

Inheritance

Wagner Truppel
Lecturer, Dept. of Computer
Science & Engineering
UC Riverside

wagner@cs.ucr.edu
<http://www.cs.ucr.edu/~wagner>

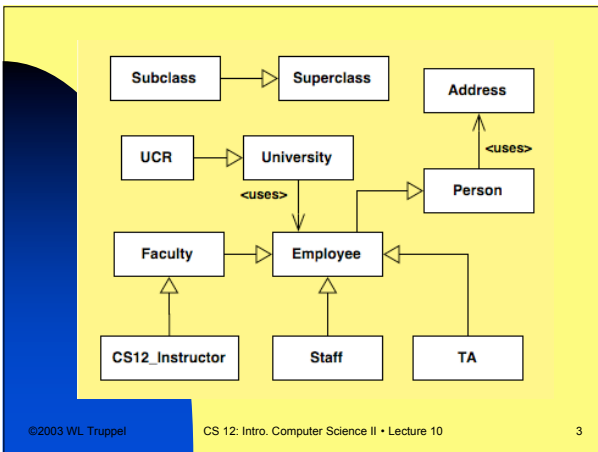
<http://www.cs.ucr.edu/cs12>

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 1

What is it ?

- Recall that classes can be organized in a hierarchical fashion (figure in next slide)
- Typically, classes which are more general appear towards the top of the hierarchy
- More specialized classes have the same member variables and most member functions from their parent classes
- They may also **add** member variables and member functions of their own
- They may also **change** the behavior of the member functions they get from their parent classes

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 2



Advantages

- Great organizational structure to help you better design your code
- "Easy" to debug, test, and maintain b/c you do it one class at a time
- Avoids un-necessary code duplication: all derived classes from a given base class get the same code from it
- Hierarchies are very expressive structures: lots of real-world stuff may be efficiently organized in hierarchies
- Allow for common coding patterns to be developed and documented, so that no one has to reinvent the wheel all the time (Design Patterns)

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 4

Is-a & has-a relations

- How to discover inheritance relations among classes?
- A **Staff** person *is an* **Employee**; an **Employee** *is a* **Person**
- A **JetEngine** *is an* **Engine**
- A **Jet** *is an* **Airplane**, but is not an **Engine**; a **Jet** *has an* **Engine**
- The **University** *uses (has)* **Employees**
- How about this ?
 - ◆ a **Person** *is an* **Employee**

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 5

C++ inheritance syntax

Person	
-	string name
+	Person(string aName)
+	string getName()

```

class Person
{
private:
    string name;

public:
    Person(string aName);
    string getName();
};

Person::Person(string aName) : name(aName)
{}
    
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 6

C++ inheritance syntax

```

classDiagram
    class Person {
        -string name
        +Person(string aName)
        +string getName()
    }
    class Employee {
        -double salary
        +Employee(string aName, double sal)
        +double getSalary()
    }
    Person <|-- Employee
        
```

```

class Employee : public Person
{
private:
    double salary;

public:
    Employee(string aName, double sal);
    double getSalary();
};

Employee :: Employee (string aName, double sal) : Person(aName), salary(sal)
{}
        
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 7

C++ inheritance syntax

```

classDiagram
    class Person {
        -string name
        +Person(string aName)
        +string getName()
    }
    class Employee {
        -double salary
        +Employee(string aName, double sal)
        +double getSalary()
    }
    class TA {
        -string course
        +TA(string aName, double sal, string crse)
        +string getCourse()
    }
    Person <|-- Employee
    Employee <|-- TA
        
```

```

class TA: public Employee
{
private:
    string course;

public:
    TA(string aName, double sal, string crse);
    string getCourse();
};

TA :: TA (string aName, double sal, string crse) : Employee(aName, sal), course(crse)
{}
        
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 8

Important

- Constructor chain
 - ◆ A constructor of a derived class (**TA**, **Employee**) must first call a constructor of its base class (**Employee**, **Person**)
 - ◆ Why ?
- Destructor chain
 - ◆ Same idea as above, but in the **opposite order**
 - ◆ Why ?
- An object of a derived class has many types (**polymorphism**)
 - ◆ A **TA** object is also an **Employee** object and is also a **Person** object

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 9

Access modifiers

- All member variables are inherited, but not all are necessarily accessible inside the derived class
 - ◆ **private** member vars are **not** directly accessible by name in the derived class
 - ◆ **private** means: no class other than the one where the member is defined (and friend classes) has access to it
- Can we have members which are directly visible only in derived classes but private for other classes?

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 10

Access modifiers

- Can we have members which are directly visible only in derived classes but private for other classes? Yes !
- New access modifier: **protected**
- **protected** means: public in derived classes, private for every other class
- These access modifiers (public, protected, private) apply to member functions as well as to member variables

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 11

Access modifiers

- All member vars and most member functions are inherited, but not all are necessarily accessible inside derived classes
 - ◆ **private** members are **not** directly accessible by name in any class other than the class they're defined in and in friend classes
 - ◆ **protected** members **are** directly accessible by name in all derived classes of the class they're defined in, but behave as private for all other classes (except friends)
 - ◆ **public** members **are** directly accessible by name in **all** classes, derived or not, friends or not
- Constructors, destructors, and the assignment operator are **not** inherited
- Private member functions are **effectively** not inherited

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 12

Access modifiers

```

classDiagram
    class Person {
        # string name
        + Person(string aName)
        + string getName()
    }
    class Employee {
        # double salary
        + Employee(string aName, double sal)
        + double getSalary()
    }
    class TA {
        # string course
        + TA(string aName, double sal, string crse)
        + string getCourse()
    }
    Person <|-- Employee
    Employee <|-- TA
        
```

```

class Person
{
    protected:
    string name;
    // rest is the same as before
};

class Employee : public Person
{
    protected:
    double salary;
    // rest is the same as before
};

class TA: public Employee
{
    protected:
    string course;
    // rest is the same as before
};
        
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 13

Redefining member fs

- **Extending:** just adding a function to a derived class which didn't exist in the base class
- **Overloading:** same name, different signatures (old news...)
- **Redefining:** changing the code of an inherited member function; that is, having different functions with the **same** signature
- **Overriding:** redefinition of **virtual** functions (more on this later)

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 14

Extending class behavior

```

classDiagram
    class Person {
        - string name
        + Person(string aName)
        + string getName()
    }
    class Employee {
        - double salary
        + Employee(string aName, double sal)
        + double getSalary()
    }
    class TA {
        - string course
        + TA(string aName, double sal, string crse)
        + string getCourse()
    }
    Person <|-- Employee
    Employee <|-- TA
        
```

```

class TA: public Employee
{
    private:
    string course;

    public:
    TA(string aName, double sal, string crse);
    string getCourse();
};

// here's an example of extending a class
string TA::getCourse()
{ return course; }
        
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 15

Redefining member fs

```

classDiagram
    class Lecturer {
        +constructor(s)
        +teach()
    }
    class Wagner {
        +constructor(s)
        +teach()
    }
    Lecturer <|-- Wagner
        
```

```

class Lecturer
{
private:
    // stuff

public:
    // constructor(s)
    // stuff
    void teach();
};

class Wagner : public Lecturer
{
private:
    // more stuff

public:
    // constructor(s)
    // more stuff
    void teach();
};

void Lecturer::teach()
{ /* do the teaching thing... */ }

void Lecturer::teach()
{ /* teach with slides ! */ }
        
```

- Wagner redefines the inherited member function `teach()` to do it differently.
- Note that now `teach()` does appear in the class declaration and in the UML diagram.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 16

Virtual functions

```

classDiagram
    class Engine {
        +constructor(s)
        +start()
        +run()
    }
    class JetEngine {
        +constructor(s)
        +run()
    }
    Engine <|-- JetEngine
        
```

- Say you have a class **Engine**, with member functions `start()` and `run()`
- Say that `start()` calls `run()`

```

class Engine
{
public:
    void start();
    void run();
};

void Engine::run() { /* run the engine */ }
void Engine::start()
{
    // do some preparatory work
    // then run the engine
    run();
}
        
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 17

Virtual functions

```

classDiagram
    class Engine {
        +constructor(s)
        +start()
        +run()
    }
    class JetEngine {
        +constructor(s)
        +run()
    }
    Engine <|-- JetEngine
        
```

- Say you have a class **JetEngine**, deriving from **Engine**
- Say that **JetEngine** redefines `run()`, but not `start()`

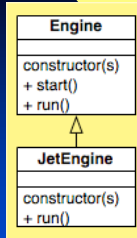
```

class JetEngine
{
public:
    void run();
};

void JetEngine ::run()
{
    // run at a higher temperature and
    // using a special fuel
}
        
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 18

Virtual functions



- Now suppose we create an object of class **JetEngine** and call **start()** on it

```
JetEngine f18;
F18.start();
```

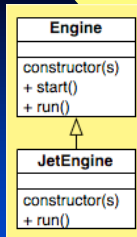
- The question is: which version of **run()** gets executed?
- You have a minute to answer...

©2003 WL Truppel

CS 12: Intro. Computer Science II • Lecture 10

19

Virtual functions



- Now suppose we create an object of class **JetEngine** and call **start()** on it

```
JetEngine f18;
F18.start();
```

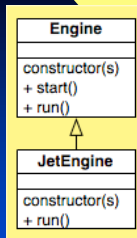
- The question is: which version of **run()** gets executed?
- No, it's not **JetEngine**'s version, even though that's what we'd like to happen. It's **Engine**'s **run()** that gets executed because that's what the compiler knew at the time **start()** was compiled.
- Is there a way to make sure it's the **JetEngine**'s version of **run()** that gets executed? Hold on to that thought...

©2003 WL Truppel

CS 12: Intro. Computer Science II • Lecture 10

20

Virtual functions



- Now, what if it didn't make sense to have **run()** defined in the class **Engine**? After all, no one knows how to run an undifferentiated engine

- Could we still have **start()** defined there, even though it calls **run()** and **run()** isn't defined there anymore?
- What would be the alternative if the answer was no?
- Another minute for you to think about it...

©2003 WL Truppel

CS 12: Intro. Computer Science II • Lecture 10

21

Virtual functions

Engine

constructor(s)

+ start()

+ virtual run()

↑

JetEngine

constructor(s)

+ run()

- The answer to both problems
 - ◆ It's **Engine's** version of **run()** that gets executed when you call **start()** on a **JetEngine** object
 - ◆ How to have **start()** be defined and call **run()** in a class that doesn't itself define **run()**
- is to tell the compiler, somehow, to wait until the right **run()** is available
- In other words, we need **late binding** (also called **dynamic binding**) as opposed to **static binding**
- This is accomplished using the keyword **virtual**

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 22

Virtual functions

Engine

constructor(s)

+ start()

+ virtual run()

↑

JetEngine

constructor(s)

+ run()

```

class Engine
{
public:
    void start();
    virtual void run();
};
void Engine::run() { /* run the engine */ }
void Engine::start()
{
    // do some preparatory work
    // then run the engine
    run();
}
    
```

- By marking **run()** in **Engine** as **virtual**, we're telling the compiler to use the implementation of the caller's class, not the implementation defined where **start()** was defined

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 23

Virtual functions

Engine

constructor(s)

+ start()

+ virtual run()

↑

JetEngine

constructor(s)

+ run()

- Now, when we do


```
JetEngine f18;
F18.start();
```
- It's *still* **Engine's** **start()** which gets executed (**JetEngine** did not redefine **start()**)
- But it's **JetEngine's** **run()** which gets called from within **start()** !
- How about the other problem? What if it doesn't make sense to have **run()** defined in **Engine**? Can we still have **start()** defined there even though it calls **run()**?

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 24

Virtual functions

```

classDiagram
    class Engine {
        + constructor(s)
        + start()
        + pure virtual run()
    }
    class JetEngine {
        + constructor(s)
        + run()
    }
    Engine <|-- JetEngine
        
```

- How about the other problem? What if it doesn't make sense to have **run()** defined in **Engine**? Can we still have **start()** defined there even though it calls **run()**?
- Same idea... make **run()** **virtual**, but give it no executing body. That is, make it a **pure virtual function**.

```

class Engine
{
public:
    void start();
    virtual void run() = 0; // now run() is a pure virtual function
};

void Engine::start()
{
    // do some preparatory work
    // then run the engine
    run();
}
        
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 25

Virtual functions

- Now, when we do
`JetEngine f18;`
`F18.start();`
- It's *still* **Engine's start()** which gets executed (**JetEngine** did not redefine **start()**)
- But it's **JetEngine's run()** which gets called from within **start()** since **Engine** doesn't even have a **run()** function defined in it.
- Now... can we try to declare an object of class **Engine**?
`Engine eng; // is this possible ?`

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 26

Virtual functions

- Now... can we try to declare an object of class **Engine**?
`Engine eng; // is this possible ?`
- It was possible before (even when **run()** was virtual), but now that **run()** is **pure** virtual, it's not possible anymore. Why?
- Well, what would happen if someone tried to call **start()** on that object?
`Engine eng;`
`eng.start();`
- Which **run()** function gets executed? There isn't any version available!
- So... a class having at least one **pure** virtual function **cannot** be instantiated. It's then called an **abstract class**. (Note the *italicized* name in the UML diagram)

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 27

The slicing problem

- You cannot slice the cake and eat it too... no, just kidding
- Ok, say that **run()** is just virtual, not pure virtual, so that **Engine** is not an abstract class (it's then called a **concrete class**)
- Now let's create some objects.
Can we do:
JetEngine f18 = Engine(); ? No !
Engine eng = JetEngine(); ? Yes !

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 28

The slicing problem

- Can we do:
JetEngine f18 = Engine(); ? No !
 - * Of course not. Not every engine is a jet engine. And you can't coerce a general engine into a specific one either...
- Engine eng = JetEngine(); ? Yes !
 - * Yes, all jet engines are engines, so this works. But... there is a problem.
 - * If we do this, we lose the jet engine's identity as a jet engine. It now thinks it's only a regular engine.
 - * Its jet-engine-specific member variables and functions are NOT accessible any more.
- This "memory loss" is called the slicing problem

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 29

The slicing problem

- The correct way to treat a derived class object as an object of its base class is to use pointers
Engine* engPtr;
JetEngine* jetPtr = new JetEngine();
engPtr = jetPtr;
- Now we can treat the jet engine as a regular engine without losing the jet engine's identity
- This is useful sometimes, but I won't go into any more detail
- Read more in the textbook

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 30

A really bad idea

- Suppose we have a bunch of animal classes: **Dog, Cat, Lion, Sheep**, etc
- Suppose they all have a member function **feed()**.
- Suppose we have a bunch of animal objects: **Dog fluffy, Dog fang, Cat garfield, Lion lippy, Sheep dolly**, and we want to feed them all. Easy, right?

```

fluffy.feed();
fang.feed();
garfield.feed();
lippy.feed();
dolly.feed();

```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 31

A really bad idea

- Yes, but what if we didn't know these animals ahead of time? That is, what if they're created interactively by the user of our program. Not a big deal, right?
- First, create a base class **Animal**, from which the others are all derived and have **feed()** be defined *only* in **Animal**

```

// pseudo-code...
void Animal::feed()
{
    if (this animal is a Dog)
        { /* do the dog-feeding thing */ }
    else if (this animal is a Cat)
        { /* do the cat-feeding thing */ }
    else if (this animal is a Lion)
        { /* do the lion-feeding thing */ }
    else if (this animal is a Sheep)
        { /* do the sheep-feeding thing */ }
}

```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 32

A really bad idea

- First, create a base class **Animal**, from which the others are all derived and have **feed()** be defined *only* in **Animal**

```

// pseudo-code...
void Animal::feed()
{
    if (this animal is a Dog)
        { /* do the dog-feeding thing */ }
    else if (this animal is a Cat)
        { /* do the cat-feeding thing */ }
    else if (this animal is a Lion)
        { /* do the lion-feeding thing */ }
    else if (this animal is a Sheep)
        { /* do the sheep-feeding thing */ }
}

```

- Bad idea!** What if you later have **Zebras, Tigers, Elephants**, etc? You're going to have to search for all these if statements and change them to include the new animals.

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 33

A smart idea

- First, create a base class **Animal**, from which the others are all derived and have **feed()** be a **pure virtual function** in **Animal**. Then, have each derived class define its own feeding behavior.
- Now, you can call **feed()** on *any* **Animal** and the right feeding method will be executed, *even if you later define other derived classes from Animal*.
- For example, suppose you have a dynamically allocated array of **Animal** objects. Then, instead of


```
for (int i = 0; i < size; i++)
{
    if (array[i] is a Dog) { /* feed a dog */ }
    elseif (array[i] is a Cat) { /* feed a cat */ }
    // etc etc etc
}
```

 you simply have


```
for (int i = 0; i < size; i++)
{ array[i].feed(); }
```

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 34

A really smart idea

- For example, suppose you have a dynamically allocated array of **Animal** objects. Then, you simply have


```
for (int i = 0; i < size; i++)
{ array[i].feed(); }
```
- But that won't quite work yet... because of the slicing problem
- If you put a bunch of **Animal** objects in an array of **Animals**, they'll lose their identity. They **have** to! Why?
- So... the solution is to have an array of *pointers* to **Animal** objects


```
Animal* array[numberOfAnimals];
// initialize the array
for (int i = 0; i < size; i++)
{ array[i]->feed(); }
```
- Late binding (the virtual function idea) allows for **automatic polymorphic behavior**

©2003 WL Truppel CS 12: Intro. Computer Science II • Lecture 10 35
