;
    char *name;
    short year_born;
    short department;
    int salary;
    double *comission_rate;
    void give_raise(
        int how_much
    );
    // ...
};

```

263

### Identifying the Common Part

```
class Employee {
  char *name;
  short year_born;
  short department;
  int salary;
  Employee *next;
  void give_raise(
    int how_much
  );
  // ...
};
```

C version:

```
struct Engineer {
  struct Employee E;
  char *degree;
  /* ... */
};
```

Indeed, inclusion is a poor man's (poor) imitation of inheritance!

```
class Engineer: Employee {
  char *degrees;
  // ...
};
```

```
class SalesPerson: Employee {
  double *commission_rate;
  // ...
};
```

264

### Inheritance

- ❑ The **Behaviour** and the **Data** associated with a subclass are an **extension** of the properties of the superclass.
- ❑ **Engineer** and **SalesPerson** extend, each in its own way, the data and behaviour of **Employee**.
- ❑ Identifying the **Employee** abstraction, helps us define more types:

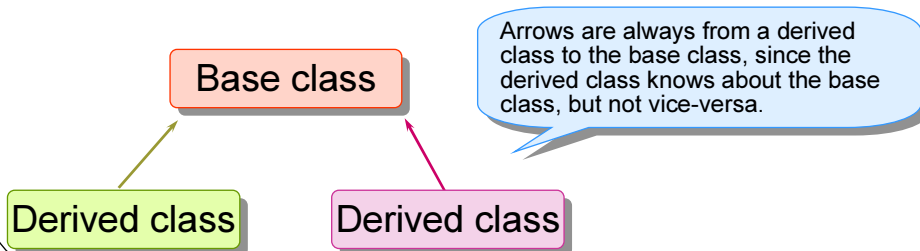
```
class Manager: Employee {
  char *degrees;
  // ...
};
```



265


## Inheritance Mechanism

- A **class** may *inherit* from a **base class**, in which case the inheriting class is called a **derived class**.
- A **class** may *override* inherited **methods** (member functions), but **not** inherited **data**.
- A *Hierarchy* of related classes, that share code and data, is created.
- **Inheritance** can be viewed as an **incremental refinement** of a base class by a new derived class.





266

## The Benefits of Inheritance

-  **Software Reusability.** Inheritance allows you to modify or extend a package somebody gave you without touching the package's code

Saves programmer time: Least important !

increase reliability, decrease maintenance cost, and, if code sharing occurs, smaller programs.
-  **Consistency of Interface.** An overridden function must have the same return type: More important !

Saves user time: Easier learning, and easier integration. guarantee that interface to similar objects is in fact similar.
-  **Polymorphism.** Different objects behave differently as a response to the same message. Most important !



267

## Inheritance Vector Example

Here is a simple version of a `Vector` class that implements an unchecked array of integers:

```
class Vector {
    int len, *buff;
public:
    Vector(int l): len(l), buff(new int[l]){}
    ~Vector()      { delete[] buff; }
    int size() const { return len; }
    int& operator[] (int i) { return buff[i]; }
};

int main()
{
    Vector v(10);
    // Populate the vector, and then use it.
    v[5] = v[6] + 5;
    // ..
}
```

268

## A Checked Vector

- We may also need a vector whose bounds are checked on every reference.
- This is done by defining a new derived class `CheckedVector` that inherits the characteristics of the base class `Vector`, and overrides some of its methods:

```
class CheckedVector: public Vector {
public:
    CheckedVector(int size): Vector(size) {}
    int& operator[] (int); // A primitive overriding
};

int& CheckedVector::operator[] (int i)
{
    if (0 > i || i >= size())
        error(); // Exceptions :-()
    return Vector::operator[] (i);
}
```



269

## Points Of Interest

- When a **class** inherits from a **class/struct**, the **public** keyword should be used, as in

```
class CheckedVector: public Vector {
...
}
```

(Don't ask why at this time.)

- Constructors are not Inherited.** If we haven't defined a constructor for `CheckedVector`, then the compiler would try to generate an empty default constructor and **fail** in doing so since there is no default constructor of the base class `Vector`.

- The construction (initialization) of the **sub-object** of the inherited class `Vector` must be done in the **initialization list** of the constructor of the derived class `CheckedVector`

```
...
CheckedVector(int size): Vector(size) {}
...
```

270

## More Points of Interest

- `CheckedVector` overrides the array reference function of `Vector`.

```
...
int& CheckedVector::operator[](int i)
....
```

- Since `CheckedVector` doesn't override the `size` function of `Vector`, the following calls the inherited function

```
...
if (0 > i || i >= size())
....
```

- An overridden function can be called as follows:

```
...
return Vector::operator[] (i);
....
```



271

## Data Hiding and Derived Classes

- **private** members of the base class are **not** accessible to the derived class
  - Otherwise, privacy is completely compromised by an inherited type.
- **public** members of the base class are accessible to anyone
- **protected** members of the base class are accessible to derived classes only

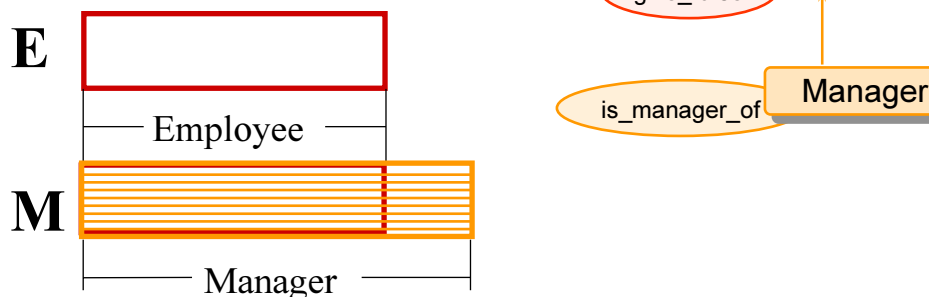
```
class Vector {
    protected:
        int len, *data;
    public:
        // ...
};
```

Protected data members make derived classes addicted

272

## Inheritance as Type Extension

```
Employee E;
Manager M;
```



This will first call the (compiler generated) type casting operator from Manager to Employee and then call the (compiler-defined or user-defined) Employee to Employee assignment operator.

```
E = M;    // OK
M = E;    // Error
```



273

## Is-A Relationship

A derived class **is a** (more specialized) version of the base class:

- Manager **is an** Employee
- CheckedVector **is a** Vector.
- LandVehicle **is a** Vehicle.

Thus, any function taking the class B as an argument, will also accept class D derived from B.

```
class Complex { ... };
Complex operator +(const Complex& c1, const
                  Complex& c2) { ... }

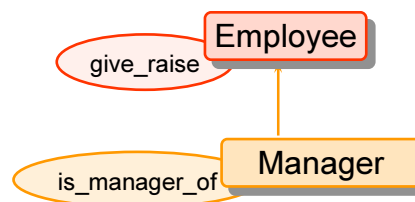
class BoundedComplex: public Complex {
    ...
} z1, z2;
Complex c = z1 + z2;
```

The last statement calls `operator+(Complex, Complex)` using type conversion.

274

## Calling Methods of Inherited Types

```
Employee E;
Manager M;
```



```
E.give_raise(10); // OK
M.give_raise(10); // OK
```

Again, **Manager** is an **Employee** but not vice-versa!

```
E.is_manager_of(...); // Error
M.is_manager_of(E); // OK
```



275

## Using Pointers to Inherited Types

A pointer to an **inherited type** is generated whenever an **inherited method** is called.

```
Employee E, *pE;
Manager M, *pM;
```

**Rules for pointer mixing:**

```
pM = &E; // Error
pE = &M; // OK
M = *pE // Error
M = *(Manager *)pE; // OK ... if you know what you are doing
M = *static_cast<Manager*>(pE); // MUCH safer
```

276

## Pointer Arrays

In many cases it is convenient to have a **mixed type collections**. The simplest and easiest way to do so is to use an array of pointers to the base type (the **superclass**).

```
Employee *Dept[100];
```

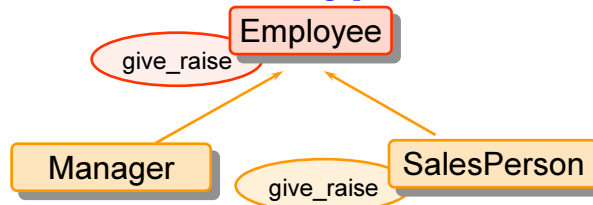
It is easy to deposit objects into the above array, however, determining what type of objects resides in each location is not so easy (and here starts REAL OOP).





## Calling Methods of Inherited Types (revisited) <sup>277</sup>

```
Employee E;
Manager M;
SalesPerson S;
```



```
E.give_raise(10); // Employee::give_raise();
M.give_raise(10); // Employee::give_raise();
S.give_raise(10); // SalesPerson::give_raise();
```

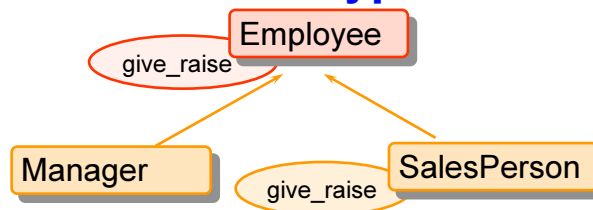
```
void SalesPerson::give_raise(int how_much)
{
    Employee::give_raise(how_much
        + commission_rate * total_sales);
}
```

## A Design Error



## Calling Methods of Inherited Types (Cont.) <sup>278</sup>

```
Employee E;
Manager M;
SalesPerson S;
```



```
Employee *Dept[len];
```

```
for (i=0; i<len; i++)
    Dept[i]->give_raise(10);
```

Since array elements are accessed via `Employee*`  
the `Employee::give_raise()` will be called **for all objects**

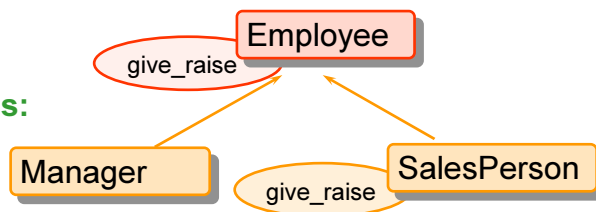
## The wrong way to override a method



## Determining the Object Type 279

Given a pointer of type `base*` (where `base` is a **base class**), how do we know the **actual class** of the object being pointed to ?

There are three solutions:



- Ensure that only objects of a **single type** are always pointed to.
- Place a **type field** in the base class and update it in the constructor of the derived class.
- Use **virtual functions**.

## Polymorphism 280

**Virtual functions** give full control over the **behavior of an object**, even if it is referenced via a **base class pointer** (or reference).

```

class Employee {
    ...
public:
    ...
    virtual void give_raise(int how_much) {...};
};
  
```

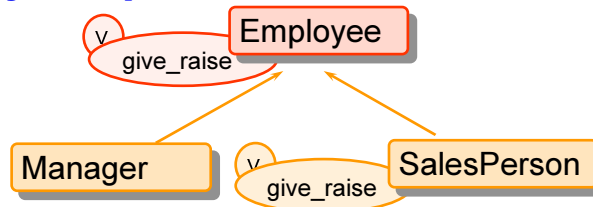
```

class SalesPerson: public Employee {
    ...
    double *commission_rate;
public:
    ...
    virtual void give_raise(int how_much) {...};
};
  
```

## Polymorphism

281

```
Employee E;
Manager M;
SalesPerson S;
```



```
Employee *Dept[len];
```

Although array elements are accessed via `Employee*` the appropriate `<class>::give_raise()` is always called !

```
for (i=0; i<len; i++)
  Dept[i]->give_raise(10);
```

Yet, `Dept[i]->is_manager_of(E)` is illegal

## Constructors and Destructors

282

**Constructors and Destructors cannot be inherited.**

They always call their base-class counterparts.

But while **Ctors are never virtual**, Dtors may be virtual.

**Any class** that has a virtual function, or even just **designed to have derived classes with virtual functions must define a virtual Dtor.**

```
class Employee {
public:
  Employee(char *n, int d);
  virtual ~Employee();
  ...
};
```

```
class Manager : public Employee {
public:
  ...
  Manager(char *n, int l, int d)
    : Employee(n,d),
      level l, group(0){}
  virtual ~Manager();
};
```

A Ctor that calls a virtual function, cannot call a derived-class' version of that function.

