12 Class Relations

Kevin Schmidt, Susan Lindsey, Charlie Dey

Class Relations: “has-a”

A class usually contains data members. Data members can be simple types or *other classes*, allowing you to make structured code.

```
class Course {
private:
    Person the_instructor;
    int year;
}
class Person {
    string name;
    ....
}
```

The relation where one object contains another, is called a “**has-a**” relation between classes.

Literal and figurative “has-a”

A line segment **has a** starting point and an end point.

Consider a line segment with a starting and ending **`Point`**. A **`Segment`** class can store those points (left) or or store one **`Point`** and compute the other (right).

```
class Segment {
private:
    Point startp, endp;

public:
    Point get_end_point() {
        return endp;
    };
}
```

```
class Segment {
private:
    Point startp;
    float length,angle;
public:
    Point get_end_point() {
        // have to calculate
    };
}
```

Polymorphism in Constructors

You have to decide what to store and what to calculate, but you can construct a Segment in two ways:

```
class Segment {
   private:
      // up to you how to implement!
   public:
      Segment(Point start,float length,float angle)
         { ... } // Constructor 1
      Segment(Point start,Point end)
         { ... } // Constructor 2
}
```

Advantage: with a good API you can change your mind about the implementation without bothering the user.

Class Relations: “is-a”

From the textbook:

In addition to the *has-a* relation, there is the “*is-a*” relation, also called *inheritance*. Here one class is a special case of another class. Typically the object of the *derived* class (the special case) then also inherits the data and methods of the *base* class (the general case).

Class Inheritance

Inheritance in Object Oriented Programming can be described as a process of *creating* new classes from *existing* classes. New classes *inherit* some of the properties and behavior of the existing classes and may *override* others.

- The *existing* class is the **base** class
- The *created* class is the **derived** class
- The derived classes' *inherited* methods may/may not **override** the **base** class methods.

“is-a”: General Case, Special Case

You can have classes where an object of one class **is a** special case of the other class.

```
class General {
protected: // note!
 int g;
public:
 General(){ g = . . . };
 void general_method() { //calculate with g};
};
class Special : public General {
public:
  Special(){ g = ... };
  void special_method() {};
};
int main() {
  Special special_object;
  special_object.general_method();
}
```

Here, the `Special` class extends the `General` class.

Here, the `Special` class extends the `General` class.

Inheritance: derived classes

Derived class **`Special`** inherits methods and data from base class **`General`**:

```
int main() {
   Special special_object;
   special_object.general_method();
```

Members and methods need to be **`protected`**, not **`private`**, to be inheritable.

Constructors

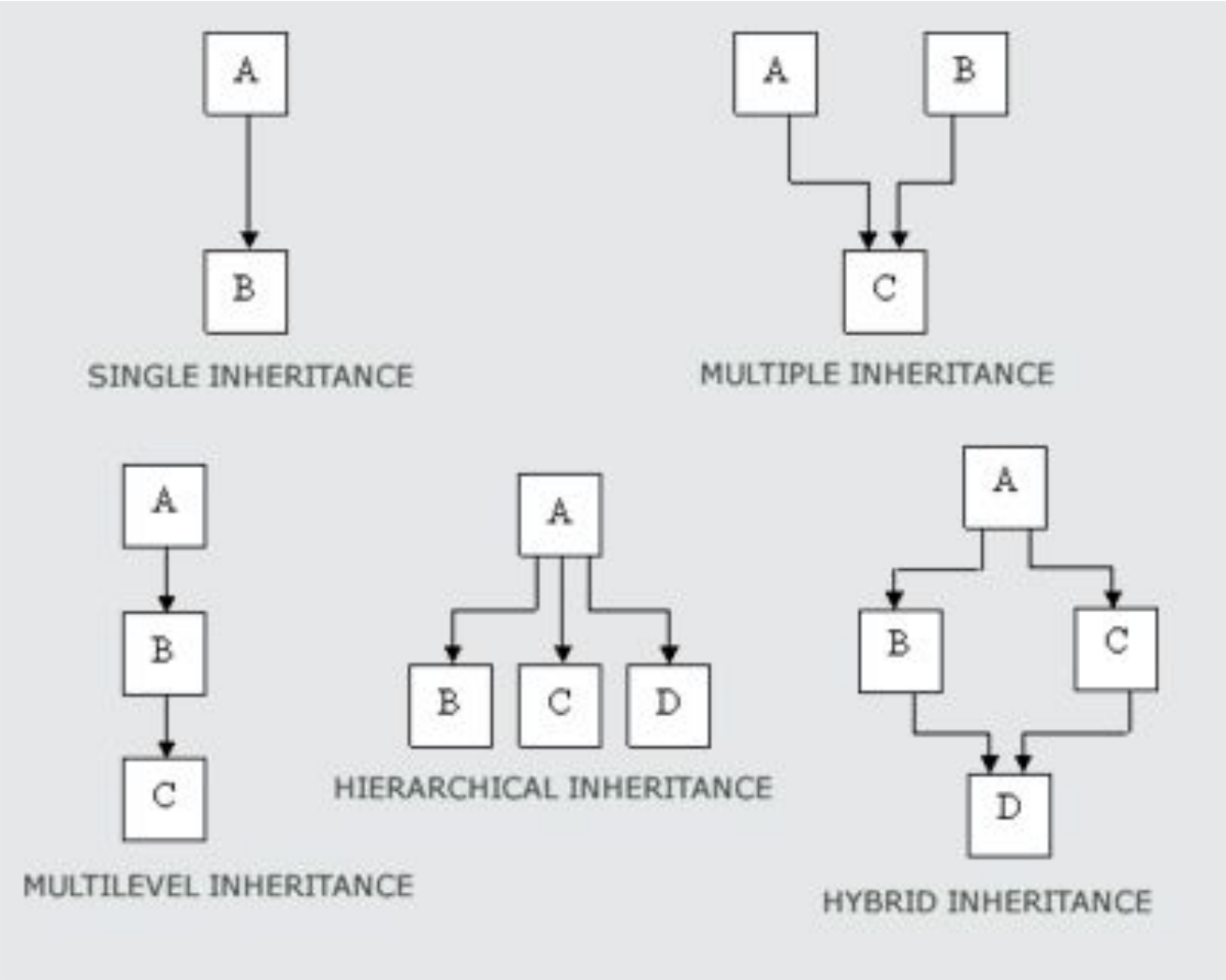
When you run the special case constructor, the general case needs to run, too.

By default the 'default constructor' would run, but you can specify a different constructor:

```
class General {
public:
   General( double x,double y ) {};
};

class Special : public General {
public:
   Special( double x ) : General(x,x+1) {};
};
```

Inheritance - go crazy



Access levels

Class methods and members/data can be:

- **`private`**, because they are only used internally;
- **`public`**, because they should be usable from outside a class object, for instance in the main program;
- **`protected`**, because they should be usable in derived classes.

Inheritance: Derived Classes

Derived class **Special** inherits methods and data from *base* class **General**:

```
int main() {
   Special special_object;
   special_object.general_method();
```

Data needs to be **protected**, not **private**, to be inheritable.