

## C++ Inheritance

### Chapter 6

## Public Inheritance

- Used to denote an **IS-A** relationship
- If derived class **D** publicly inherits from base class **B**, then every object of **D** can be used wherever a **B** may be used, **but not visa-versa.**
- Syntax: `class D : public B`
- Math can trip you up
  - Is a square a rectangle?

## Is it an IS-A ?

```

class Bird {
    public: fly();
};
class Penguin : public Bird {
    public: void fly(); ???
};

class Bird {
};
class FlyingBird() : public Bird {
    public: void fly();
};

class Penguin : public Bird {
};

```

## Inheritance is a Union

- The derived class gets the data members of the base class (not necessarily visible)
- The derived class may add more data members (sizeof object increases)
- The derived class inherits the public methods of the base class
- The derived class may override inherited methods and/or add new ones
- Implication: no array of base class

## Simple Example

```

class Animal {
    void speak() {"can't speak"}      Animal a; a.speak();
};
class Dog : public Animal {           Dog d; d.speak()
    void speak() {"woof"}
};
class Cat : public Animal {          Cat c; c.speak();
    void speak() {"meow"}
};
class Pig : public Animal {         Pig p; p.speak();
    void speak() {"oink"}
};
class DebateClub {
    bool tryout (const Animal&a) { a.speak() } ???
}

```

## C++ Dispatch != Java Dispatch

- Java dispatch is always **dynamic**
- C++ dispatch defaults to **static**
- Dynamic dispatch
  - determine actual method at runtime
  - What are the implications?
- Static dispatch
  - determine method at compile time

## C++ Static Dispatch

- Just a normal function call
- Compiler sees `a.speak()`
  - finds the function
  - generates code to call it (e.g. `jsr speak`)
  - linker will actually resolve address

## C++ Dynamic Dispatch

- Compiler sees `a.speak()`
- If we want to have dynamic dispatch, compiler must generate code like:
  - Find the function named "speak"
  - Invoke it (i.e. dispatch)
- Where is this lookup table stored?
  - each class has a table of virtual functions
- How is it referenced?
  - each object of class has a pointer (vptr) to table

## Function Lookup

- Can't just have has table of name/address (why not?)
- Furthermore, **operator+** is not a legal function name!
- Compiler does name **mangling** to give every function a "unique" name
- G++ translates
  - `std::string article::format(void)` to
  - `_ZN9wikipedia7article8formatEV`

## Implications of Dynamic Dispatch

- Object size grows by `sizeof(vptr)`
- Invoking a function requires extra lookup code (space and time)
- C++ made static dispatch the norm because of efficiency concerns.

## Specifying Dynamic Dispatch

- Use the C++ keyword **virtual** to mark member functions that should have **dynamic** dispatch
- Done on a function by function basis
- General rules:
  - If the intent is that a member function **may** be overridden, mark it virtual and it will be virtual for all derived classes
  - If **any** function virtual, destructor **should** be virtual
  - **Don't** override methods that aren't declared virtual
  - **Don't** call virtual methods in constructors, destructors

## C++ equivalent of super

- Use the scope resolution operator `::`
- Derived class D has a function `foo()` that overrides `foo()` in the base class B.
- For D's `foo()` to call B's `foo()` write `B::foo()`
- D's constructor can access B's constructor in the initialization list by `B(parameters)`

## Pure Virtual Functions

- A virtual function whose declaration is followed by **= 0**
- Declare as: `virtual foo(parameter list) = 0;`
- A class with at least one pure virtual function is like a Java abstract class
- A class with nothing but pure virtual functions is like a Java interface

## Non Virtual, Virtual, and Pure Virtual Functions

- Specifies **what** you want to inherit
- Non virtual: you inherit interface and a mandatory implementation (i.e. you should **not** override)
- Virtual: you inherit an interface and a default implementation which you **may** override.
- Pure virtual: you inherit an interface and you **must** supply an implementation.

## Inherited Names may be Hidden

```

Class Base {
    void mf1();
    void mf1(int); // overloaded name
    void mf3();
};
Class Derived : public Base {
    void mf1(int); // overrides base method
    void mf4();
}
Base b; b.mf1(); b.mf1(3); b.mf3(); b.mf4();
Derived d; d.mf1(5); d.mf3(); d.mf4(); d.mf1();

```

## Separate Interface/Implementation

- The interface of a function is its name, parameter list and return type
- The implementation of a function is the code that defines how the function works.
- There is often a default implementation that is put in the base class.
- Can prevent problems by separating (see Effective C++ by Scott Meyers)

## Airports and Airplanes

```
class Airport { ... }
class Airplane {
public:
    virtual void fly (const Airport& destination);
};
void Airplane::fly (const Airport& destination) {
    // default implementation of flying
};
class Boeing : public Airplane { ... }; // doesn't override fly
class Airbus : public Airplane { ... }; // doesn't override fly
Class Glider : public Airplane { .. }; // doesn't override fly – what is result?
```

## Alternate Design

```
class Airport { ... }
class Airplane {
public: virtual void fly (const Airport& destination) = 0; // an interface!
protected : flyJet(const Airport& destination);
};
void Airplane::flyJet (const Airport& destination) {
    // default code for flying a jet
};
class Boeing : public Airplane {
public virtual void fly (const Airport& destination) { flyJet(destination); }
};
class Airbus : public Airplane {
public virtual void fly (const Airport& destination) { flyJet(destination); }
};
Class Glider : public Airplane { .. }; // forced to override fly()
```

## Private Inheritance

- Used to model a HAS-A, or CAN\_BE\_IMPLEMENTED\_WITH
- Substitute **private** for **public** in declaration
- Nothing from base class visible from derived class.
- Purely an implementation technique. Base class provides most of what you need and a lot you don't need and don't want exposed.

## Multiple Inheritance

- C++ allows class to inherit from more than one parent
- May be used the way Java interfaces are used (i.e. implement multiple interfaces) if the classes are pure virtual classes
- May also be used to inherit from multiple concrete classes
- Syntax `class FooBar : public Foo, public Bar`

## Interesting issues

- Recall that inheritance is a union. So, if a class inherits from multiple classes, just union them all together.
- What happens if parent classes have identical data members or functions?
  - Multiple copies of everything?
  - Can have **virtual** inheritance!
  - How to distinguish which one you want?
  - Scope resolution operator `::` can help
- Multiple inheritance often is a graph rather than a tree
- Ask Dr Draper!

## Slicing

- Occurs when a derived type is assigned to a base type (e.g. in operator =)
- Derived class generally contains additional data members, thus its size is bigger.
- Memory layout is data member of base, followed by data members of derived classes.
- Slicing cuts off derived class data members
- What about virtual functions?

## Casting down the chain

- In Java one can: `Dog d = (Dog)Animal;`
  - A **ClassCastException** is thrown if the cast fails
- C++ provides a similar mechanism with **dynamic\_cast<>**
  - `Dog* d = dynamic_cast<Dog*>animalPtr`
  - The returned value is **null** if the cast fails

## Factories and Singletons

- A Factory is a function that returns different sub classes based on input
- Return value is generally a pointer to the base class object. Because of virtual methods, the object behaves correctly.
- Singleton is a class that has **one and only one** instance. How is this accomplished?

## Member vs non-member functions

- Object Oriented tells us to collect data and functions together.
- But, another goal of OO is encapsulation (i.e. information hiding)
- These conflict in the member/non-member decision.
- So, if something can be done without direct access to object's data, you may prefer non-member function. These are often "convenience" functions. Can be separated from class itself.

## Summary

- Inheritance is an **IS-A** relationship
- C++ provides both static/dynamic dispatch
- Destructor should be virtual if **any** functions are virtual.
- Slicing forces access to polymorphic objects via pointers or references
- C++ interfaces are classes with only pure virtual functions