CSE 1325: Object-Oriented Programming

Lecture 06 – Chapter 9

Enum Classes, Operator Overloading, Inheritance, and Use Cases

Mr. George F. Rice

george.rice@uta.edu

Based on material by Bjarne Stroustrup

www.stroustrup.com/Programming

ERB 402

Office Hours:

Tuesday Thursday 11 - 12

Or by appointment

Lecture 5
Quick Review

- A statement that introduces a name with an associated type into a scope is a **declaration**.
  A statement that also fully specifies that entity is a **definition**.
- Why should programs #include header files instead of cpp files? **To avoid circular compilation dependencies**
- True or False: Memory is allocated for a variable when it is declared. **False, memory is allocated when defined**
- True or False: It is an error to include the same file in both the header and the body of a class file. **False**
- What is the purpose of “#ifndef __FILE_H” and “#define __FILE_H” at the top of a header file? **To ensure the file is only processed once per compiler invocation**

Lecture 5
Quick Review

- A special kind of member function that initializes an instance of its class is a **constructor**.
  Direct assignment is a shortcut to assign a value to a class's private variables in a constructor.
  Delegated constructor (or **constructor chaining**) implements one constructor by calling another.
- True or **False**: A default constructor with no parameters is always provided.
  It is only provided if no other constructors are declared
- To modify a parameter, the parameter must be passed by **reference** - otherwise, the method only has a copy of the parameter.
  A **const** reference cannot modify the parameter even though it references to the original data.
- Constexpr specifies that a variable can be evaluated at **compile time**.
- A **namespace** is a named scope, which can be accessed via the **using** keyword or the **membership (::)** operator.
- A **Makefile** builds a C++ program optimally with a simple “make” command.

Overview

- Classes
  - Data validation
  - Structs vs Classes
- Enum Classes
- Operator Overloading
- Inheritance
- More UML
  - Relationships
  - Use Cases
  - Customization

Definitions of OOP

- One perspective is by *capabilities* or *language features*

  Object-Oriented Programming =

  Encapsulation (covered it!)

  \+ **Inheritance** (introduced today!)

  \+ **Polymorphism** (coming soon!)

- Another perspective is by *structure*
  - A structured program is organized around actions and logic
    - "The algorithm's the thing", with apologies to The Bard
  - An object-oriented program is organized around **data** encapsulated in **classes** that also contain related **methods**
    - "The data's the thing"

Classes are Key

- The class is fundamental to object-oriented programming
  - A class directly represents a *concept* in a program
    - A physical item – candy bars in a vending machine, products on Amazon.com, players in a football simulation
    - A logical item – debits and credits on an accounting ledger, positions in 3D space in the solar system, mathematical concepts like complex numbers
  - A class is a *user-defined* type
    - You create variables of it just like ints and doubles
    - You may use it with templates such as vector<>
    - You may define operators for it, commonly
      - = (assignment operator)
      - + - * (binary arithmetic operators)
      - += -= *= (compound assignment operators)
      - == !=
  - A class' functionality may be extended via inheritance *without modifying the class itself*

Members and Member Access

- One way of looking at a class

```
// This class' name is X. Use it as a type: X x;
class X {

    // data members, attributes or fields (they store information,
                                           usually private or protected)

    // methods (they manipulate the information above, often public)

    // friend functions (they are NOT members and not part of the class,
                         but they are granted access to its private fields)
};
```

C Review
Structs

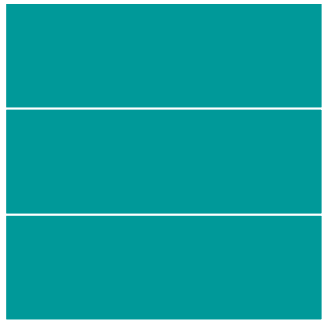
A struct is a class with public data by default and (*by convention only*) no methods

Date:

my_birthday: y

m

d



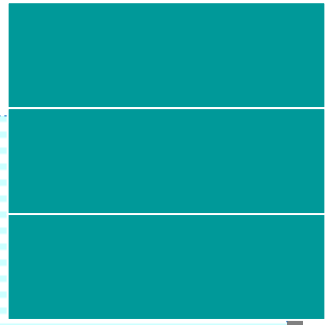
```
// simplest Date (just data)

struct Date {
    int y,m,d;   // year, month, day
};

Date my_birthday;// a Date variable (object)

my_birthday.y = 12;
my_birthday.m = 30;
my_birthday.d = 1950; // oops! (no day 1950 in month 30)

// later in the program, we'll have a problem!
```

Data Validation - ensuring that a program operates on clean, correct and useful data.
Validation Rules – Algorithmically enforceable constraints on the correctness, meaningfulness, and security of input data

C Review
Structs

Helper functions *could* provide code to manipulate structs

- Often in the same namespace
- A very “C” thing to do!



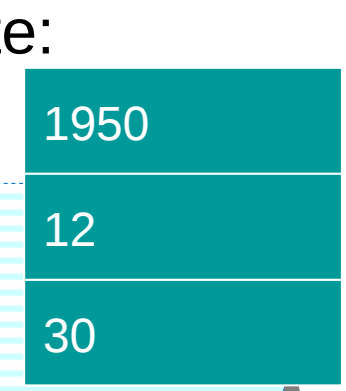

```
// simple Date (with a few helper functions for convenience)
struct Date {
    int y,m,d;   // year, month, day
};

Date my_birthday;// a Date variable (object)


// helper functions:

void init_day(Date& dd, int y, int m, int d); // check valid date and initialize
                                              // Note: this y, m, and d are local

void add_day(Date& dd, int n);    // increase the Date by n days

init_day(my_birthday, 12, 30, 1950); // run time error: no day 1950 in month 30
```

My Take

A helper function is a “poor man’s” method *in my opinion*, and classes with methods should be preferred. I have read arguments that they have value even with classes in reducing the number of formal methods in a class. Not buying it.

C Review
Date as Struct

The C approach...

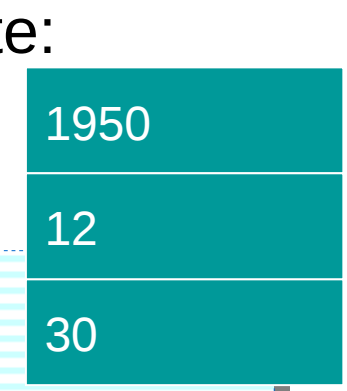



```
// simple Date
//       guarantee initialization with constructor
//       provide some notational convenience
struct Date {
    int y,m,d;                    // year, month, day
    Date(int y, int m, int d);    // constructor: check for valid date and initialize
    void add_day(int n);          // increase the Date by n days
};

// …
Date my_birthday;                  // error: my_birthday not initialized
Date my_birthday {12, 30, 1950};   // oops! Runtime error
Date my_day {1950, 12, 30};        // ok
my_day.add_day(2);                 // January 1, 1951
my_day.m = 14;                     // ouch! (now my_day is a bad date)
```

Public fields require great discipline to use without errors (don’t try this at home – or ESPECIALLY at work!).

Date as Class

The C++ approach...




```
// simple Date (control access)
class Date {
  public:
    Date(int y, int m, int d) : _year{y}, _month{m}, _day{d} { }

    // access methods
    void add_day(int n);        // increase the Date by n days
    int month() { return _month; }   // “getter”
    int day() { return _day; }
    int year() { return _year; }
  private:
    int _year;
    int _month;
    int _day;
};

// …
Date my_birthday {1950, 12, 30};      // ok
cout << my_birthday.month() << endl;  // we can read
my_birthday._month = 14;              // error: Date::_month is private
                                      //      (this is a feature)
```

Classes Help Protect Data Validity

- The notion of a “valid Date” is an important special case of the idea of a valid value
- Design your classes so that fields are guaranteed to be valid
  - Otherwise we have to check for validity all the time
  - Remember that users of your class are exceptionally clever, so program defensively
- Invariant classes, once instanced with valid data, remain valid
  - Variant classes must validate every time data changes
  - Validation may be provided via (sometimes public) methods

Enumeration Classes

- An **enum** (enumeration) **class** is a *managed* user-defined type, specifying its set of values (its enumerators)
  - Unlike simple enums, class enums cannot be interchanged with ints or other class enums

```
enum Color { red, green, blue };                    // plain enum
enum Currency { dollar, pound, yen };               // plain enum
enum class Horse { pony, clydesdale, mustang};      // enum class
enum class Car { mustang, tesla, model_t};          // enum class

int main() {

    Color color = Color::red;  // OK
    int num = color;           // OK but bad (color is not an int!)
    cout << (color == Currency::dollar) << endl;  // OK but REALLY bad

    Horse h = Horse::mustang;
    Car c = Car::mustang;

    int i = h;                 // error – h is not an int, but a Horse
    cout << (c == h) << endl;  // error – c and h are different mustang types
}
```

Enumeration Classes

```
enum Color { red, green, blue };                    // plain enum
enum Currency { dollar, pound, yen };               // plain enum
enum class Horse { pony, clydesdale, mustang};      // enum class
enum class Car { mustang, tesla, model_t};          // enum class
int main() {
    Color color = Color::red;  // OK
    int num = color;           // OK but bad (color is not an int!)
    cout << (color == Currency::dollar) << endl;  // OK but REALLY bad
    Horse h = Horse::mustang;
    Car c = Car::mustang;
    int i = h;                 // error - h is not an int, but a Horse
    cout << (c == h) << endl;  // error - c and h are different mustang types
```

```
ricegf@pluto:~/dev/cpp/201801/06$ make test_enums
g++ --std=c++14 -c test_enums.cpp
test_enums.cpp: In function 'int main()':
test_enums.cpp:13:33: warning: comparison between 'enum Color' and 'enum Currency' [-Wenum-compare]
     cout << (color == Currency::dollar) << endl;  // OK but REALLY bad
                                 ^

test_enums.cpp:18:13: error: cannot convert 'Horse' to 'int' in initialization
     int i = h;                 // error - h is not an int, but a Horse
             ^

test_enums.cpp:19:16: error: no match for 'operator==' (operand types are 'Car' and 'Horse')
     cout << (c == h) << endl;  // error - c and h are different mustang types
              ^

test_enums.cpp:19:16: note: candidate: operator==(Horse, Horse) <built-in>
test_enums.cpp:19:16: note:   no known conversion for argument 1 from 'Car' to 'Horse'
test_enums.cpp:19:16: note: candidate: operator==(Car, Car) <built-in>
test_enums.cpp:19:16: note:   no known conversion for argument 2 from 'Horse' to 'Car'
Makefile:67: recipe for target 'test_enums.o' failed
make: *** [test_enums.o] Error 1
```

Enum Classes Help Validate the Date

```
enum class Month {jan, feb, mar,
                  apr, may, jun,
                  jul, aug, sep,
                  oct, nov, dec};

class Date {
  public:
    Date(int y, Month m, int d) : year{y}, month{m}, day{d} { }
  private:
    int year;
    Month month;
    int day;
};

int main() {
    Date my_birthday(1950, 30, Month::dec);  // error: 2nd argument not a Month
    Date my_birthday(1950, Month::dec, 30);  // OK
}
```

Great! Enum classes are not integers, so they are more accurately type checked. Let's try printing out some dates!

Enum Classes Help Validate the Date

```
#include <iostream>
using namespace std;

enum class Month {jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};

class Date {
  public:
    void set_date(int y, Month m, int d) {year=y; month=m; day=d;}
    void print_date() {cout << month << " " << day << ", " << year << endl;}
  private:
    int year;
    Month month;
    int day;
};

int main() {
    Date my_birthday;
    my_birthday.print_date();
    my_birthday.set_date(1950, Month::dec, 30);  // OK
    my_birthday.print_date();
}
```

Uh oh – enum classes really are NOT just ints!

```
ricegf@pluto:~/dev/cpp/201701/16$ g++ -std=c++11 enum_class_no_constructor.cpp
enum_class_no_constructor.cpp: In member function ‘void Date::print_date()’:
enum_class_no_constructor.cpp:9:32: error: cannot bind ‘std::ostream {aka std::basic_ostream<char>}’ l
value to ‘std::basic_ostream<char>&&’
     void print_date() {cout << month << " " << day << ", " << year << endl;}
                                ^
```

Enum Classes Require String Conversion

(like other classes)

```
enum class Month {jan, feb, mar,
                  apr, may, jun,
                  jul, aug, sep,
                  oct, nov, dec};

string month_to_string(Month m) {
  switch(m) {
    case Month::jan: return "January"; break;
    case Month::feb: return "February"; break;
    case Month::mar: return "March"; break;
    case Month::apr: return "April"; break;
    case Month::may: return "May"; break;
    case Month::jun: return "June"; break;
    case Month::jul: return "July"; break;
    case Month::aug: return "August"; break;
    case Month::sep: return "September"; break;
    case Month::oct: return "October"; break;
    case Month::nov: return "November"; break;
    case Month::dec: return "December"; break;
    default: return "Unknown"; break; // Or thrown a runtime exception
  }
}
```

This is rather awkward and brute force, but it works!

But shouldn't it be a to_string method of Month?

Yep. But regrettably, enum classes don't allow methods! *sigh*

Putting Maps to Use to Convert Enum Classes to Strings

```
enum class Month {jan, feb, mar,
                  apr, may, jun,
                  jul, aug, sep,
                  oct, nov, dec};

string month_to_string(Month m) {
  map<Month, string> m_to_str =
  {
      {Month::jan, "January"},
      {Month::feb, "February"},
      {Month::mar, "March"},
      {Month::apr, "April"},
      {Month::may, "May"},
      {Month::jun, "June"},
      {Month::jul, "July"},
      {Month::aug, "August"},
      {Month::sep, "September"},
      {Month::oct, "October"},
      {Month::nov, "November"},
      {Month::dec, "December"},
  };
  return m_to_str[m];
}
```

Map is just like vector, except the subscript can be *any type* – in this case, our enum class.

Map makes make month_to_string less wordy and very tabular, with little extra punctuation. But we *still* have a “helper function”. :-(

Instead, we could define the << operator for our enum class and just cout << month! We’ll get to that improvement shortly.

With much effort and arcane library calls, you *could* cast the enum class into an integer. If you do, then I must ask: *Why were you using an enum class to begin with?*

Now We Can Print Dates
(Crudely)

```
#include <iostream>
using namespace std;

// enum class code goes here

class Date {
  public:
    Date(int y, Month m, int d) : year{y}, month{m}, day{d} { }
    void print_date() {cout << month_to_string(month) << " " <<
                          day << ", " << year << endl;}
  private:
    int year;
    Month month;
    int day;
};

int main() {
    Date my_birthday{1950, Month::dec, 30};  // OK
    my_birthday.print_date();
}
```

```
ricegf@pluto:~/dev/cpp/201801/06$ g++ --std=c++11 enum_class_io.cpp month_alt.cpp
ricegf@pluto:~/dev/cpp/201801/06$ ./a.out
December 30, 1950
ricegf@pluto:~/dev/cpp/201801/06$
```

Operator Overloading

- We can define the “<<” operator for our Month and Date classes via Operator Overloading

Operator Overloading - Providing a user-defined meaning to a pre-defined operator (e.g., +, ==, <<) for a user-defined type (class).

- Most C++ operators can be overloaded
  - The key is to know the method or function signature, e.g., the parameter types and return values
  - Let’s teach Date (and Month) from earlier in this lecture about streams

Date / Month Streams

```
class Date {
  public:
    Date(int y, Month m, int d) : year{y}, month{m}, day{d} { }
    Date() : Date(1970, Month::jan, 1) { }
    friend ostream& operator<<(ostream& os, const Date& date);
  private:
    int year;
    Month month;
    int day;
};
```

An ostream is an “output stream”. Think cout, our most famous ostream!

Here we’re streaming an enum class...

```
ostream& operator<<(ostream& os, const Month& month) {
  os << month_to_string(month);
  return os;
}
```

...and here we’re streaming our Date class

```
ostream& operator<<(ostream& os, const Date& date) {
  os << date.month << " " << date.day << ", " << date.year;
  return os;
}
```

Operator<< is Date’s *friend*. That simply means this function can access Date’s private data – day, month, and year.

This really cleans up our main code!

```
int main() {
  Date unix_epoch;  // or unix_epoch{}
  cout << unix_epoch << endl;
  Date my_birthday{1950, Month::dec, 30};
  cout << my_birthday << endl;
}
```

Date / Month Streams

```
class Date {
  public:
    Date(int y, Month m, int d) : year{y}, month{m}, day{d} { }
    Date() : Date(1970, Month::jan, 1) { }
    friend ostream& operator<<(ostream& os, const Date& date);
  private:
    int year;
    Month month;
    int day;
};

ostream&
  os << m
  return
}

ostream&
  os << d
  return
}
```

An ostream is an “output stream”. Think cout, our most famous ostream!

```
ricegf@pluto:~/dev/cpp/201801/06$ make enum_class_streams
g++ --std=c++14 -c enum_class_streams.cpp
g++ --std=c++14 -c date.cpp
g++ --std=c++14 -c month.cpp
g++ --std=c++14 -o enum_class_streams enum_class_streams.o date.o month.o
ricegf@pluto:~/dev/cpp/201801/06$ ./enum_class_streams
January 1, 1970
December 30, 1950
ricegf@pluto:~/dev/cpp/201801/06$
```

is Date’s simply function Date’s – day, year.

This really cleans up our main code!

```
int main() {
  Date unix_epoch;  // or unix_epoch{}
  cout << unix_epoch << endl;
  Date my_birthday{1950, Month::dec, 30};
  cout << my_birthday << endl;
}
```

Pre- and Post-Increment

- You can overload other operators, too

```
Month& operator++(Month& m) {   // Pre-increment, e.g., ++m
  switch(m) {
    case Month::jan: m = Month::feb; break;
    case Month::feb: m = Month::mar; break;
    case Month::mar: m = Month::apr; break;
    case Month::apr: m = Month::may; break;
    case Month::may: m = Month::jun; break;
    case Month::jun: m = Month::jul; break;
    case Month::jul: m = Month::aug; break;
    case Month::aug: m = Month::sep; break;
    case Month::sep: m = Month::oct; break;
    case Month::oct: m = Month::nov; break;
    case Month::nov: m = Month::dec; break;
    case Month::dec: m = Month::jan; break;
  }
  return m;
}

Month operator++(Month& m, int) {  // Post-increment, e.g., m++
  Month result{m};
  ++m;
  return result;
}
```

Since enum classes aren't ints, incrementing is a bit of a pain.

Or use a map (left as an exercise for the above-average student).

Pre- and Post-Increment

```
#include "date.h"
#include <iostream>
using namespace std;

int main() {
    // Post-increment in a 3-term for
    Month m{Month::oct};
    for (int i = 0; i<6; ++i) cout << m++ << endl;
    cout << endl;

    // Wait - use the month as the "counter"!
    for (Month m=Month::oct; m != Month::apr ; ++m) cout << m << endl;
}
```

```
ricegf@pluto:~/dev/cpp/201801/06$ g++ --std=c++14 enum_class_increment.cpp month.cpp date.cpp
ricegf@pluto:~/dev/cpp/201801/06$ ./a.out
October
November
December
January
February
March

October
November
December
January
February
March
ricegf@pluto:~/dev/cpp/201801/06$
```

Prefix or Postfix Increment As a Method or Function

- C++ allows most operators to be defined as a method OR a function
  - We usually prefer the method approach
  - But... enum classes have no methods :-(

Increment/decrement operators

Increment/decrement operators increment or decrement the value of the object.

Operator name	Syntax	Overloadable	Prototype examples (for `class T`)	
			Inside class definition	Outside class definition
pre-increment	`++a`	Yes	`T& T::operator++();`	`T& operator++(T& a);`
pre-decrement	`--a`	Yes	`T& T::operator--();`	`T& operator--(T& a);`
post-increment	`a++`	Yes	`T T::operator++(int);`	`T operator++(T& a, int);`
post-decrement	`a--`	Yes	`T T::operator--(int);`	`T operator--(T& a, int);`

http://en.cppreference.com/w/cpp/language/operator_incdec

Another Example: A Box

More Detail This Time

- Let's write a Box class with 2 operators
  - + adds the boxes' linear dimensions
  - << streams (h,w,d) to represent the box

```
// Test the Box class
int main( ) {
   Box box1(6.0, 7.0, 5.0);     // Declare box1 of type Box
   Box box2(12.0, 13.0, 10.0);  // Declare box2 of type Box
   Box box3 = box1 + box2;

   cout << "Box 1: " << box1 << " volume " << box1.getVolume() <<endl;
   cout << "Box 2: " << box2 << " volume " << box2.getVolume() <<endl;
   cout << "Box 3: " << box3 << " volume " << box3.getVolume() <<endl;

   return 0;
}
```

Example: The Box Interface

```
#include <iostream>
using namespace std;

class Box {
  public:
     Box (double len, double bre, double hei)
        : length(len), breadth(bre), height(hei) { }

     double getVolume(void) {
        return length * breadth * height;
     }

     // Overload + operator to add two Box objects.
     Box operator+(const Box& b);

     // Overload << operator to stream a string representation
     friend ostream& operator<<(ostream& os, const Box& b);

  private:
     double length;      // Length of a box
     double breadth;     // Breadth of a box
     double height;      // Height of a box
};
```

As a Method

The operator+ is a member method. “This” is the left operand, b is the right operand, and we return a new Box representing the sum.

As a Function

Here the operator<< is NOT a member method, but is a “friend” of this class* – hence, it can access Box private variables. The stream os is the left operand, b is the right operand, and we return a reference to the updated stream.

* This is necessary for operator<< and operator>>. Can you see why?

Example: The Box Implementation

```
Box Box::operator+(const Box& b) {
   return Box(length + b.length, breadth + b.breadth, height + b.height);
}
```

For operator+, we simply instance a new Box object, passing the constructor the sum of our and the right operand's lengths, breadths, and heights, respectively.

```
ostream& operator<<(ostream& os, const Box& b) {
   os << '(' << b.length << ',' << b.breadth << ',' << b.height << ')';
   return os;
}
```

Note the LACK of “Box::” preceding operator<< - this is NOT a member method, but rather a “friend” of the Box class. Thus, operator<< must be declared in the Box interface with the keyword “friend” (granting permission to access its private variables), but it is NOT part of the Box implementation.

The implementation is straightforward. We simply use the same syntax as we would use with cout, but with os instead. Then we return os.

```
ricegf@pluto:~/dev/cpp/08$ g++ box.cpp
ricegf@pluto:~/dev/cpp/08$ ./a.out
Box 1: (6,7,5) volume 210
Box 2: (12,13,10) volume 1560
Box 3: (18,20,15) volume 5400
ricegf@pluto:~/dev/cpp/08$
```

Operator Overloading

- You can define only existing operators
  - You can't create your own custom operator such as $$ or @
- You can define operators only with the usual number of operands
  - *E.g.*, no unary **<=** (less than or equal) and no binary **!** (not)
- Overloaded operators must have *at least* one user-defined type as an operand
  - **int operator+(int,int);** *//* error: you can't overload built-in +
  - **Vector operator+(const Vector&, const Vector &);** *//* ok
- Advice (not language rule):
  - Overload operators only with their conventional meaning
  - **+** should be addition, ***** be multiplication, **[]** be access, **()** be call, etc.
  - You must determine what is "conventional" with the classes you define
- Advice (not language rule):
  - Don't overload unless you really have to

Permissible Operator Overloading

- The following operators **<u>can</u>** be overloaded:
  - + - * / % ^
  - & | ~ ! , =
  - < > <= >= ++ --
  - << >> == != && ||
  - += -= /= %= ^= &=
  - |= *= <<= >>= [ ] ( )
  - → ->* new new [ ] delete delete [ ]
- The following operators **<u>cannot</u>** be overloaded:

  :: .* . ?:

Inheritance
with People

- When you inherit from an ancestor, you acquire (many of) their assets.
  - Some may be redirected by a will
- When a class inherits from an ancestor class, it acquires (many of) its methods and fields.
  - Some may be redirected by keyword directives

"Assets"

The Heir

The Ancestor

```
class Bad_area : public exception {
class View     : public DrawingArea {
```

Inheritance
with Classes

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

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

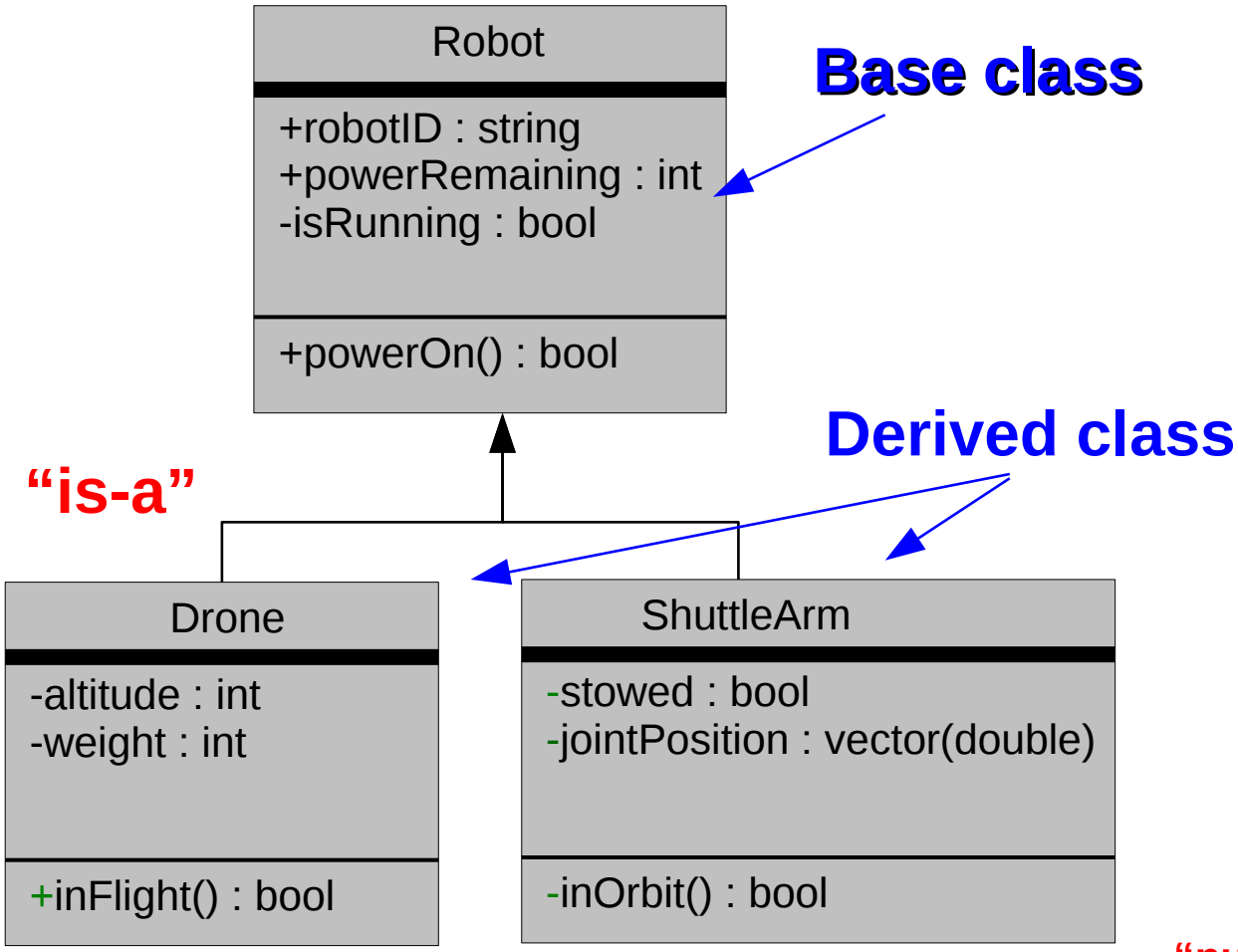
"Assets"

The Heir — The Ancestor

```
class Bad_area : public exception {
class View     : public DrawingArea {
```

Derived Class — **Base Class**

Terminology




```
class Robot {
```

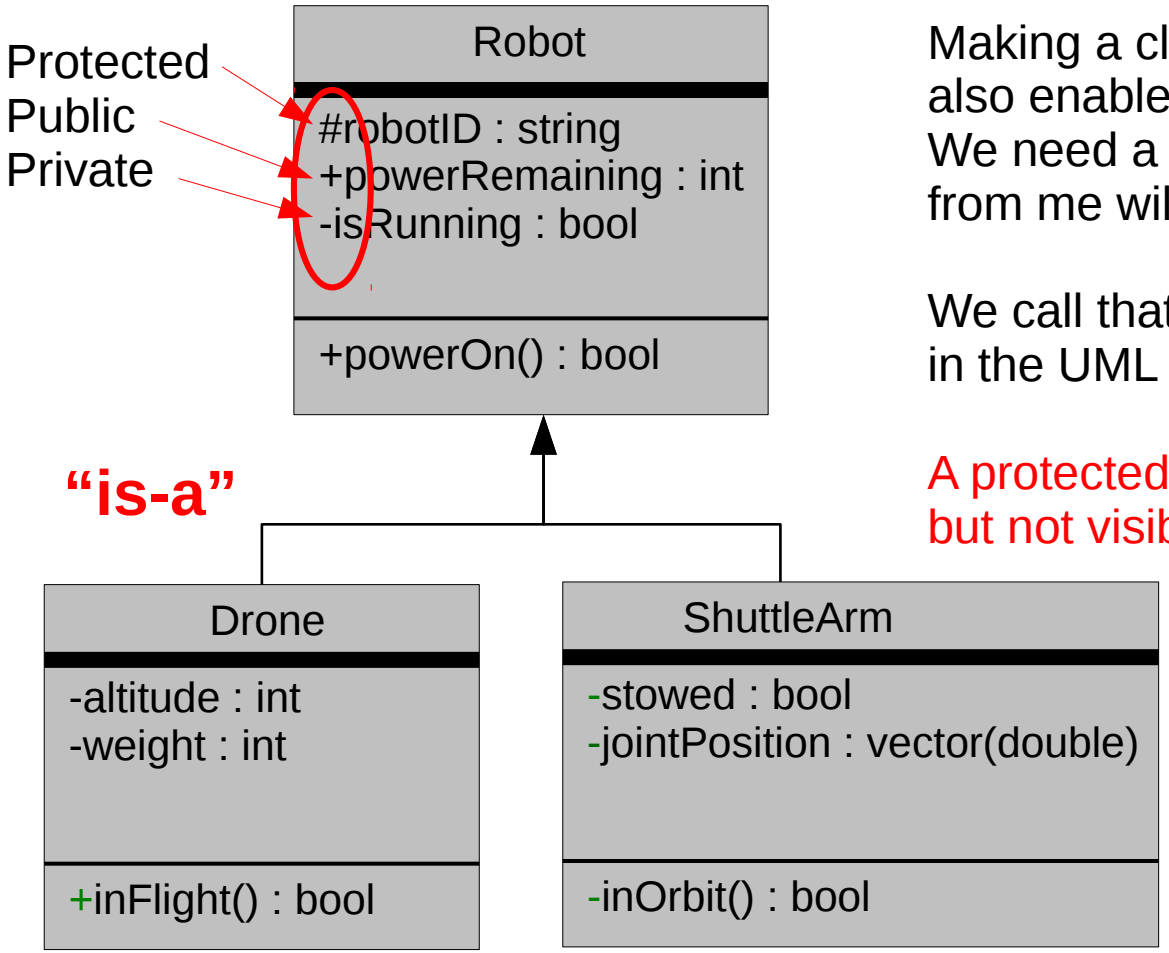
All 3 have robotID, isRunning, powerRemaining, and powerOn().

ONLY Robot has isRunning (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.

Protected Class Members

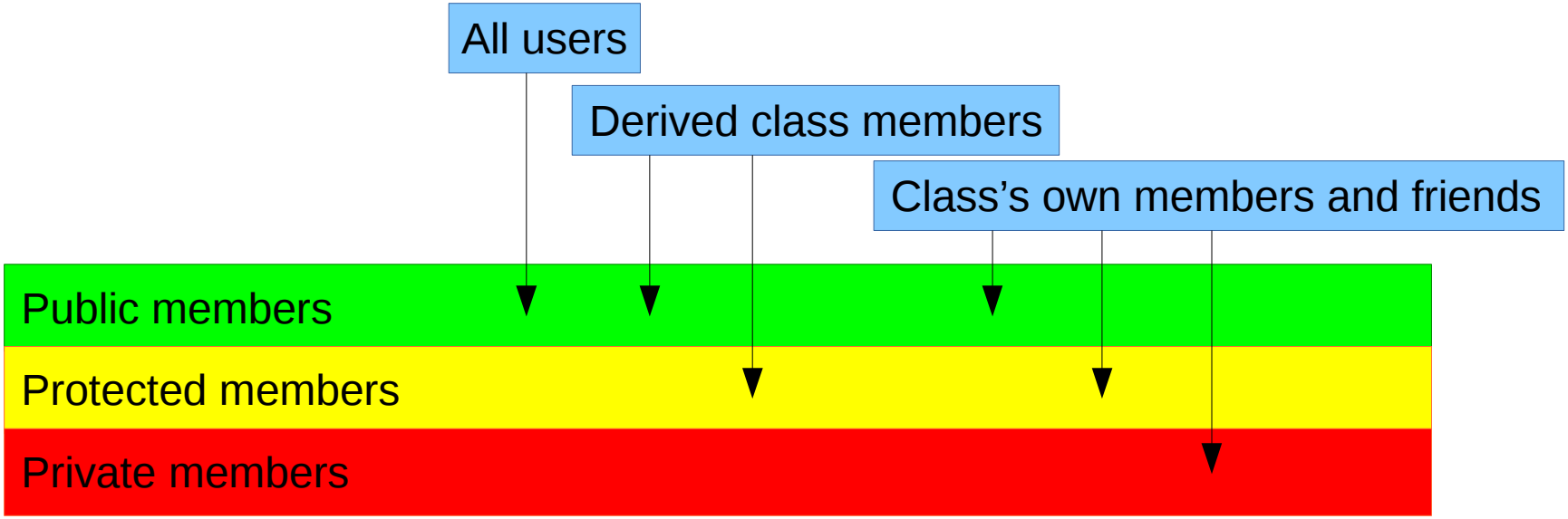


Making a class member public so it can be inherited also enables it to be accessed from main() et. al. We need a middle ground - "only classed derived from me will have access".

We call that middle ground "protected", represented in the UML with '#'.

A protected member is inherited by derived classes, but not visible outside the class hierarchy (except by friends, of course).

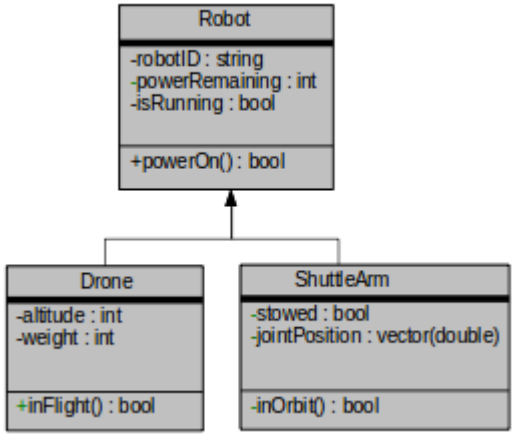
Access model

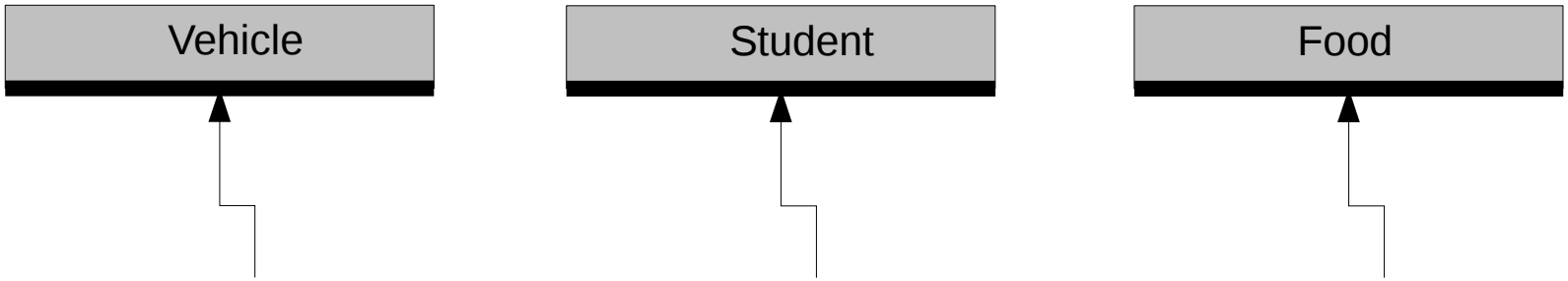


- A member (data, function, or type member) or a base can be
  - Private, protected, or public

Specify a Class Hierarchy

- A class hierarchy defines the inheritance relationships between classes
- EXERCISE: Define class hierarchies based on these base classes





Benefits of Inheritance

- Interface inheritance
  - A function expecting a shape (a **Shape&**) can accept any object of a class derived from Shape.
  - Simplifies use
    - sometimes dramatically
  - We can add classes derived from Shape to a program without rewriting user code
    - Adding without touching old code is one of the “holy grails” of programming
- Implementation inheritance
  - Simplifies implementation of derived classes
    - Common functionality can be provided in one place
    - Changes can be done in one place and have universal effect
      - Another “holy grail”

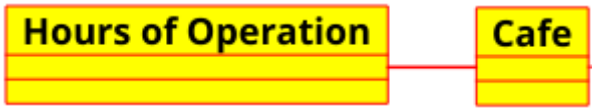
Classes Share Other Relationships

- A class may inherit the public and protected members of one or more other classes
- A class may include, or be built from, other classes
  - Composition is actually a different approach to inheritance, as we'll see later
- A class may have a dependency on a class
- A class may have a more general association with another class

Relationships are more easily understood in the UML

UML Class Relationships
Association

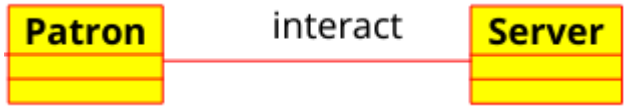
- A line simply indicates a general association
  - One class has a relationship to another class



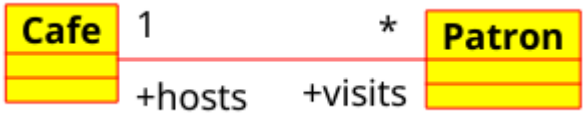
General

Directional

  - The association can be named



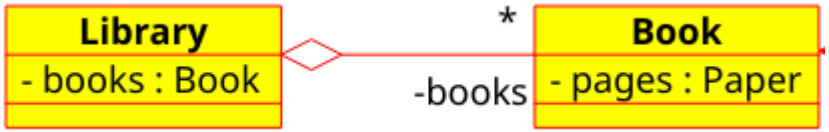
  - Each class may include multiplicity and role



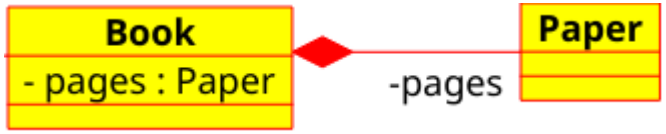
UML Class Relationships

Aggregation and Composition

- Aggregation shows a class being aggregated or constructed of one or more other classes
  - The library includes many books



- Composition additionally shows that the existence of the aggregated classes depends on the aggregate
  - Books are composed of many pages



UML Class Relationships

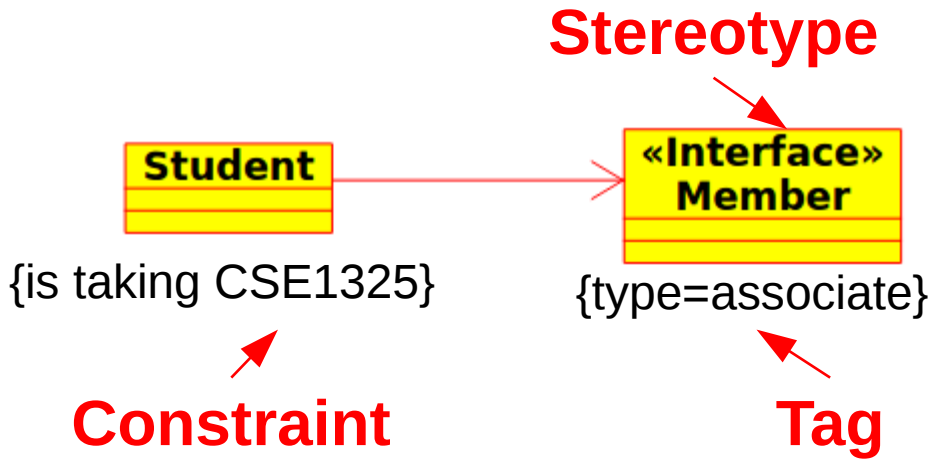
Inheritance and Dependency

- Inheritance shows an “is a” relationship
  - A duck “is a” bird

- Dependency shows that one class depends in some way on another
  - Mainwin depends on the Gtkmm package

Extending the UML

- UML offers 3 distinct ways to extend its notation
  - **Stereotype** – guillemets-enclosed specialization, e.g., «interface» may represent a C++ class declaration (a .h file) rather than a definition (a .cpp file)
  - **Tag** – curly-brace-enclosed assignment of value to tag name, which e.g., may prove useful to identify attributes of a class or control code generation
  - **Constraint** – curly-brace-enclosed Boolean expression that adds a condition, restriction, or assertion on the class



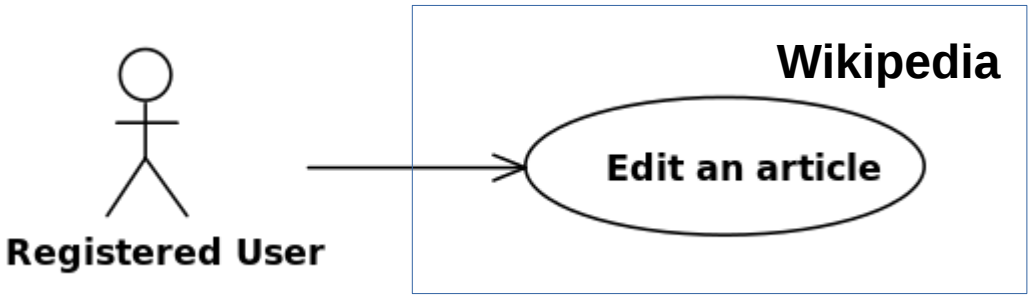
A given tool may specify certain semantic interpretations it will apply to specific extensions, but the UML spec does NOT limit what stereotypes, tags, or constraints you may create for your diagrams.

YOU decide what these mean!

Modeling the Requirements: The Use Case Diagram

- A use case is a series of related interactions that enable an actor (e.g., a user) to achieve a goal.
  - Thus, a use case is always from the *actor's* perspective
  - A use case may be specified graphically using other UML diagrams and / or textually using a form
- A use case diagram graphically depicts the required use cases and their relationship to actors (users) and each other within a system
  - The diagram is read “<Actor> uses <System> to <Use Case>”

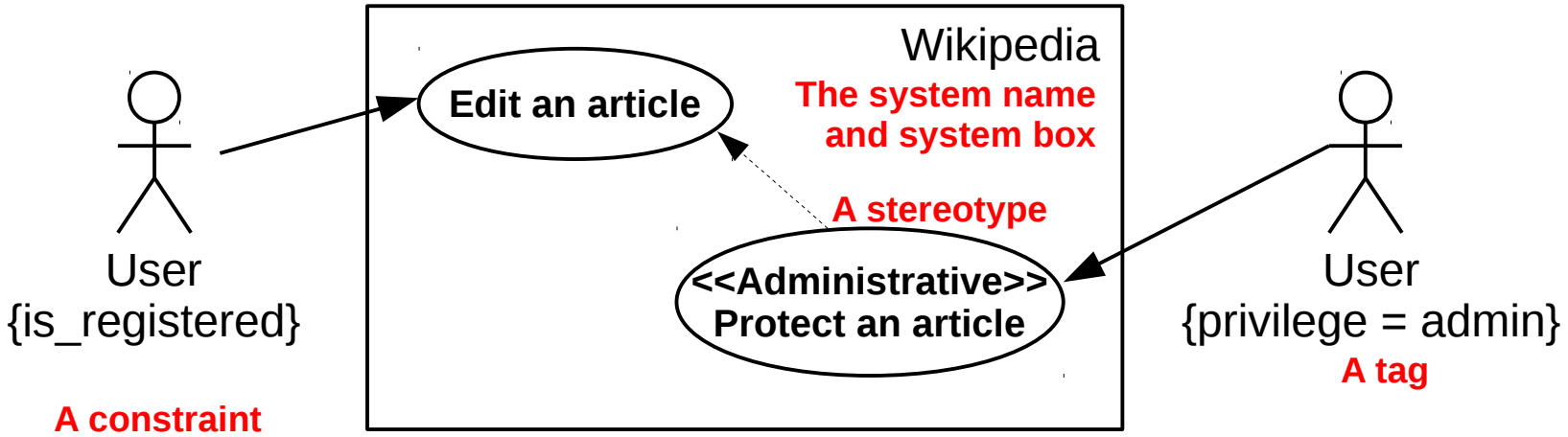
Read this diagram as
“Registered User uses Wikipedia to Edit an article.”



Use case diagram by Softzen and released as Public Domain

Use Cases are *Classes*

- Though drawn as ovals and stick figures, Use Cases and Actors are simply classes
  - You can use the same relationships as with classes
    - Include, extend, derive, ...
  - You can <<stereotype>>, {tag=value}, and {is_constrained} them
- Use cases help model *any* requirements



Documenting a Use Case

- Each use case may be documented using one or more of the following
  - Text and pictures, often from a template
  - UML Sequence diagram
  - UML Statechart diagram
  - UML Activity diagram
- We'll cover the Activity and Sequence diagrams next week, and the Statechart diagram later this year

Use Case: Edit an article

Primary Actor: Member *(Registered User)*

Scope: a Wiki system

Level: ! *(User goal or sea level)*

Brief: *(equivalent to a user story or an epic)*

The member edits any part (the entire article or just a section) of an article he/she is reading. Preview and changes comparison are allowed during the editing.

Stakeholders

...

Postconditions

Minimal Guarantees:

Success Guarantees:

- The article is saved and an updated view is shown.
- An edit record for the article is created by the system, so watchers of the article can be informed of the update later.

Preconditions:

The article with editing enabled is presented to the member.

Triggers:

The member invokes an edit request (for the full article or just one section) on the article.

Basic flow:

1. The system provides a new editor area/box filled with all the article's relevant content with an informative edit summary for the member to edit. If the member just wants to edit a section of the article, only the original content of the section is shown, with the section title automatically filled out in the edit summary.
2. The member modifies the article's content till satisfied.
3. The member fills out the edit summary, tells the system if he/she wants to watch this article, and submits the edit.
4. The system saves the article, logs the edit event and finishes any necessary post processing.
5. The system presents the updated view of the article to the member.

Extensions:

2-3.

a. Show preview:

1. The member selects *Show preview* which submits the modified content.
2. The system reruns step 1 with addition of the rendered updated content for preview, and informs the member that his/her edits have not been saved yet, then continues.

b. Show changes:

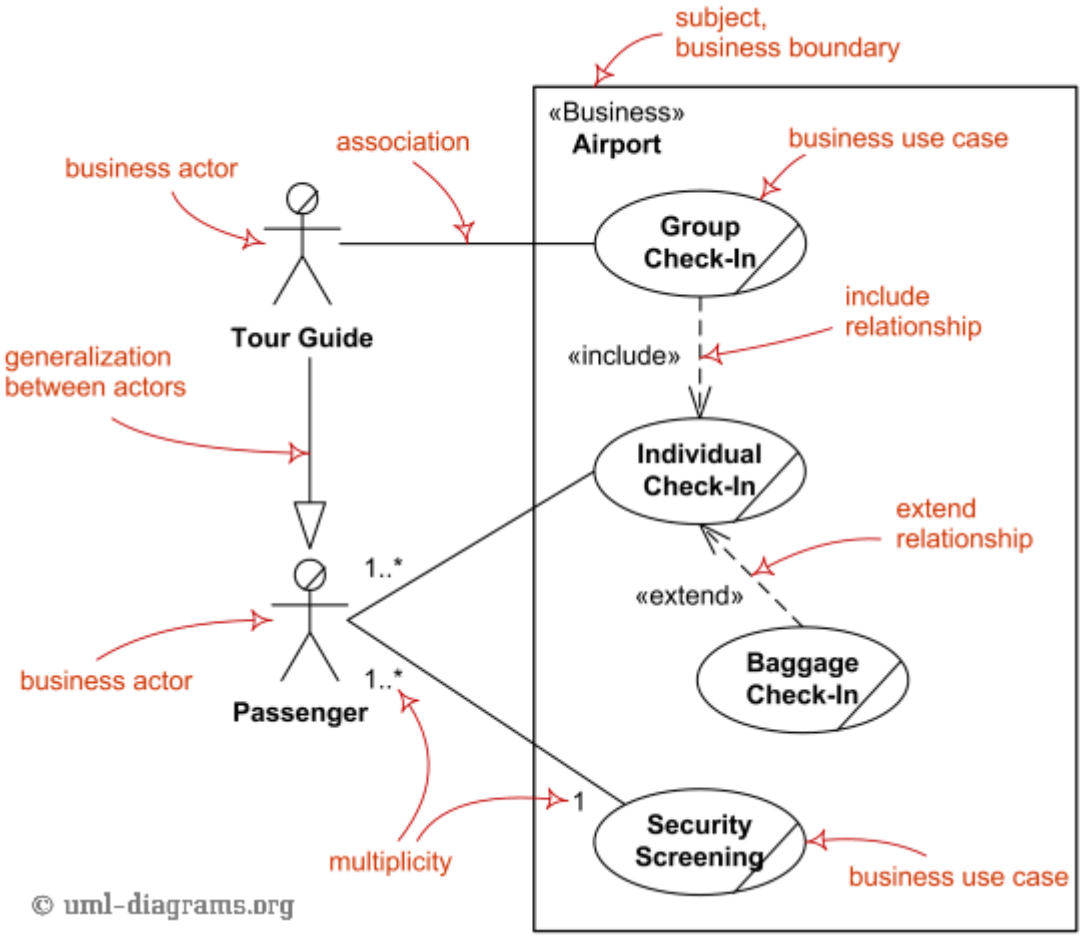
1. The member selects *Show changes* which submits the modified content.
2. The system reruns step 1 with addition of showing the results of comparing the differences between the current edits by the member and the most recent saved version of the article, then continues.

c. Cancel the edit:

1. The member selects *Cancel*.
2. The system discards any change the member has made, then goes to step 5.

4a. Timeout:

Basic Graphical Elements of a UML Use Case Diagram

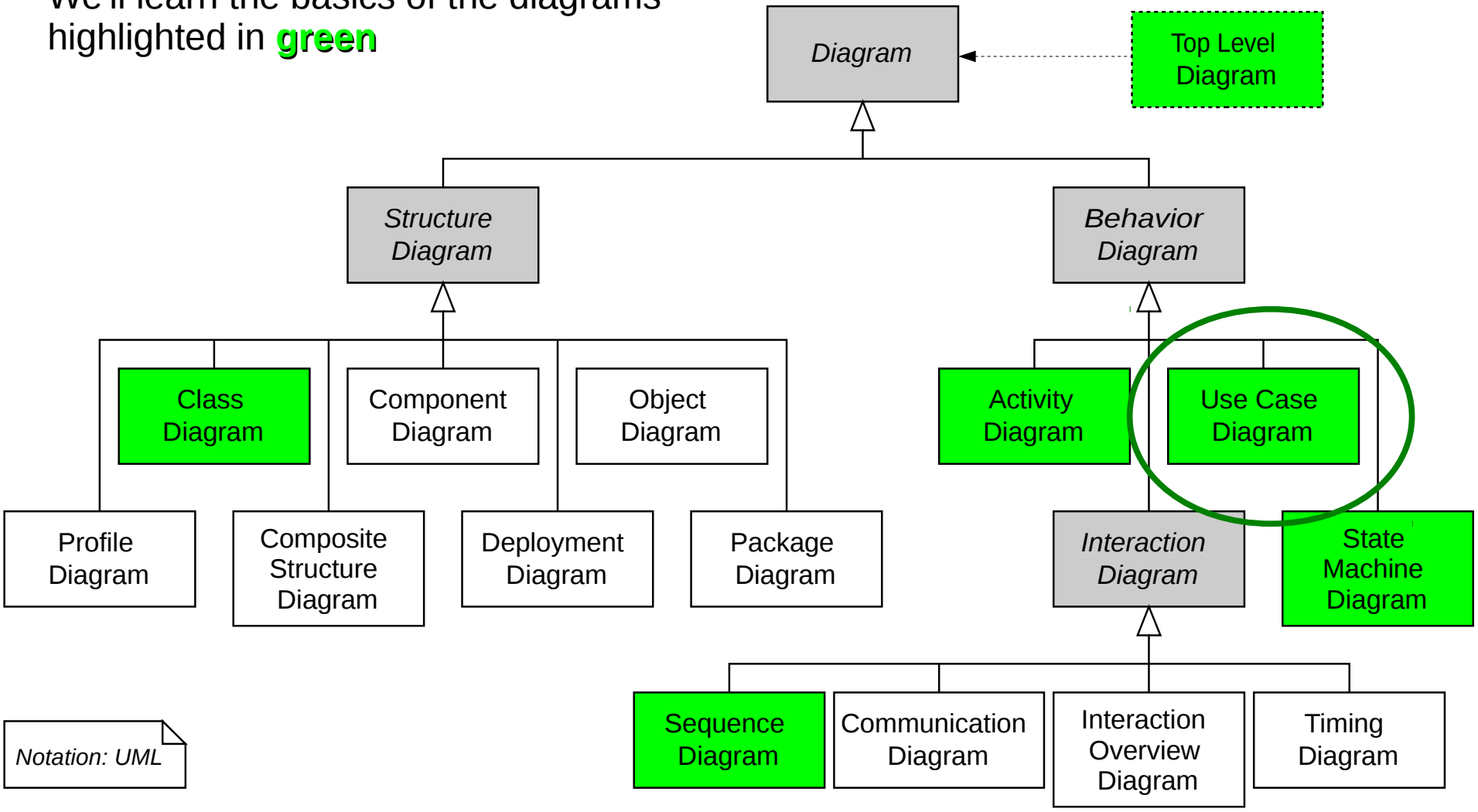


http://www.uml-diagrams.org/use-case-diagrams.html

Used with written permission.

The Use Case Diagram in Context

We'll learn the basics of the diagrams highlighted in **green**



Original source: Wikipedia, Public Domain SVG

What We Learned Today

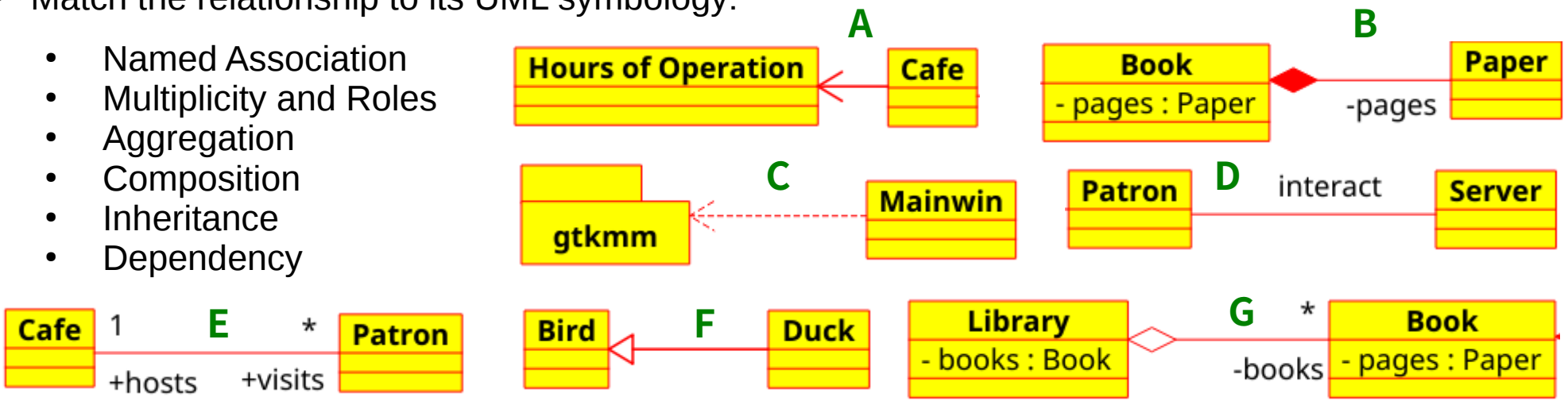
- Enum Classes enforce types
  - NOT just another name for an int!
  - Type enforcement catches more errors at compile time
  - But NOT a class – no methods or fields
- Operators can be defined for our own types using “operator” + symbol, e.g., operator+
  - You must follow the C++ declaration exactly
  - You cannot invent your own operators
- Inheritance implements the “isa” relationships
  - All public / protected base class members become members of the derived class
  - The derived class can change inherited members and add new members
  - The tree of inheritance relationships is the “class hierarchy”
- And more UML: Relationships, Extensions, and Use Cases (Oh, my!)

Quick Review

- The PIE definition of Object-Oriented Programming posits 3 foundations: _____, _____, and _____. Define 2 of the 3.
- True or False: A class is a user-defined type.
- True or False: The only difference in C++ between a struct and a class is the default visibility of its members.
- True or False: A class (type) may not be used to instantiate templates such as vector
- What is a friend function?
- True or False: A friend function is a member of the class with which it is friends.
- Ensuring that a program operates on clean, correct and useful data is _____ _____.
- Algorithmically enforceable constraints on the correctness, meaningfulness, and security of input data is _____ _____.
- Why should classes be designed to guarantee that the data they encapsulate is valid?
- True or False: An enum class is just an enum that also permits methods.
- What is the best way to print out (stream) an enum class variable?

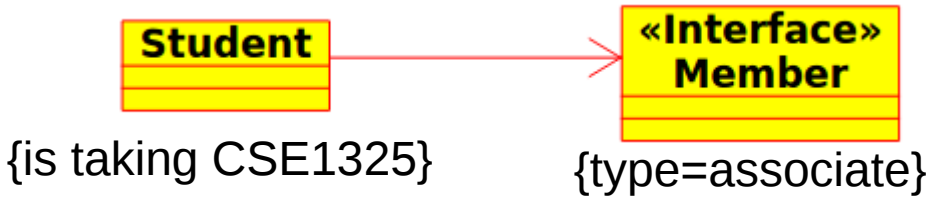
Quick Review

- Providing a user-defined meaning to a pre-defined operator (e.g., +, ==, <<) for a user-defined type (class) is _____ _____.
- To overload the + operator for a class, define the _____ method.
- True or False: C++ allows you to define new operators, such as @@.
- Reuse and extension of fields and method implementations from one class to other classes is called _____. The original class is called the _____ class, and the other classes are called _____ classes.
- A ____ _____ defines the inheritance relationships between classes.
- In specifying a base class in the derived class, the keyword “public”, “protected”, or “private” is specified. What are the implications of this?
- Match the relationship to its UML symbology:
  - Named Association
  - Multiplicity and Roles
  - Aggregation
  - Composition
  - Inheritance
  - Dependency



Quick Review

- Identify the 3 UML extensions is this diagram. What do they mean?



- A series of interactions that enable an actor to achieve a goal is a _____ _____.
- True or false: In UML, a use case is treated as a class.

For Next Class

- (Optional) Review Chapter 9 in Stroustrop
  - Do the Drills!
- Skim Chapter 10 for next Tuesday
  - Multiple Inheritance Considerations
  - File I/O – beyond the bare basics

Homework #3

This assignment involves writing a word / phrase guessing game (always a good way to learn a new language and new technology). We'll utilize a Makefile, use the ddd (or gdb) debugger, employ git (of course!), and use Umbrello to design our class and use case diagrams.

Due Thursday, February 8 at 8 am.