Inheritance

Class inheritance and polymorphism
Suppose we want to computerize our personnel records.

First, we identify the types of employees that we have (two in this example):

```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

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

class Engineer: Employee {
    char *degree;
    // ...  
};

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

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

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

- The *Behavior* and the *Data* associated with every derived class are extension of the properties of the base (original) class.
- Thus, class `Engineer` and class `SalesPerson` extend, each one in its own way, the data and behavior of class `Employee`.
- Identifying the `Employee` common part helps us in defining more types, e.g.:

```cpp
class Manager: Employee {
    char *degrees;
    // ...
};
```
A class may \textit{inherit} from a \textit{base class}, in which case the inheriting class is called a \textit{derived class}.

A class may \textit{override} inherited \textit{methods} (member functions), but not inherited \textit{data}.

A \textit{Hierarchy} of related classes, that share code and data, is created.

Inheritance can be viewed as an \textit{incremental refinement} of a base class by a new derived class.

Arrows always point from the 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 the programmer to modify or extend a package someone else gave her without touching the package's code. Saves programmer time:

- **Consistency of Interface.** Same method prototypes. An overridden function appears the same to users. Saves user time: Easier learning and easier integration. Guarantees that interface to similar objects is in fact similar.

- **Polymorphism.** Different objects behave differently as a response to the same message.

All those services are available thanks to the power of Abstraction.

6

Introduction to Systems Programming
Here is a simple version of a Vector class which 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?

Function is forbidden! In other situations we may want to use `=default`, the compiler-generated (default) version.

C++11
A Checked Vector

We may also need a vector whose bounds are checked at every access.

We can define a new derived class `CheckedVector` which 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) {} // A primitive overriding
        int& operator[](int i) { 
            if (0 > i || i >= size())
                error(); // Need to throw an exception
            return Vector::operator[](i);
        }
};
```
When a class inherits from another class, the keyword `public` should be used, as in
```cpp
class CheckedVector: public Vector {
    ...
}
```
(This will be explained in class.)

Constructors are **not** Inherited. If we had not defined a constructor for `CheckedVector`, then the compiler would have tried to generate an empty default constructor, which would call an empty constructor of the base class `Vector`, and would fail in doing so since such a constructor does not exist. (Why?)

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

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

But C++11 allows delegation of constructors.
Points of Interest  (cont.)

- 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 class.)
- **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:
        // ...
};
```

Protected data members make derived classes addicting.
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! The compiler would not make guesses
```
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 an argument of type (class) B, will also accept there an argument of type (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+` using (twice) type conversion.
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!
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;` // **But it passes compilation**
- `M = *static_cast<Manager*>(pE);` // **MUCH safer**

**Introduction to Systems Programming**
In many cases it is convenient to have a **mixed type collection**. The simplest and easiest way to do so is to use an array of pointers to the base type.

```c
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);
}

A Design Error
Employee E;
Manager M;
SalesPerson S;

Employee *Dept[len];

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

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

The wrong way to override a method!
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 (that is, avoid the issue)
- Place a **type field** in the base class and update it in the constructor of the derived class (drawback: need to foresee the future)
- Use **RTTI** (explicitly inquire about object’s type)
- Let the compiler do the work!
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 (cont.)

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_addr) final;
};

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

Employee  E;
Manager    M;
SalesPerson S;

Employee *Dept[len];

Although all array elements are accessed via Employee*, the appropriate \texttt{<class>::give_raise()} is always called! The decision of which function to call is made in \textit{run-time}!

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

Yet, Dept[i]->is_manager_of(E) is illegal. Why?
Constructors and Destructors

Destructors are never inherited. They always call their base-class counterparts.

While Ctors are never virtual, Dtors may be virtual!

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

```cpp
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();
   };
```

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

Given that being a virtual function is an implementation detail, it makes sense to make it hidden, i.e., called by a simple interface function.

Virtual functions better be protected if they are used by derived classes (refinement), or 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
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
    ...
};

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.*