CSE 1325: Object-Oriented Programming

Lecture 19

Custom C++ Types

OOP in C++

Mr. George F. Rice

george.rice@uta.edu

Office Hours:
Prof Rice 12:30 Tuesday and Thursday in ERB 336
For TAs see this web page

2000 mockingbirds == 2 kilomockingbird

Today's Topics

- Types in C++
  - Enum, enum class, struct, and class
  - Friends, guards and initialization lists
- Intro to Inheritance in C++
  - Protected class members
  - Class hierarchies
  - Implementation
  - A taste of polymorphism
  - Pure virtual methods
- Multiple Inheritance in C++
  - Resolving ambiguity
  - The Diamond Problem
  - Wither multiple inheritance?

Writing Simple C++ Types

- C++ provides 4 custom types
  - **Enum**: Just a list of words assigned an integer each
    - Note that enums in C++ do NOT contain data or methods, so they are an extremely limited version of a "class"
  - **Enum Class**: An enum that won't interact with other types
    - Harder to use, but enables more code validation by the compiler
    - Despite the confusing name, *still* contains no data or methods!
  - **Struct**: A list of data declarations with no supporting code
    - By convention, at least. Actually, **`struct`** is *almost* a synonym for **`class`** in C++.
    - In CSE1325, we will *always* use **`class`**, NOT **`struct`**!
  - **Class**: Both data and supporting code, much like Java

Defining a C++ Enum

- A C++ enumeration is similar to Java except
  - No methods or additional fields are allowed
  - Integer values may be assigned in the declaration in place of Java's constructor syntax with unlimited fields
  - Printing the enum shows the int, NOT the enum name (though often **`operator<<`** is overridden to fix this)

```
enum Month {January =  1, February =  2, March     =  3,
            April   =  4, May      =  5, June      =  6,
            July    =  7, August   =  8, September =  9,
            October = 10, November = 11, December  = 12};

 int main() {
   Month month = January;
   std::cout << "January is " << month
      << ", May is " << May
      << ", and December is " << December
      << "." << std::endl;
 }
```

```
ricegf@pluto:~/dev/cpp/201908/03/code_from_slides$ g++ --std=c++17 enum.cpp
ricegf@pluto:~/dev/cpp/201908/03/code_from_slides$ ./a.out
January is 1, May is 5, and December is 12.
ricegf@pluto:~/dev/cpp/201908/03/code_from_slides$
```

Sequential Enum Values

- A C++ enumeration begins numbering at 0 by default
  - If an int is specified, numbering continues from there

```
enum Month {January =  1, February, March    ,
            April       , May     , June     ,
            July        , August  , September,
            October     , November, December };

 int main() {
   Month month = January;
   std::cout << "January is " << month
      << ", May is " << May
      << ", and December is " << December
      << "." << std::endl;
 }
```

```
ricegf@pluto:~/dev/cpp/201908/03/code_from_slides$ g++ --std=c++17 enum.cpp
ricegf@pluto:~/dev/cpp/201908/03/code_from_slides$ ./a.out
January is 1, May is 5, and December is 12.
ricegf@pluto:~/dev/cpp/201908/03/code_from_slides$
```

Defining a C++ Enum Class

- An enum *class* is an enum with no int equivalents
  - The type is strictly enforced
  - **`Month m = 3;`** is an error!
  - Printing is still NOT automatic as it is in Java

```
#include <iostream>

enum class Month {Jan, Feb, Mar, Apr, May, Jun,
                  Jul, Aug, Sep, Oct, Nov, Dec};

std::string to_string(Month m) {
    switch(m) {
        case Month::Jan: return "January"  ;
        case Month::Feb: return "February" ;
        case Month::Mar: return "March"    ;
        case Month::Apr: return "April"    ;
        case Month::May: return "May"      ;
        case Month::Jun: return "June"     ;
        case Month::Jul: return "July"     ;
        case Month::Aug: return "August"   ;
        case Month::Sep: return "September";
        case Month::Oct: return "October"  ;
        case Month::Nov: return "November" ;
        case Month::Dec: return "December" ;
        default: return "Unknown";
    }
}

int main() {
   Month month = Month::Jan;
   std::cout << "January is " << to_string(month)
      << ", May is "          << to_string(Month::May)
      << ", and December is " << to_string(Month::Dec)
      << "." << std::endl;
}
```

:: is the C++ membership operator
(same for enums as . in Java)

Enum Summary

- C++ supports simple enum and enum classes
  - Enums are identical to C – basically names for ints
    - Though you can at least specify the int for each enumeration using assignment
  - Enum *classes* are more strictly enforced
    - Instead of “GREEN” you MUST use Color::GREEN
    - Again, :: is the “membership” operator in C++
    - Important: An enum class is NOT a class! Just a strict enum.
  - No constructors or members with enum OR enum class
    - This is different from Java’s enum, which is a full-fledged class with constructors, methods, fields...

C++ Struct

A C++ struct is a class with public data by default
and (*by convention only*) no methods

```cpp
#include <iostream>

struct Date {
    int year, month, day;
};

int main() {
    Date birthday;
    birthday = {12, 30, 1950};  // Dr. Stroustrup’s, not mine!
    std::cout << birthday.month << '/'
              << birthday.day   << '/'
              << birthday.year  << std::endl;
}
```

Oops!

```
student@cse1325:/media/sf_dev/06$ make struct1
g++ --std=c++17    struct1.cpp   -o struct1
student@cse1325:/media/sf_dev/06$ ./struct1
30/1950/12
student@cse1325:/media/sf_dev/06$ 
```

In my humble opinion, a struct is a C feature. C++ should use classes!
Not everyone agrees with me…
C++ programmers often call fields “class variables”
and methods “class functions”. We’ll stick with fields & methods.

C++ Class

A C++ class looks suspiciously like a Java class, except

- C++ classes are always public and must end with a ;
- Visibility is by sections (e.g., public:) rather than per member
- Members are (often) declared in a .h file but defined in a .cpp file
  - This separates interface from implementation, which I consider an *excellent* feature!

```
#include <iostream>

class Date {
  public:
    Date(int year, int month, int day)
        : _year{year}, _month{month}, _day{day} {
        if (1 > month || month > 12) throw std::runtime_error{"Invalid month"};
        if (1 > day   ||   day > 31) throw std::runtime_error{"Invalid day"};
    }
    void print_date() {
        std::cout << _month << '/' << _day << '/' << _year << std::endl;
    }
  private:
    int _year, _month, _day;
};
```

Constructors

Methods

Private Data

The ; is *required*!

```
student@cse1325:/media/sf_dev/06$ make class
g++ --std=c++17    class.cpp   -o class
student@cse1325:/media/sf_dev/06$ ./class
12/30/1950
student@cse1325:/media/sf_dev/06$
```

C++ .h and .cpp Files

Declarations of methods and fields are specified in the .h file

date.h

```
#include <iostream>

class Date {
  public:
    Date(int year, int month, int day);
    void print_date();
  private:
    int _year, _month, _day;
};
```

Implementations (definitions) are specified in the .cpp file

- `Date::print_date()` means "the method `print_date` in the class `Date`"
- Since we specify the classname, implementations may be in any file(s) we like

date.cpp

```
#include "date.h"

Date::Date(int year, int month, int day)
    : _year{year}, _month{month}, _day{day} {
    if (1 > month || month > 12) throw std::runtime_error{"Invalid month"};
    if (1 > day   ||   day > 31) throw std::runtime_error{"Invalid day"};
}
void Date::print_date() {
    std::cout << _month << '/' << _day << '/' << _year << std::endl;
}
```

Why .h and .cpp are Useful

Interface / Declarations

complex.h:

```
class Complex {
    double _x, _y;
  public:
    Complex(double x, double y);
    double magnitude();
};
```

Defines

Uses

complex.cpp:

```
#include "complex.h"
#include <cmath>  # sqrt
//definitions:
Complex::Complex(double x, double y)
       : _x{x}, _y{y} { }
double Complex::magnitude() {
    return sqrt(_x*_x + _y*_y);
}
```

test_complex.cpp:

```
#include "complex.h"
include <iostream>

int main() {
    Complex c{3.0, 4.0};
    std::cout << c.magnitude()
              << std::endl;
}
```

Implementations / Definitions

- A header file (here, **complex.h**) defines an interface between user code and implementation code (usually in a library)
- Add the same **#include** declarations in both **.cpp** files (definitions and test_complex)
- .hxx and .cxx sometimes used for .h and .cpp

```
student@cse1325:/media/sf_dev/05$ make complex
g++ --std=c++17 -c test_complex.cpp
g++ --std=c++17 -c complex.cpp
g++ --std=c++17 -o complex test_complex.o complex
student@cse1325:/media/sf_dev/05$ ./complex
5
```

The .h Guard

Interface / Declarations

complex.h:

```
#ifndef __COMPLEX_H
#define __COMPLEX_H

class Complex {
    double _x, _y;
  public:
    Complex(double x, double y);
    double magnitude();
};
#endif
```

"class Complex{ }" can only be compiled *once* per g++ call. These 3 standard preprocessor instructions ("the guard") enforce this. ***Always*** add to your .h files!

"#pragma once" is a simpler but **non-standard** equivalent guard.

C++ added modules and an import statement similar to Java in version 20.

complex.h:

```
#pragma once

class Complex {
    double _x, _y;
  public:
    Complex(double x, double y);
    double magnitude();
};
```

Initialization Lists

- Unlike Java (and every other language I know), C++ calls the default constructor for every field *before* the class constructor begins
  - What if a field type has no default constructor, or we want to use a different constructor?
  - If we do nothing, the compiler will generate an error
- C++ supports *initialization lists*, which specify the fields' constructors to call *before* this class' constructor runs

```
Date::Date(int year, int month, int day)
    : _year{year}, _month{month}, _day{day} {
    if (1 > month || month > 12) throw std::runtime_error{"Invalid month"};
    if (1 > day   ||   day > 31) throw std::runtime_error{"Invalid day"};
}
```

```
Complex::Complex(double x, double y)
       : _x{x}, _y{y} { }
```

Note the curly braces (<u>not</u> parentheses)
We are specifying these fields' constructors in these examples

Initialization List Example

```
class First {
  public:
    First(std::string a_string) : s{a_string} {}
    std::string first_string() {return s;}
  private:
    std::string s;
};

class Second {
  public:
    Second(First f) {
        first = f;
    }
    std::string second_string() {return first.first_string();}
  private:
    First first;
};

int main() {
    First f{"Initialization lists are important!\n"};
    Second s{f};
    std::cout << s.second_string() << std::endl;
}
```

Initialization list specifies how to construct the fields.

NO initialization list (the “Java Way”)

We vote: Will this program:

- ☐ NOT compile?
- ☐ Compile but NOT run?
- ☐ Compile and run correctly?

Initialization Lists

Initialization list specifies how to construct the fields.

```
class First {
  public:
    First(std::string a_string) : s{a_string} {}
    std::string first_string() {return s;}
  private:
    std::string s;
};

class Seco
  public:
    Secon
        f
    }
    std::s
  private
    First
};

int main(
    First
    Secon
    std::
}
```

gram:

un?
orrectly?

```
ricegf@pluto:~/dev/cpp/cse1325-prof/init$ make
Do you know why bad_init.cpp won't compile?
Can you fix it?
--
g++ --std=c++17 -o bad_init bad_init.cpp
bad_init.cpp: In constructor 'Second::Second(First)':
bad_init.cpp:33:23: error: no matching function for call to 'First::First()'
 Second::Second(First f) {
                       ^
bad_init.cpp:14:1: note: candidate: First::First(std::__cxx11::string)
 First::First(std::string a_string) : s{a_string} {}
 ^
bad_init.cpp:14:1: note:   candidate expects 1 argument, 0 provided
bad_init.cpp:5:7: note: candidate: First::First(const First&)
 class First {
       ^
bad_init.cpp:5:7: note:   candidate expects 1 argument, 0 provided
bad_init.cpp:5:7: note: candidate: First::First(First&&)
bad_init.cpp:5:7: note:   candidate expects 1 argument, 0 provided
Makefile:4: recipe for target 'bad_init' failed
make: [bad_init] Error 1 (ignored)
ricegf@pluto:~/dev/cpp/cse1325-prof/init$
```

Constructor Chaining and Default Parameters

- Called “delegated constructors” in C++, a similar syntax permits one constructor to rely on another in the same class
  - This avoids code duplication
  - Sometimes a default parameter is simpler

The default Date constructor delegates to the 3-int constructor

```
Date::Date(int year, Month month, int day)
    : _year{year}, _month{month}, _day{day} {
    if (1 > month || month > 12) throw std::runtime_error{"Invalid month"};
    if (1 > day   ||   day > 31) throw std::runtime_error{"Invalid day"};
}
Date::Date() : Date(1970, Month::January, 1) { }
```

Default parameters may accomplish the same goal

```
class Date {
  public:
    Date(int year=1970, Month month=Month::January, int day=1);
```

Destructors

- Destructors run when the object is deleted
  - Complements constructors – free resources such as heap memory allocated by the constructor
- No parameters and cannot be explicitly invoked
- Default destructor does nothing

```
#include <iostream>
#include <vector>

class Rando {
  public:
    Rando() { // I'm the constructor
        std::cerr << "Constructing v" << std::endl;
        v = new std::vector<int>; // Allocate mem
        for(int i=0; i< 100; ++i)
            v->push_back(rand() % 100);
    }
    ~Rando() { // I'm the destructor!
        std::cerr << "Destructing v" << std::endl;
        delete v;                 // Free mem
    }
    void printv() {
        for(int i : *v) std::cout << i << ' ';
        std::cout << std::endl;
    }
  private: std::vector<int>* v;
};

int main() {
    Rando r;      // Construct a Rando on the stack
    r.printv();   // Print out its vector from heap
}                 // Rando's destructor runs here!
```

Destructors

- Destructors run when the object is deleted
  - Complements constructors – free resources such as
- Default destructor does nothing

We'll discuss **`virtual`** soon, but in C++, always declare destrustors as **`virtual`**!

```
#include <iostream>
#include <vector>

class Rando {
  public:
    Rando() { // I'm the constructor
        std::cerr << "Constructing v" << std::endl;
        v = new std::vector<int>; // Allocate mem
```

```
ricegf@antares:~/dev/202201/19/code_from_slides/cpp_inheritance$ make destructor
g++ --std=c++17 -o destructor destructor.cpp
Now type ./destructor to execute the result

ricegf@antares:~/dev/202201/19/code_from_slides/cpp_inheritance$ ./destructor
Constructing v
83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 62 23 67 35 29 2
 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70 13 26 91 80 56 73 62 70 96 81 5 25 84 2
7 36 5 46 29 13 57 24 95 82 45 14 67 34 64 43 50 87 8 76 78 88 84 3 51 54 99 32 60 76 68 39 12
 26 86 94 39
Destructing v
ricegf@antares:~/dev/202201/19/code_from_slides/cpp_inheritance$
```

```
  private: std::vector<int>* v;
};

int main() {
    Rando r;      // Construct a Rando on the stack
    r.printv();   // Print out its vector from heap
}               // Rando's destructor runs here!
```

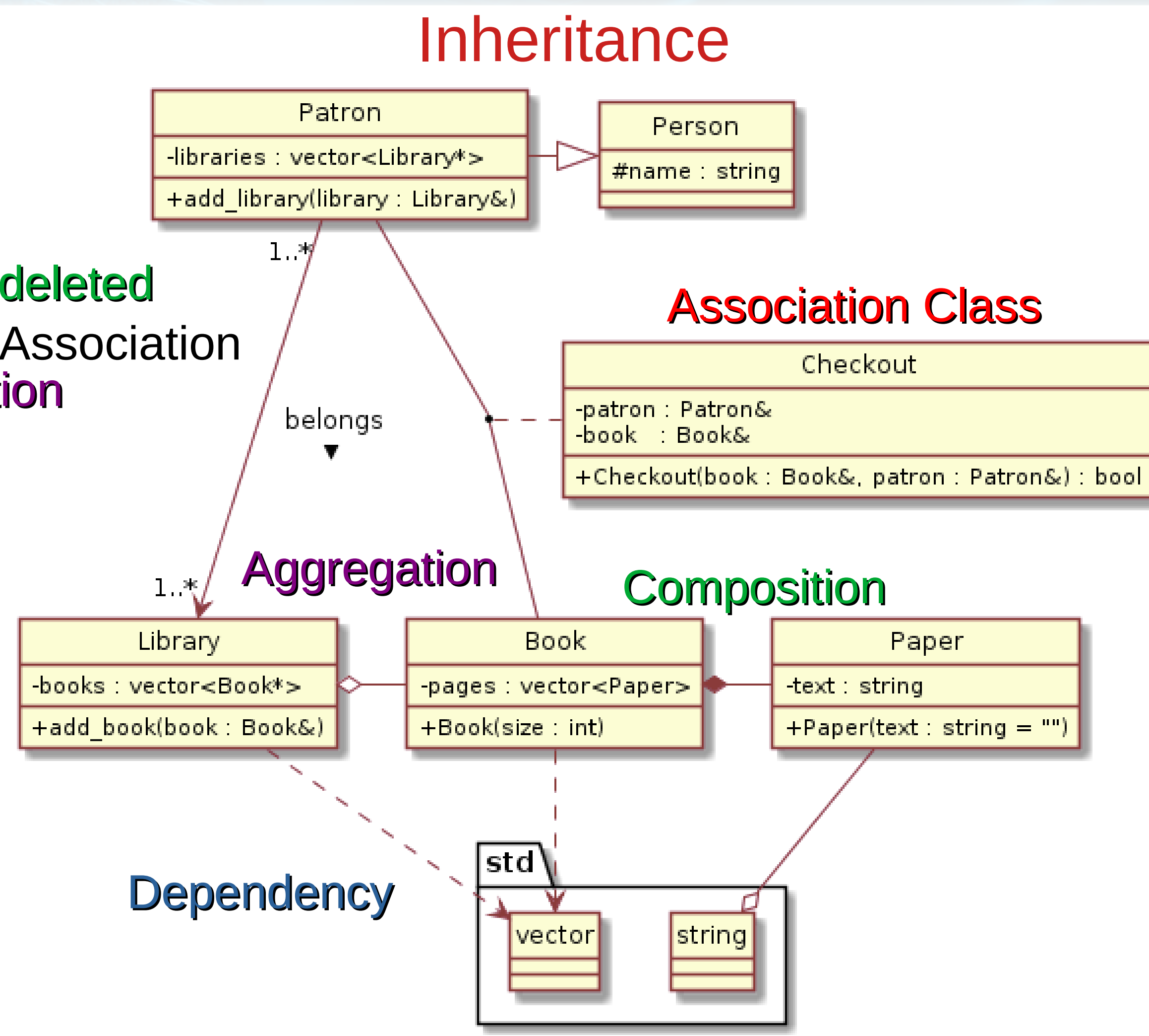
C++ Does Relationships Too!
UML Relationships Summary

Use #include for dependencies

Use values for composition
- Instance the fields in this class
- Delete fields when the compositor is deleted

Use pointers or references for aggregation
- and association classes
- Accept references in add() method
- Never delete the referenced objects

Object-Oriented Programming is as easy as
- Polymorphism
- Inheritance
- Encapsulation



This URL generates the above diagram:

http://www.plantuml.com/plantuml/uml/PLBDRi8m6BldAQnEGkKdrNQjjegDqwGTaFO0KsWZY5gIAWdJDiQxhrz2e6YNG9pF_cp3qdbX_M7VCTSgtGihzgWxuTopzrPj3bw-raQ_gn-9UxPJZKIjRDr9ni8KtjJ62lkD8mF7nfZMeSIdhBsnZo_3TLO137E8flcWvvmEbA2toPITaWxTeqWljd8aiXQzj54a3EMEI9HGsWVwAj3NqZgZIU0EMknfm0t-zVQwOIsyfH7QqGNNQhpd76JijujGVlv4cTAEZQzsicxDmPCmkOzRIiHram26p4Yf6J2_q6xwMFoZJr4Iknl5e3v22Y-__3Kdvk4blq7nX2mBPdjHjqEn0zuG6L4iPiUHCkEKu4G-4aZaQ0RPGr7CCd1U8NvDVP19sHczTQZICF-j47yLbgOC3p4P7PZn68NbaR00mLCKnQMRC3W0LV9vKHfZiH3EJMg2iQNK3Vsb_m00

Inheritance
with C++ Classes

- **Inheritance** – Reuse and extension of fields and method implementations from another class

- The original class is called the **base class (superclass)** (e.g., exception)
- The extended class is called the **derived class (subclass)** (e.g., Bad_area)

"Assets"

The Heir The Ancestor

```
class Bad_area : public exception {
```

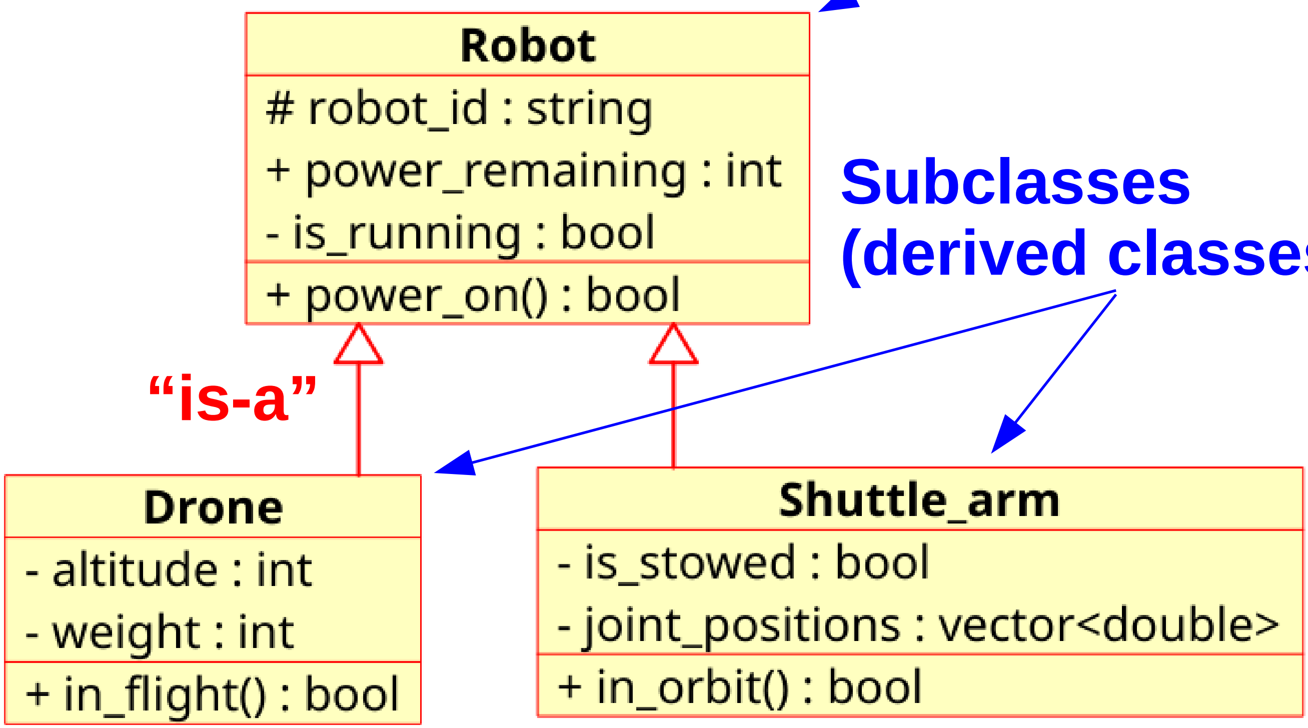
Subclass
("Derived Class")

Superclass
("Base Class")

Terminology

Superclass (base class)

Subclasses (derived classes)




```
class Robot {
```

<u>All 3</u> classes can access robot_id, powerRemaining, and power_on().

ONLY Robot can access is_running (because it's private)

```
class Drone :
    public Robot {
```

```
class ShuttleArm :
    public Robot {
```

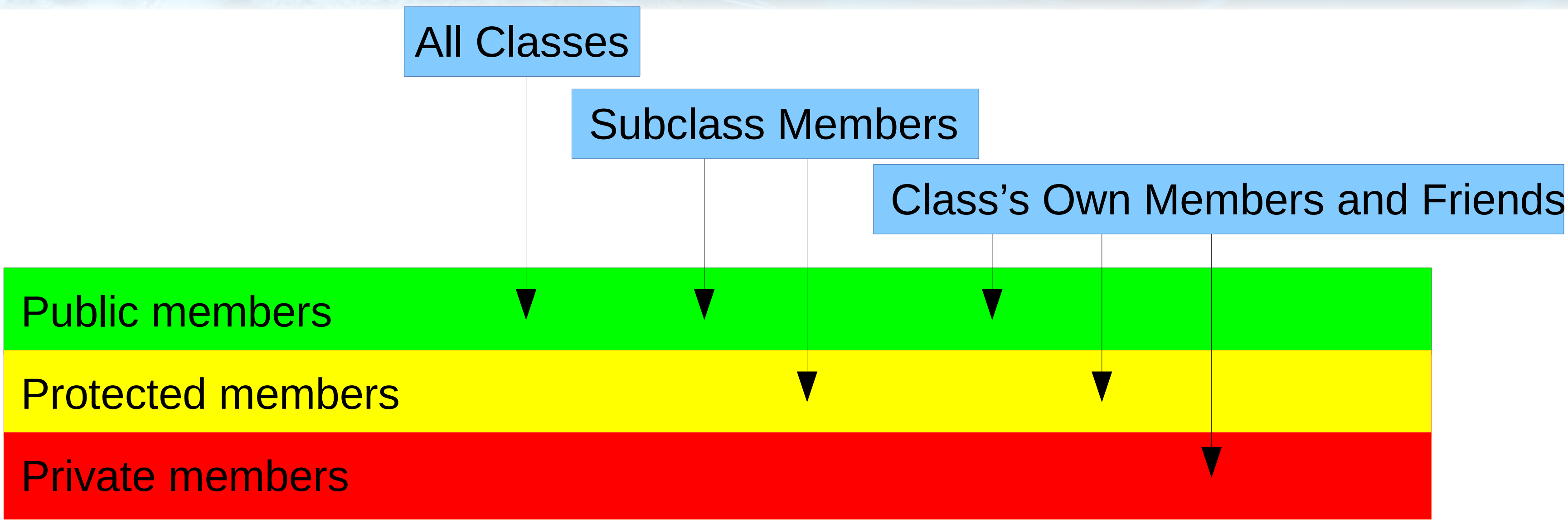
"public" means that all public fields in Robot will be public in ShuttleArm. "private" would make public fields in Robot private in ShuttleArm – similar to instancing Robot as a private field of Shuttle_arm.

Note: C++ does NOT support package-private visibility.
Java does NOT support protected or
private inheritance.

Friends

- A class may declare another class or function as a friend
  - The friend may access its protected and private members
  - Friendship does NOT inherit – a friend of the superclass is NOT a friend of the subclass unless explicitly so declared
- We'll need friendship to override the << and >> operator (among others)
  - Next lecture!

C++ Access Model



- A member (data, function, or type member) or a superclass can be
  - **Public** – **Anyone** can call a public method, access a public constant, and access or modify a public variable
  - **Protected** – **Only class members, subclass members, and friends** can call a protected method, access a protected constant, and access or modify a protected variable
  - **Private** – **Only class members and friends** can call a private method, access a private constant, and access or modify a private variable
  - C++ does NOT support **package-private** visibility, but it does support file pseudo-visibility
- C++ is the only major language to support **friends** – Java does NOT

Back to the Barnyard
Simple Inheritance

```
student@cse1325:/media/sf_dev/07$ make barnyard_simple
g++ --std=c++17 -c barnyard_simple.cpp
g++ --std=c++17 -o barnyard_simple barnyard_simple.o
student@cse1325:/media/sf_dev/07$ ./barnyard_simple
W E L C O M E   T O   T H E   B A R N Y A R D !
Generic critter sound!
Generic critter sound!
Generic critter sound!
Generic critter sound!
Generic critter sound!
Generic critter sound!
```

```
#include <iostream>
#include <vector>
#include <chrono>
#include <thread>

class Critter {
  public:
    Critter(int frequency) : _frequency{frequency}, _timer{0} { }
    ~Critter() { }
    void count() {if (++_timer > _frequency) _timer = 0;}
    void speak() {if (!_timer) std::cout << "Generic critter sound!" << std::endl; }
  protected:
    int _frequency;
    int _timer;
};
int main() {
  std::vector<Critter> critters{Critter{13}, Critter{11}, Critter{7}, Critter{3}};

  std::cout << "W E L C O M E   T O   T H E   B A R N Y A R D !" << std::endl;
  for (int i=0; i<120; ++i) {
    for (Critter& c: critters) { c.count(); c.speak(); }
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
  }
}
```

Timer is just a counter. When expired, the critter makes a sound.
Frequency is how many calls to count() between sounds.

The above idiom pauses the program for 50 milliseconds, or 0.05 seconds

Simple Inheritance

What code needs to change to add barnyard animals (cow, chicken, dog...)?

```
#include <iostream>
#include <vector>
#include <chrono>
#include <thread>

class Critter {
  public:
    Critter(int frequency) : _frequency{frequency}, _timer{0} { }
    ~Critter() { }
    void count() {if (++_timer > _frequency) _timer = 0;}
    void speak() {if (!_timer) std::cout << "Generic critter sound!" << std::endl; }
  protected:
    int _frequency;
    int _timer;
};
int main() {
  std::vector<Critter> critters{Critter{13}, Critter{11}, Critter{7}, Critter{3}};

  std::cout << "W E L C O M E   T O   T H E   B A R N Y A R D !" << std::endl;
  for (int i=0; i<120; ++i) {
    for (Critter& c: critters) { c.count(); c.speak(); }
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
  }
}
```

Using Inheritance

```
#include <iostream>, <vector>, <chrono>, <thread>


class Critter {
  public:
    Critter(int frequency) : _frequency{frequency}, _timer{0} { }
    ~Critter() { }
    void count() {if (++_timer > _frequency) _timer = 0;}
    void speak() {if (!_timer) std::cout << "Generic critter sound!" << std::endl; }
  protected:
    int _frequency;
    int _timer;
};

class Cow : public Critter {
  public:
    Cow(int frequency) : Critter(frequency) { }
    void speak() { if (!_timer) cout << "Moo! Mooooo!" << endl; }
};
class Dog : public Critter {
  public:
    Dog(int frequency) : Critter(frequency) { }
    void speak() { if (!_timer) cout << "Woof! Woof!" << endl; }
};
class Chicken : public Critter {
  public:
    Chicken(int frequency) : Critter(frequency) { }
    void speak() { if (!_timer) cout << "Cluck! Cluck!" << endl; }
};
```

The superclass remains **unchanged**.
The subclasses inherit the implementation of count.
Note: **Constructors *never* inherit!**
But the superclass constructor may be specified in the subclass constructor.

Chaining uses superclass name `Critter` instead of the `super` keyword as in Java.

Reusing Methods with Inheritance

```
int main() {
  std::vector<Dog> dogs {Dog{11}, Dog{9}, Dog{3}};
  std::vector<Cow> cows {Cow{7}, Cow{13}};
  std::vector<Chicken> chickens {Chicken{2}, Chicken{5}};

  std::cout << "W E L C O M E   T O   T H E   B A R N Y A R D !" << std::endl;
  for (int i=0; i<120; ++i) {
    for (auto& c: dogs)     { c.count(); c.speak(); }
    for (auto& c: cows)     { c.count(); c.speak(); }
    for (auto& c: chickens) { c.count(); c.speak(); }
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
  }
}
```

The Dog, Cow, and Chicken classes all used the count() method from Critter.

```
student@cse1325:/media/sf_dev/07$ make barnyard_animals
g++ --std=c++17 -c barnyard_animals.cpp
g++ --std=c++17 -o barnyard_animals barnyard_animals.o
student@cse1325:/media/sf_dev/07$ ./barnyard_animals
W E L C O M E   T O   T H E   B A R N Y A R D !
Cluck! Cluck!
Woof! Woof!
Cluck! Cluck!
Cluck! Cluck!
Woof! Woof!
Moo! Mooooo!
Cluck! Cluck!
```

What Happened to Critter::speak?

```
int main() {
  std::vector<Dog> dogs = {Dog{11}, Dog{9}, Dog{3}};
  std::vector<Cow> cows = {Cow{7}, Cow{13}};
  std::vector<Chicken> chickens = {Chicken{2}, Chicken{5}};

  std::cout << "W E L C O M E   T O   T H E   B A R N Y A R D !" << std::endl;
  for (int i=0; i<120; ++i) {
    for (auto& c: dogs)     { c.count(); c.Critter::speak(); }
    for (auto& c: cows)     { c.count(); c.Critter::speak(); }
    for (auto& c: chickens) { c.count(); c.Critter::speak(); }
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
  }
}
```

Still there – you just have to explicitly *ask* for it!

(Yes, it looks weird. It's C++!)

In Java, you can access only the direct superclass' method. In C++, since you specify the actual class name (Critter), all superclass implementations are accessible.

```
student@cse1325:/media/sf_dev/07$ make barnyard_animals_2
g++ --std=c++17 -c barnyard_animals_2.cpp
g++ --std=c++17 -o barnyard_animals_2 barnyard_animals_2.o
student@cse1325:/media/sf_dev/07$ ./barnyard_animals_2
W E L C O M E   T O   T H E   B A R N Y A R D !
Generic critter sound!
Generic critter sound!
Generic critter sound!
Generic critter sound!
Generic critter sound!
```

Can We Simplify Main?

- It's awkward to keep a separate vector for each subtype
- Why not keep a single vector of type Critter – since, after all, a Dog, Cow, or Chicken "is a" Critter! Right?
- Well, sure you can – but it's harder in C++ than Java!
  - **Your vector must contain *pointers* to Critters** and its derivations, not actual objects
    - Or references – but that's actually *more* awkward
  - **Critter::speak must be declared *virtual***
    - That is, the superclass is required to give *explicit permission* for its subclasses to override its methods
    - If a class contains any virtual method, it should *also* declare a (usually empty) virtual destructor

Preview of Coming Attractions

A Taste of Polymorphism in C++

```
#include <iostream>
#include <vector>
#include <chrono>
#include <thread>

class Critter {
  public:
    Critter(int frequency) : _frequency{frequency}, _timer{0} { }
    virtual ~Critter() { }
    void count() {if (++_timer > _frequency) _timer = 0;}
    virtual void speak() { if (!_timer) std::cout << "Generic critter sound!" << std::endl;
}
  protected:
    int _frequency;
    int _timer;
};
class Cow : public Critter {
  public:
    Cow(int frequency) : Critter(frequency) { }
    void speak() { if (!_timer) std::cout << "Moo! Mooooo!" << std::endl; }
};
class Dog : public Critter {
  public:
    Dog(int frequency) : Critter(frequency) { }
    void speak() { if (!_timer) std::cout << "Woof! Woof!" << std::endl; }
};
class Chicken : public Critter {
  public:
    Chicken(int frequency) : Critter(frequency) { }
    void speak() { if (!_timer) std::cout << "Cluck! Cluck!" << std::endl; }
};
```

A virtual method virtually (ahem) always needs a virtual destructor (unless a superclass already declared one). This ensures that a subclass object's destructor will be called even if invoked from a superclass variable (even though destructors do NOT inherit).

Virtual allows the method of a *sub*type to be accessed via a variable of *this* type, i.e., *polymorphically*

Preview of Coming Attractions

A Taste of Polymorphism

```
int main() {
  std::vector<Critter*> critters = {new Dog{11},new Dog{9},new Dog{3},
                                    new Cow{7}, new Cow{13},
                                    new Chicken{2}, new Chicken{5}};

  std::cout << "W E L C O M E   T O   B A R N Y A R D" << std::endl;
  for (int i=0; i<120; ++i) {
    for (auto c: critters) { c->count(); c->speak(); }
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
  }
}
```

Pointer to Critter

"new" instances Dog et. al. on the heap and returns a *pointer* to the instance

-> accesses a class member via a *pointer*

So yes, we can simplify – but, it's complicated.*

```
student@cse1325:/media/sf_dev/07$ make barnyard_animals_poly
g++ --std=c++17 -c barnyard_animals_poly.cpp
g++ --std=c++17 -o barnyard_animals_poly barnyard_animals_poly.o
student@cse1325:/media/sf_dev/07$ ./barnyard_animals_poly
W E L C O M E   T O   T H E   B A R N Y A R D !
Cluck! Cluck!
Woof! Woof!
Cluck! Cluck!
Cluck! Cluck!
Woof! Woof!
Moo! Mooooo!
Cluck! Cluck!
Woof! Woof!
```

* Yes, the irony is not lost on me. Trust me.

Abstract Methods

(called "Pure Virtual Method" in C++)

- Often, a method in an interface can't be implemented
  - E.g. the data needed isn't "known" until the subclass is implemented
  - We must ensure that a subclass implements that method
  - So we make it a "pure virtual method" by assigning 0 to the declaration
- This is how we define **abstract classes** in C++

```
#include <iostream>

class A {
  public:
    virtual void m() = 0;
};

class B : public A {
  public:
    virtual void x();
};

void B::x() {std::cout << "x of B" << std::endl;}

int main() {
  B b;
  b.x();
}
```

Setting a method "= 0" makes it "pure virtual". This means
(1) we needn't – and indeed *cannot* – define A::m(),
(2) A *cannot* be instanced, and
(3) any subclass of A *must* override and implement m() before it can be instanced.

But B doesn't provide a definition of m() as required by A!
I have a bad feeling about this...

Abstract Methods

(called "Pure Virtual Method" in C++)

- Ofte…
  - E…
  - V…
  - S…
- This…

```
#include…

class A…
  public…
    virt…
};

class B…
  public…
    virt…
};

void B::…

int main…
  B b;
  b.x();
}
```

…(),

…by A!

```
student@cse1325:/media/sf_dev/07$ make pure_virtual_bad
g++ --std=c++17 -c pure_virtual_bad.cpp
pure_virtual_bad.cpp: In function ‘int main()’:
pure_virtual_bad.cpp:16:5: error: cannot declare variable ‘a’ to be of abstract
type ‘A’
   A a;
     ^
pure_virtual_bad.cpp:3:7: note:   because the following virtual functions are pu
re within ‘A’:
 class A {
       ^
pure_virtual_bad.cpp:5:18: note:        virtual void A::m()
     virtual void m() = 0;
                  ^
pure_virtual_bad.cpp:17:5: error: cannot declare variable ‘b’ to be of abstract
type ‘B’
   B b;
     ^
pure_virtual_bad.cpp:8:7: note:   because the following virtual functions are pu
re within ‘B’:
 class B : public A {
       ^
pure_virtual_bad.cpp:5:18: note:        virtual void A::m()
     virtual void m() = 0;
                  ^
Makefile:97: recipe for target 'pure_virtual_bad.o' failed
make: *** [pure_virtual_bad.o] Error 1
student@cse1325:/media/sf_dev/07@
```

Correct Pure Virtual Method

- An abstract class can ONLY be used as a superclass, parameter type, or return type

Incorrect

```
#include <iostream>

class A {
  public:
    virtual void m() = 0;
};
class B : public A {
  public:
    virtual void x();

};
void B::x() {
   std::cout << "x of B" << std::endl;
}



int main() {
  A a;
  B b;
  b.x();
}
```

Correct

```
#include <iostream>

class A {
  public:
    virtual void m() = 0;
};
class B : public A {
  public:
    virtual void x();
    void m() override;
};
void B::x() {
    std::cout << "x of B" << std::endl;
}
void B::m() {
    std::cout << "m of B" << std::endl;
}

int main() {

  B b;
  b.x();
}
```

Correct Pure Virtual Method

- An abstract class can ONLY be used as a superclass, parameter type, or return type

Incorrect

```cpp
#include <iostream>

class A {
  public:
    virtual void m() = 0;
};
class B : public A {
  public:
    virtual void x();

};
void B::x() {
   std::cout << "x of B" << std::endl;
}



int main() {
  A a;
  B b;
  b.x();
}
```

Correct

```cpp
#include <iostream>

class A {
  public:
    virtual void m() = 0;
};
class B : public A {
  public:
    virtual void x();
    void m() override;
};
void B::x() {
    std::cout << "x of B" << std::endl;
}
void B::m() {
    std::cout << "m of B" << std::endl;
}
```

```
student@cse1325:/media/sf_dev/07$ make pure_virtual_fixed
g++ --std=c++17 -c pure_virtual_fixed.cpp
g++ --std=c++17 -o pure_virtual_fixed pure_virtual_fixed.o
student@cse1325:/media/sf_dev/07$ ./pure_virtual_fixed
x of B
student@cse1325:/media/sf_dev/07$
```

Rethinking Critter as Pure Virtual

```
#include <iostream>, <vector>, <chrono>, <thread>

class Critter {
  public:
    Critter(int frequency) : _frequency{frequency}, _timer{frequency} { }
    virtual ~Critter() { }
    void count() {if (++_timer > _frequency) _timer = 0;}
    virtual void speak() = 0;
  protected:
    int _frequency;
    int _timer;
};

class Cow : public Critter {
  public:
    Cow(int frequency) : Critter(frequency) { }
    void speak() override { if (!_timer) std::cout << "Moo! Mooooo!" << std::endl; }
};
class Dog : public Critter {
  public:
    Dog(int frequency) : Critter(frequency) { }
    void speak() override { if (!_timer) std::cout << "Woof! Woof!" << std::endl; }
};
class Chicken : public Critter {
  public:
    Chicken(int frequency) : Critter(frequency) { }
    void speak() override { if (!_timer) std::cout << "Cluck! Cluck!" << std::endl; }
};
```

Remember our Barnyard? Since generic critters don't exist, we should probably make Critter a *pure virtual* method.

Rethinking Critter as Pure Virtual

```
#include <iostream>, <vector>, <chrono>, <thread>

class Critter {
  public:
    Critter(int frequency) : _frequency{frequency}, _timer{frequency} { }
    virt
    void
    virt
  protec
    int
    int
};

class Co
  public
    Cow(
    void                                                                  ll; }
};
class Do
  public
    Dog(
    void speak() override { if (!_timer) std::cout << "Woof! Woof!" << std::endl; }
};
class Chicken : public Critter {
  public:
    Chicken(int frequency) : Critter(frequency) { }
    void speak() override { if (!_timer) std::cout << "Cluck! Cluck!" << std::endl; }
};
```

```
student@cse1325:/media/sf_dev/07$ make barnyard_animals_pure_virtual
g++ --std=c++17 -c barnyard_animals_pure_virtual.cpp
g++ --std=c++17 -o barnyard_animals_pure_virtual barnyard_animals_pure_virtual.o
student@cse1325:/media/sf_dev/07$ ./barnyard_animals_pure_virtual
W E L C O M E   B A C K   T O   T H E   B A R N Y A R D !
Cluck! Cluck!
Woof! Woof!
Cluck! Cluck!
Cluck! Cluck!
Woof! Woof!
Moo! Mooooo!
Cluck! Cluck!
Woof! Woof!
Woof! Woof!
Woof! Woof!
Cluck! Cluck!
```

Now we can instance new cows, dogs, and chickens, but *not* generic critters – just like on a real farm!

All images are public domain: https://www.maxpixel.net/Country-Farm-Border-Collie-Animal-Tyre-Dogs-870301 https://picryl.com/media/chicken-feet-dirt-farm-animals-de219e

Multiple Inheritance

- What if a subclass is derived from more than one superclass?
  - This is called *multiple inheritance*
  - You inherited traits from both your biological mother and father* – *multiple inheritance*
- With multiple inheritance, each superclass's members are laid out in memory after the subclass's members
  - Note that Java does NOT support multiple inheritance for *classes*, although it does for *interfaces*

Multiple Inheritance is a subclass inheriting class members from two or more superclasses.

* Unless you're a clone, in which case – *single inheritance*!

Multiple Inheritance in UML and C++

- In C++, just list multiple comma-separated superclasses

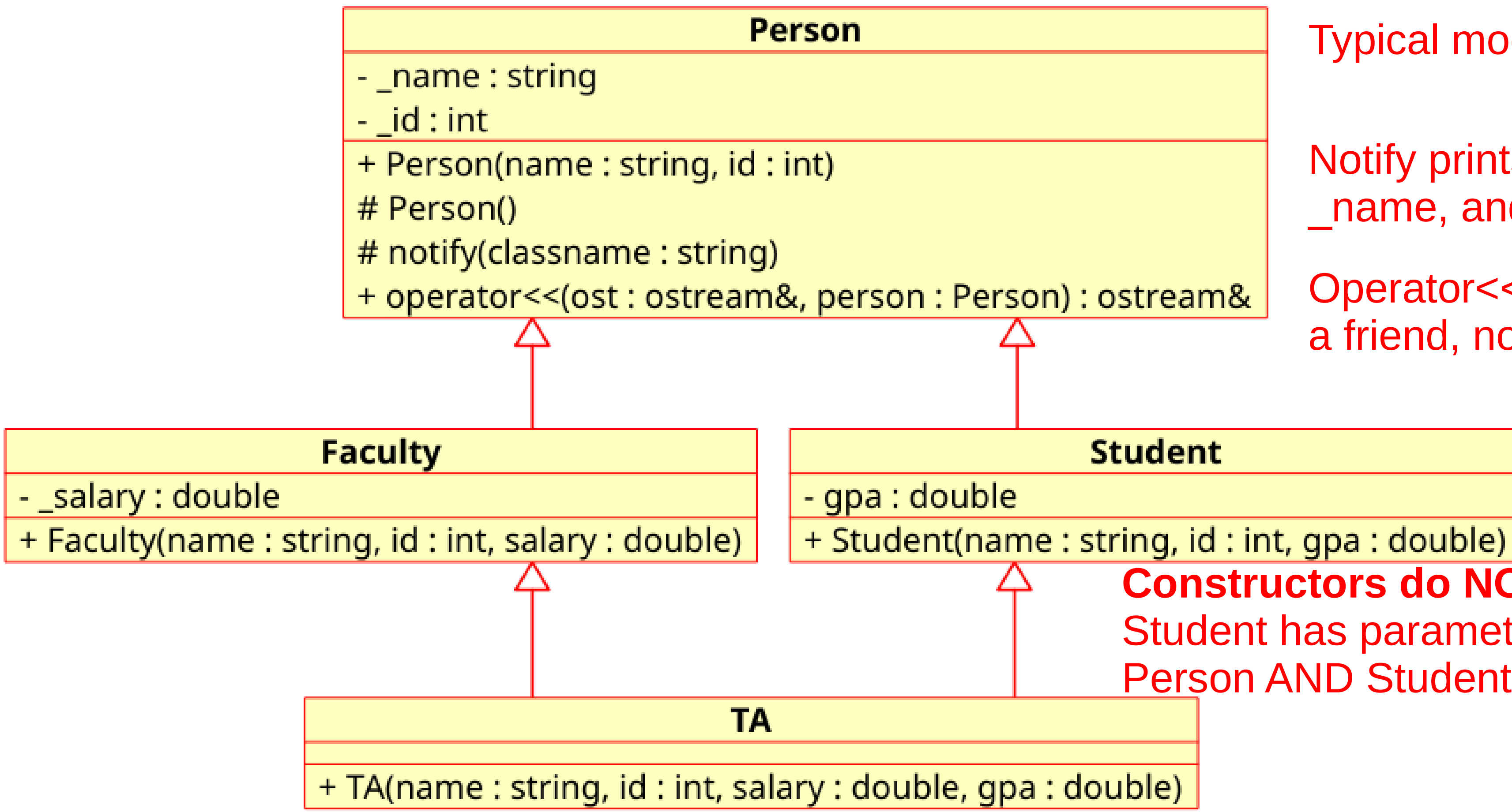
multiple_inheritance.cpp

```
#include <iostream>
class A {
  public:
    A()  { std::cout << "A's constructor called" << std::endl;}
    ~A() { std::cout << "A's destructor called" << std::endl;}
};
class B {
  public:
    B()  {std::cout << "B's constructor called" << std::endl;}
    ~B() {std::cout << "B's destructor called" << std::endl;}
};
class C: public A, public B {
  public:
    C()  {std::cout << "C's constructor called" << std::endl;}
    ~C() {std::cout << "C's destructor called" << std::endl;}
};
int main() {
    C c;
}
```

```
ricegf@antares:~/dev/202201/19/code_from_slides/cpp_inheritance$ ./multi
A's constructor called
B's constructor called
C's constructor called
C's destructor called
B's destructor called
A's destructor called
ricegf@antares:~/dev/202201/19/code_from_slides/cpp_inheritance$
```

Constructors are called in the order listed.
Destructors are called in the reverse order listed.

More Multiple Inheritance



Typical modern Person

Notify prints classname, _name, and _id

Operator<< is (obviously) a friend, not a method

Constructors do NOT inherit!
Student has parameters for Person AND Student constructor.

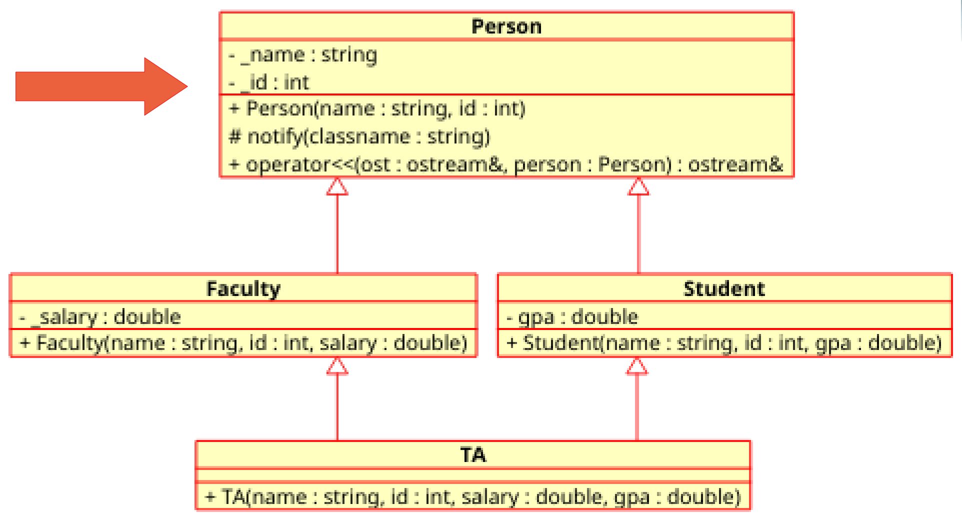
A TA is both Faculty and Student. TA has parameters for each class from which it inherits, back to Person.

Class Person

```
#include <iostream>
#include <ostream>


class Person {
  public:
    Person(std::string name, int id)
        : _name{name}, _id{id}  {
        notify("Person");
    }
    friend std::ostream& operator<<(std::ostream& ost, Person& person);
  protected:
    void notify(std::string classname) {
        std::cout << classname << ' ' << _name << " constructed" << std::endl;
    }
  private:
    std::string _name;
    int _id;
};

std::ostream& operator<<(std::ostream& ost, Person& person) {
    ost << person._name << " (" << person._id << ')';
    return ost;
```



Notify is primarily used to announce execution of a constructor.
It's protected, and thus also available to Faculty, Student, and TA.

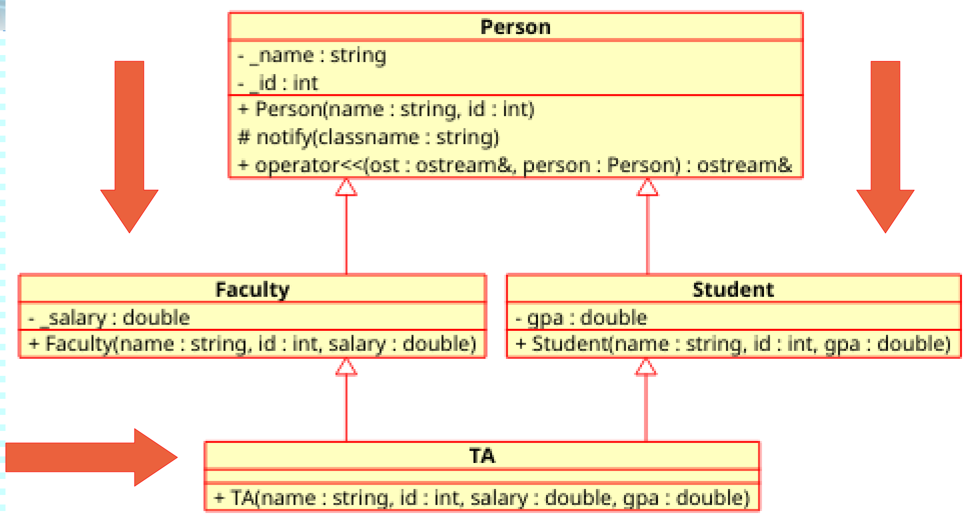
Faculty, Student, and TA instances are also Person instances.
They can also be streamed out with this overload!
(Preview of next week's lecture – don't miss it!)

Classes Faculty, Student, and TA

```
class Faculty : virtual public Person {
    double _salary;
public:
    Faculty(std::string name,
            int id, double salary)
        : Person(name, id), _salary{salary} {
       notify("Faculty");
    }
};

class Student : virtual public Person {
    double _gpa;
public:
    Student(std::string name, int id, double gpa)
        : Person(name, id), _gpa{gpa} {
        notify("Student");
    }
};

class TA : public Faculty, public Student {
public:
    TA(std::string name, int id, double salary, double gpa)
        : Person(name, id), Student(name, id, gpa), Faculty(name, id, salary) {
        notify("TA");
    }
};
```

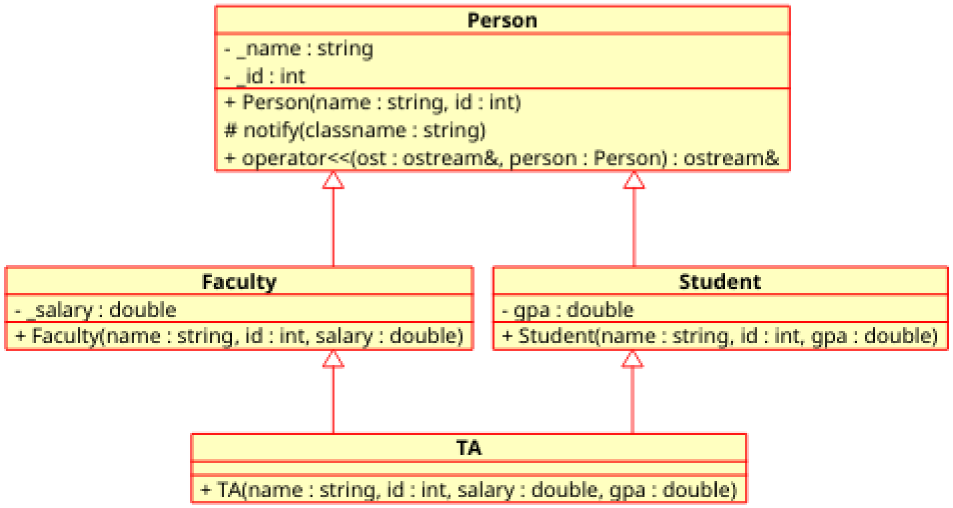


Faculty and Student first delegate to Person, then construct their own fields.

TA first delegates to Person, Student, and Faculty. **The order is irrelevant**: C++ will invoke each ancestor's constructor *exactly once* as specified *here*, in the order declared on the class declaration.

Main

```
student@cse1325:/media/sf_dev/07$ make ta
g++ --std=c++17 -c ta.cpp
g++ --std=c++17 -o ta ta.o
student@cse1325:/media/sf_dev/07$ ./ta
Person Wang Fang constructed
Faculty Wang Fang constructed
Student Wang Fang constructed
TA Wang Fang constructed
Our TA is Wang Fang (100032918)
student@cse1325:/media/sf_dev/07$
```



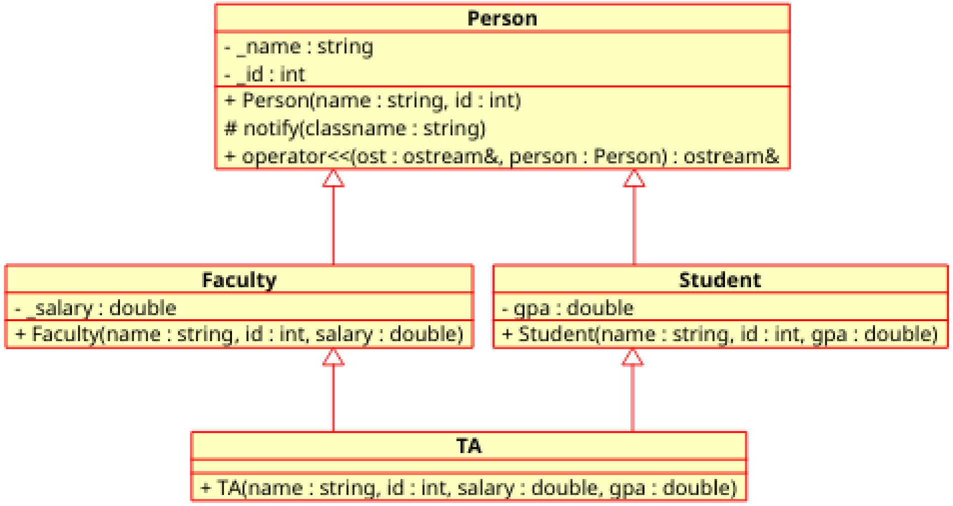

```
int main()  {
    TA ta("Wang Fang", 100032918, 14.50, 3.92);
    std::cout << "Our TA is " << ta << std::endl;
}
```

Note that each class’ constructor is called exactly once *as specified by class TA*. Delegation of a constructor is not “calling” that constructor; it merely specifies how that constructor should be invoked. C++ defines the actual order of invocation.

Need proof?

Feeding Bad Data to Student and Faculty Constructors as a Test

```
student@cse1325:/media/sf_dev/07$ make ta_test
g++ --std=c++17 -c ta_test.cpp
g++ --std=c++17 -o ta_test ta_test.o
student@cse1325:/media/sf_dev/07$ ./ta_test
Person Wang Fang constructed
Faculty Wang Fang constructed
Student Wang Fang constructed
TA Wang Fang constructed
Our TA is Wang Fang (100032918)
student@cse1325:/media/sf_dev/07$
```




```
class TA : public Faculty, public Student  {
public:
    TA(std::string name, int id, double salary, double gpa)
        : Person(name, id), Student("", 0, gpa), Faculty("", 0, salary) {
        notify("TA");
    }
};

int main()  {
    TA ta("Wang Fang", 100032918, 14.50, 3.92);
    std::cout << "Our TA is " << ta << std::endl;
}
```

No difference! It doesn't *matter* that Student delegates to Person; C++ uses ONLY TA's delegation to construct Person as part of TA. If TA didn't delegate to Person, C++ would attempt to call Person{};

Summary

- C++ supports both enum and enum classes
  - But neither supports members
- C++ supports classes similar to Java
  - Specify interface in .h, implementation in .cpp
  - Includes destructors to free resources allocated in the constructor
  - Visibility regions rather than individual keywords per declaration
  - No package-private visibility (but file pseudo-visibility)
- C++ supports inheritance similar to Java
  - Custom exceptions simply inherit from std::exception or its subclasses
  - Polymorphism only works with **virtual** superclass members and **pointers**
  - Include a virtual destructor with any superclass having virtual methods
  - Abstract (pure virtual) methods are set to 0, e.g., **`void m() = 0;`**
  - Multiple inheritance of classes is fully supported