THE UNIVERSITY
of ADELAIDE

School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

More about Inheritance

adelaide.edu.au

seek LIGHT

Previously on ADDS

- Class Hierarchies
  - Behavior Separation
  - How to build a good class hierarchy? Focus on the requirements
- Inheritance
  - Build classes from other classes
  - Why do we need inheritance?
  - Benefits
  - The derived class and the base class have an IS-A relationship
  - We don't actually inherit the constructors but we can (and should) use them in our own.

Example
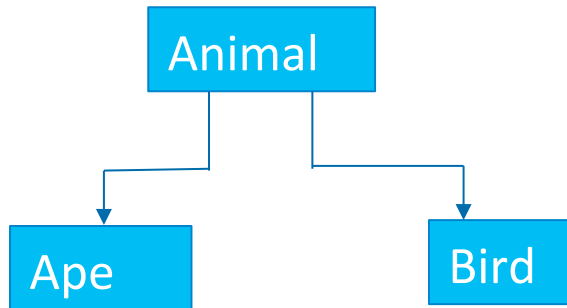

```
class Animal{
private:
    string location;

public:
    Animal();
    Animal(string loc);
    string getLocation();
};
```

```
class Bird : public Animal {
public:
    Bird();
    Bird(string loc);
};
```

```
Bird::Bird( ): Animal( ) { }
Bird::Bird(string loc):Animal(loc) {}
```

Access Specifiers

- getLocation method in Animal is **public** -> All of its children can use it.

- Animal may have **private** methods, e.g. idHash method that it uses for internal tracking -> Only Animal can access it.

- What about something like Chromosome information?
  - Animal AND its children should be able to use it, but not anyone else
  - **Protected**

```
class Animal{
private:
    string location;
    string IdHash();

protected:
    string ChromosomeInfo();

public:
    Animal();
    Animal(string loc);
    string getLocation();
};
```

Protected keyword

- **Private members** can only be accessed from the inside of the class.
- **Public members** can be accessed from any other classes.
- A **protected member** (data field or function) in a base class can be accessed in its derived classes.

- While, technically, we inherit almost everything from a parent, we can't USE everything from the parent, unless it's set up correctly.

Access Specifier on Inheritance

- Things that we inherit and use from a parent, are not necessarily visible from outside our class.

class Child : [access specifier] Parent { … };

```
class Bird : public Animal { ... };
class Bird : private Animal { ... };
class Bird : protected Animal { ... };
```

Types of Inheritance

Public inheritance			
Base access specifier	**Derived access specifier**	**Derived class access?**	**Public access?**
Public			
Private			
Protected			

Protected inheritance			
Base access specifier	**Derived access specifier**	**Derived class access?**	**Public access?**
Public			
Private			
Protected			

What else can you guess?

Hint: protected inheritance vs private inheritance

Private inheritance			
Base access specifier	**Derived access specifier**	**Derived class access?**	**Public access?**
Public			
Private			
Protected			

Types of Inheritance (continue)

- With public inheritance
  - Public members of parent become ▮▮▮▮▮ members of child
  - Protected members of base become ▮▮▮▮▮ members of child
- With protected inheritance
  - Public and protected members of parent become ▮▮▮▮▮ members of child
- With private inheritance
  - Public and protected members of parent become ▮▮▮▮▮ members of child

- **Quick Quiz**

True or false?

Grand children always have access to protected members of grand parents.

False.

Inheritance: some points

- Regardless of access specifier:
  - The derived class inherits all public and protected members from the base class
  - The derived class has no access to the private members of the base class
  - The constructors and destructors of the base class are not inherited
  - The assignment operator is not inherited
  - The friend functions and classes of the base class are not inherited

Example

- Suppose you want to compare two objects to see if they're equal:

- You could use the accessor methods to carry out comparison - what's the problem with that?

- You could access rectly - what's the poten

```
bool equal(Circle c1, Circle c2)
{
    return (c1.getRadius() == c2.getRadius());
}
```

- Let's say we want to define an external function that could still get access to the private members of a class.

```
bool equal(Circle c1, Circle c2)
{
    return (c1.radius == c2.radius);
}
```

Friend

- A friend function of a class has access to the private members of that class.
- A friend function can directly read and change the value of a member variable.

```cpp
class Circle{

public:
    friend bool equal (Circle c1, Circle c2);


};


bool equal(Circle c1, Circle c2)
{
    return (c1.radius == c2.radius);
}
```

Friend Class

```
class A{
  friend class B;

private:
  string secret;
};
```

- You can also have a friend class.

- B has access to secret.

Friend

- Friendship isn't reciprocal.
- Friendship isn't transitive.
- Friendship isn't inherited.

- You should use members when you can and friends when you have to.
  - The choice is based on design, syntactic suitability, and when it's much harder to do it the other way.

Overloading

- Overloading lets us reuse the same name but for different situations.

- Functions have the same name but different parameter.
  - Overloading functions can make programs clearer and more readable.
  - Based on different return types?
    - No. Overloaded functions must have different parameter lists

Overloading Operators

- This isn't just limited to functions.
- We can overload operators (+,-,<<,>>) as well.

Operators That Can Be Overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete

Operators That Can Not Be Overloaded			
?:	.	.*	::

Friends and operators

- If we overload an operator, we want to mimic the existing operator behavior, but using one of our classes.

```cpp
class Circle{
public:
  friend Circle operator +(Circle, Circle);
private:
  int area;
};

Circle operator +(Circle c1, Circle c2){
  Circle tmp;
  tmp.area = c1.area + c2.area;
  return tmp;
}
```

Multiple Inheritance

- When something inherits behavior from two (or more) different objects!

- Deriving directly from more than one class is called multiple inheritance.

```
class Child : public Parent1, public Parent2{

    /* more code goes here*/
}
```

- How does this look like?
- What are the possible problems?

Multiple Inheritance

- The child gets all of the parents' behaviors.
- The order of derivation matters only for the order of default initialization and cleanup by constructors and destructors.

- What happens if you derive from one class more than once? I mean indirectly!
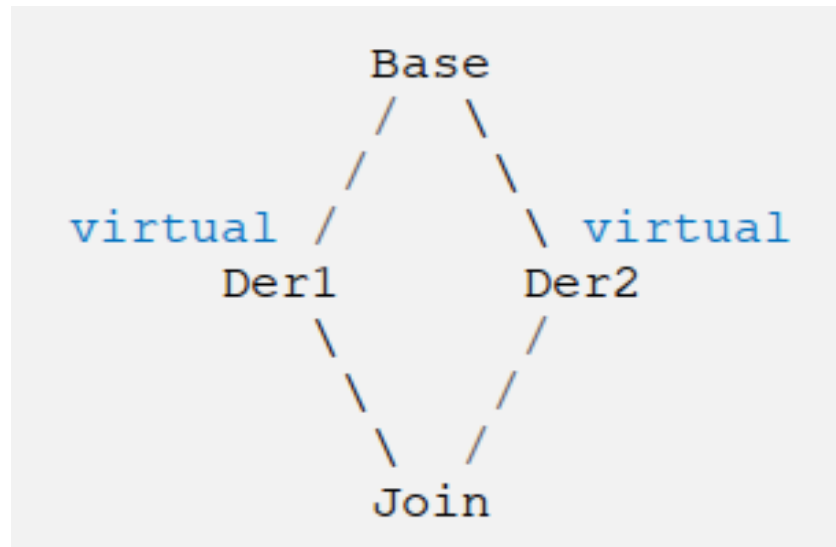
Example

```
class Animal { ... };
class Bird : public Animal { ... };
class SeaC : public Animal { ... };

class Penguin :
  public Bird, public SeaC { ... } ;
```

- – This will compile.

- • But we have ambiguity. Why?

  - – We have to refer to `Bird::Animal` or `SeaC::Animal`.

  - – This is called a qualified class name.

Using `virtual`

- Referring to the same base class through different inheritance pathways is ambiguous if we are referring to **two *different* versions of the base class**.

- Using `virtual` in the declaration of inheritance means that the grand child inherit only one subobject of the base class.

```
                    Base
                    /   \
                   /     \
          virtual /       \ virtual
             Der1           Der2
                 \         /
                  \       /
                   \     /
                    \   /
                    Join
```

- Problem solved by using virtual

Example

```
class Animal { ... };
class Bird : virtual public Animal { ... };
class SeaC : virtual public Animal { ... };

class Penguin :
  public Bird, public SeaC { ... } ;
```

– This will compile.

- We no longer have ambiguity because all references to `Animal` in `Penguin` go to the same class.

- Look carefully at where we use virtual to control this.

Multiple Inheritance

- It's possible to mix up the use of `virtual` and have some derived classes sharing a base class and some not.

- Always make sure you understand why you are using a keyword!

Summary

- Inheritance is a very useful technique but you need to know how to use it effectively.
    - Behaviors are inherited.
    - Access modifiers.
    - When should I overload?
    - Multiple inheritance

- We'll talk about this more in Polymorphism.
    - When should I redefine/override?