1

Chapter 8

The C++ Programming Language

Inheritance

2

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 {
    char* name;
    short year_born;
    short department;
    int   salary;
    char* degrees;
    void give_raise(int how_much);
    …
};
```

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

3

Identifying the Common Part: Inclusion in C

```
struct Employee {
    char* name;
    short year_born;
    short department;
    int   salary;
    void give_raise(int how_much);
    …
};
```

```
struct Engineer {
    struct Employee E;
    char* degrees;
    …
};
```

```
struct SalesPerson{
    struct Employee E;
    double commission_rate;
    …
};
```

4

Identifying the Common Part: Inheritance in C++

base class

```
class Employee {
    char* name;
    short year_born;
    short department;
    int   salary;
    void give_raise(int);
    …
};
```

derived class

```
class Engineer: Employee {
    char* degrees;
    …
};
```

derived class

```
class SalesPerson: Employee {
    double commission_rate;
    …
};
```

derived class

```
class Manager: Employee {
    short level;
    bool is_manager_of(Employee);
    …
};
```

5

Inheritance Mechanism

- A class may *inherit* from a *base* class. The inheriting class is called a *derived* class.
- Inheritance can be viewed as an incremental refinement of a base class by a new derived class.
- The derived class has all the fields and methods the base class has, plus new ones it defines.
- A class may *override inherited methods* (member functions), but not inherited *data*.
- *A hierarchy* of related classes, that share code and data, is created.

base class

derived class

derived class

Arrows are always from a derived class to the base class, since the derived class knows about the base class, but not the opposite.

6

The Vector

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

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

7

The Checked Vector

Suppose we want a vector whose bounds are checked on every reference.

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

      // support unchecked access for efficiency
      int& elementAt(int i) { return Vector::operator[](i); }
      …
};

int& CheckedVector :: operator[](int i) {
   if (i < 0 || i >= size())
      error();
   return elementAt(i);
}
```

```
int main() {
   CheckedVector v(100);

   cin >> id;
   v[id] = 5;            // overridden checked access
   for (int i=0; i<v.size(); ++i) {
      v.elementAt(i)++;  // checking is unnecessary
   }
}
```

8

Derived Class Constructors

1. Constructors are never inherited. If a constructor is not defined for the derived class, the compiler automatically generates a constructor for it. This constructor first calls the *default constructor* of the base class, and then initializes the new data members of the derived class using their default constructors.

2. If the derived class defines a constructor of its own, first the *default constructor* of the base class is called, and only afterwards the derived class constructor.

3. If a constructor other than the default constructor should be invoked for the base class, this can be specified by passing parameters to it in the *initialization list*:

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

9

Points of Interest

`CheckedVector` overrides the indexing operator 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 explicitly:

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

10

Data Hiding and Derived Classes

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

```
class Vector {
   protected:
      int length,
      int* data;
   public:
      …
};
```

Protected data members may make derived classes dependent on the base class implementation.

11

Back to the Employees

```
class Employee {
   private:
      char* name;
      short year_born;
      short department;
      int   salary;
   public:
      void give_raise(int);
   …
};
```

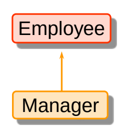
```
class Engineer: public Employee {
   private:
      char* degrees;
   …
};
```

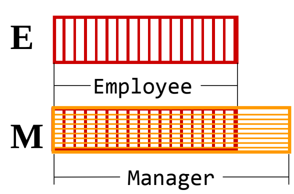
```
class Manager: public Employee {
   private:
      short level;
   public:
      bool is_manager_of(Employee);
   …
};
```

```
class SalesPerson: public Employee {
   private:
      double commission_rate;
   …
};
```

12

Inheritance as Type Extension



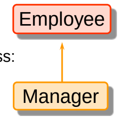



```
Employee  E;
Manager   M;

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

1. Call the (compiler generated) type casting operator from `Manager` to `Employee`, "slicing" all parts of `Manager` not contained in `Employee`.
2. Call the (compiler-defined or user-defined) `Employee` to `Employee` assignment operator.

Is-A Relationship

13



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

- `Manager` is an `Employee`
- `CheckedVector` is a `Vector`.

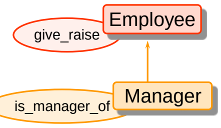
Thus, any function taking class B or B& as an argument, will also accept any class D derived from B.

```
bool inSameDept(Employee e1, Employee e2) {
   return e1.department == e2.department;
}
bool earnsMore(const Employee& e1, const Employee& e2) {
   return e1.salary > e2.salary;
}
void swap(Employee& e1, Employee& e2) {
  Employee t = e1;
  e1 = e2;
  e2 = t;
}
```

```
Manager m1, m2;
if (inSameDept(m1,m2))      // slices M1 and M2 to Employee
    swap(m1,m2);            // Unexpected result !
if (earnsMore(m1,m2))       // no slicing
    cout << m1.get_name() << "earns more" << endl;
```

Calling Methods of Inherited Types

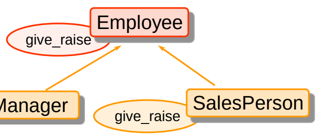
14




```
Employee  E;
Manager   M;

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

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

Calling Methods of Inherited Types

15

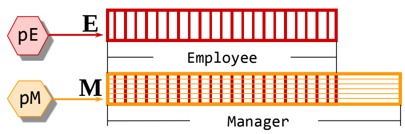


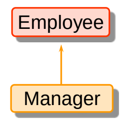

```
void SalesPerson::give_raise(int how_much) {
   Employee::give_raise(how_much + commission_rate * total_sales);
}
```

```
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()
```

Pointers to Inherited Types

16

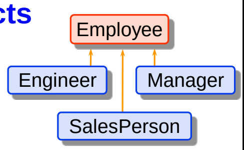





```
Employee  E, *pE;
Manager   M, *pM;

pE = &M;      //  OK
pM = &E;      //  Error
M  = *pE      //  Error
M  = *( (Manager*)pE );     //  OK … if you know what you are doing
```

Mixing Objects

17




```
Manager m;
SalesPerson s;

Employee dept[100];
dept[0] = m;  // Information lost
dept[1] = s;  // Information lost
```

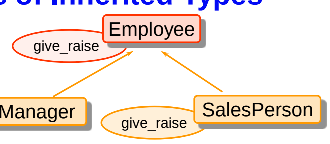
Mixed type collections should be implemented by an array of *pointers* to the base type:

```
Manager m;
SalesPerson s;

Employee* dept[100];
dept[0] = &m;
dept[1] = &s;
```

It is easy to store objects in this array, but not so easy to determine what type of object resides at each location once it's there !

Calling Methods of Inherited Types

18




```
Employee* dept[len];

for (int i=0; i<len; i++)
   dept[i]->give_raise(10);  // always calls Employee::give_raise() !
```

Determining the Object Type

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

Easy solution:
- use only objects of a single type

Bad solution:
- place an explicit type field in the base class

Better solution:
- use runtime type information

Best solution:
- use virtual functions

Polymorphism

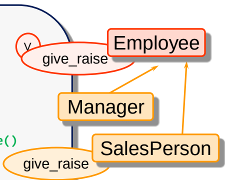
Virtual functions give full control over the behavior of an object if it is referenced via a base class pointer (or reference).

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

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


Polymorphism

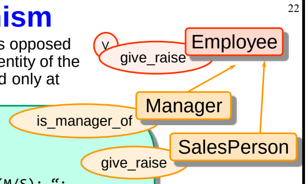
works **only** with pointers or references




```
Employee e;
Manager m;
SalesPerson s;

Employee* pe;

pe = &e;
pe->give_raise(10);    // Employee::give_raise()

pe = &m;
pe->give_raise(10);    // Employee::give_raise()

pe = &s;
pe->give_raise(10);    // SalesPerson::give_raise()

Employee e1 = s;
e1.give_raise(10);     // Employee::give_raise()

Employee& e2 = s;
e2.give_raise(10);     // SalesPerson::give_raise()
```


Polymorphism

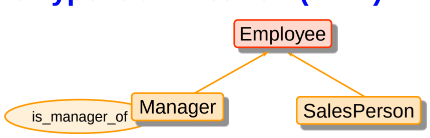
Polymorphism enables dynamic binding (as opposed to static binding). This means that the identity of the virtual function being called is determined only at run-time.




```
Employee* Dept[len];

for (int i=0; i<len; i++) {
   cout << "Enter type of employee (M/S): ";
   cin >> employee_type;
   if (employee_type == 'M')
      dept[i] = new Manager(...);
   else
      dept[i] = new SalesPerson(...);
   …
}

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

Dept[3]->is_manager_of(*Dept[1]); // Error
```

Although array elements are accessed via `Employee*`, **and** their type not known in advance, the appropriate `<class>::give_raise()` is always called !

Run Time Type Identification (RTTI)




```
Employee* Dept[len];

for (i=0; i<len; i++) {
   for (j=0; j<len; j++) {
      if (((Manager*)(Dept[i]))->is_manager_of(*Dept[j])
         cout << i << " is manager of " << j << endl;
         // Error: is_manager_of() does not always exist
   }
}
```


Run Time Type Identification (RTTI)

The operator `typeid` can identify an object's type at run time.
- It can be applied to an object or to a type name.
- It returns an object of type `type_info`.
- The type_info class has the overloaded `operator==`

In most cases, RTTI is **not** the right solution. Prefer virtual functions wherever possible.

```
Employee *Dept[len];

for (i=0; i<len; i++) {
   for (j=0; j<len; j++) {
      if (typeid(*Dept[i]) == typeid(Manager))
         if (((Manager*)(Dept[i]))->is_manager_of(*Dept[j])
            cout << i << " is manager of " << j << endl;
   }
}
```

25

Constructors

Constructors are never inherited. Each class must have its own constructor. If a class does not define a constructor, a default one is generated.

A constructor is never virtual. When constructing a new object, we always know its exact type. In other words, a constructor is never called through a pointer to a base class without knowing the precise type of the object.

```
Employee* pe = new SalesPerson();   // call SalesPerson constructor
```

When an object is constructed, it first calls its base class constructor, then its own constructor. If not specified otherwise, the default constructor of the base class is called.

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

26

Destructors

As opposed to a Ctor, a Dtor may be called through a pointer to a base class:

```
void deleteEmp(Employee* p) {
   delete p;   // calls the appropriate Dtor
}
```

the Dtor Employee::~Employee() **must** be virtual or else the wrong Dtor will be called.

Thus, any class that has virtual functions, or just may have derived classes with virtual functions, must define a virtual destructor.

```
class Employee {
   public:
      Employee(char *n, int d);
      virtual void give_raise(int how_much) { … };
      virtual ~Employee() { … };
      …
};
```

27

Abstract Classes

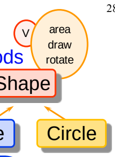
```
class Employee {
   …
   public:
      …
      virtual void give_raise(int how_much) = 0;   // pure virtual
};
```

- A class with one or more pure virtual functions is called an *abstract* class. Objects cannot be created from an abstract class
- Every derived class *must* implement virtual functions (or pass on the buck)
- A derived class that implements *all* pure virtual functions becomes a *concrete* class and can be used to generate objects. Otherwise it remains abstract.
- A pointer to an abstract class can be defined. In practice, it will always point to some concrete class deriving from the abstract class.
- Most useful for defining *interfaces*

28

Interfaces

An abstract class having *only* pure virtual methods



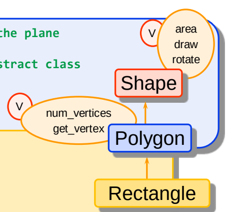

```
class Shape {
   public:
      virtual double area() const = 0;
      virtual void draw(int, int) const = 0;
      virtual void rotate(double) const = 0;
      …
};
```

```
class Square: public Shape {
   private:
      double edge_length;
   public:
      double area() const { return square(edge_length); }
      void draw(int x,y) const { … }
      void rotate(double angle) const { … }
      …
};
```

```
class Circle: public Shape {
   private:
      double radius;
   public:
      double area() const { return PI*square(radius); }
      void draw(int x,y) const { … }
      void rotate(double angle) const { // do nothing }
      …
};
```

29

```
struct Point2d { double x, y; };   // point in the plane

class Polygon: public Shape {      // another abstract class
   public:
      virtual int num_vertices() = 0;
      virtual Point2d get_vertex(int i) = 0;
};
```




```
class Rectangle: public Polygon {
   …
   void draw() { … }
   int num_vertices() { return 4; }
   Point2d get_vertex(int i) { … }
   …
};
```

```
double drawAll(const list<Shape*>& shapes, int x, int y) {
   for (list<Shape*>::const_iterator s = shapes.begin();
                                     s!= shapes.end(); s++) {
      s->draw(x,y);
   }
}

// new polygon that is the convex hull of p
Polygon* convexHull(const Polygon* p) { … }
```

30

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