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:

```c
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
    );
    // ...
};
```
Identifying the Common Part

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

C version:
```c
struct Engineer {
    struct Employee E;
    char *degree;
    /* ... */
};
```

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

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

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

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

Arrows are always from a derived class to the base class, since the derived class knows about the base class, but not vice-versa.
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. Same method prototype. An overridden function appears the same to users.

- 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.
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 appears the same to users.
- 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.
Inheritance Vector Example

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

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

int main()
{ Vector v(10);
    // Populate the vector, and then use it.
    // ..
}
```

What do we gain with such a primitive class?

In different situations we may want to use it as a template in real life.

What do we gain with such a primitive class?

In different situations we may want to use it as a template in real life.

C++11
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:

```cpp
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);
    }
};
```
When a `class` inherits from a `class/struct`, the `public` keyword should be used, as in

```cpp
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`

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

C++11 allows inherited constructors - by delegation
More Points of Interest

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

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

- **Since** **CheckedVector** doesn’t override the **size** function of **Vector**, the following calls the inherited function.

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

- An overridden function can be called using explicit scope:

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

```cpp
class Vector {
    protected:
        int len, *data;
    public:
        // ...
};
```
Inheritance as Type Extension

```java
Employee E;
Manager M;

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

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.
Is-A Relationship

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

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

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

```cpp
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.

Why is the result of type `Complex`?
Calling Methods of Inherited Types

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

Again, Manager is an Employee but not vice-versa!
Using Pointers to Inherited Types

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

```cpp
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

C-style casting is **dangerous** (not just error-prone)
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)

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);
}
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

Static type vs. Dynamic type
Determining the Object Type

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 four 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 RTTI (explicitly inquire about object’s type)
- Let the compiler do the work
Polymorphism

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

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) { ... };
    virtual void newAddress(string new_addrss) final;
};

class SalesPerson: public Employee { // The C++11 way
    ...
    double *commission_rate;
    public:
    ...
    void give_raise(int how_much) override { ... };
    void newAddress(string new_addrss); // Error
};
Polymorphism

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

Destructors cannot be inherited. They always call their base-class counterparts.

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

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

A Ctor of a base class that calls a virtual function, won’t call a derived-class’ version of that function.
Visibility of Virtual Functions

Given that being virtual is an implementation detail, it’s reasonable that it is hidden – called by a simple interface function.

They better be protected if they are used by derived classes (refinement) and private when they are fully replaced (overridden).

give_raise() should be private when Employee’s version is meant to be take it or leave it.

class Employee {
    public:
        Employee(char *n, int d);
        degree promote(string);
    protected:
        virtual give_raise(int);
};

class Manager : public Employee {
    public:
        ... Manager(char *n, int l, int d) : Employee(n,d), level l, group(0){}
    protected:
        give_raise(int) override;
};

Non-virtual, calls give_raise()
Visibility of Virtual Destructors

Destructors should be **virtual** only when they are used polymorphically – usually by using `delete` on a base class pointer.

If not, they better be non-virtual and **protected**.

```cpp
class Employee {
    public:
        Employee(char *n, int d);
    protected:
        ~Employee(); // non-virtual
        ... 
};
```

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

*Expecting derived classes that `delete` polymorphically.*

Look for "Virtuality", by Herb Sutter