CSCI 103 – Unit 5d
Inheritance

CSCI 103L Teaching Team

Recall: Constructor Initialization

```
Student::Student()
{
  name = "Tommy Trojan";
  id = 12313
  scores.resize(10);
}
```

```
Student::Student() :
    name(), id(), scores()
    // calls to default constructors
{
  name = "Tommy Trojan";
  id = 12313
  scores.resize(10);
}
```

You can still assign data members in the {…}

But any member not in the initialization list will have its default constructor invoked before the {…}

- You can still assign values in the constructor but realize that the **default constructors** will have been called already
- So generally if you know what value you want to assign a data member it's **good practice** to do it in the initialization list

```
Student::Student() :
    name("Tommy"), id(12313), scores(10)
{  }
```

This would be the preferred approach especially for any non-scalar members (i.e. an object)

Object Oriented Design Concepts

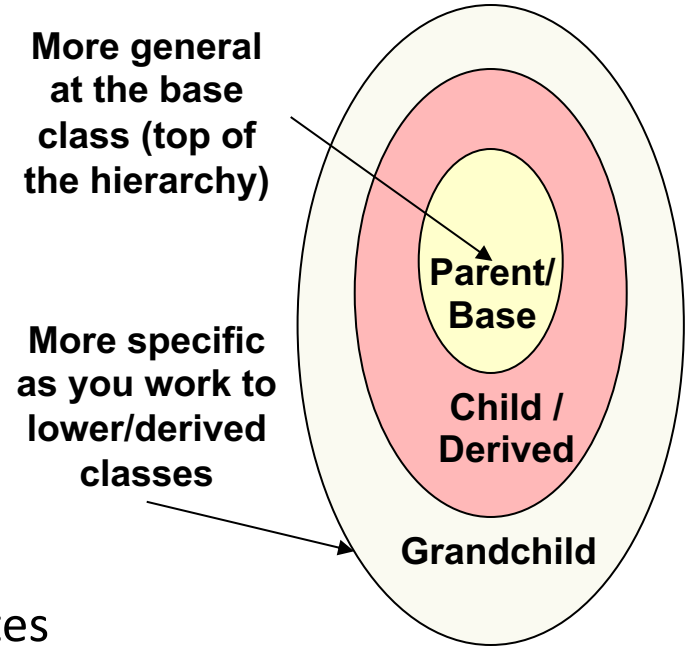
- Encapsulation and Abstraction
  - Combine data and operations on that data into a single unit and only expose a desired public interface and prevent modification/alteration of the implementation
- Inheritance
  - Creating new objects (classes) from existing ones to specify functional relationships and extend behavior
- Polymorphism
  - Using the same expression to support different types with different behavior for each type

Coupling

- Coupling refers to how much components depend on each other's implementation details (i.e. how much work it is to remove one component and drop in a new implementation of it)
  - Placing a new battery in your car vs. a new engine
  - Adding a USB device vs. a new video adapter to your laptop
- OO Design seeks to reduce coupling as much as possible by
  - Creating well-defined interfaces to update (write) or access (read) the state of an object
  - Allow alternate implementations that do NOT require interface changes

Inheritance

- A way of defining interfaces, re-using classes and extending original functionality
- Allows a new class to inherit all the data members and member functions from a previously defined class
- Works from more general objects to more specific objects
  - Defines an "is-a" relationship
  - Square is-a rectangle is-a shape
  - Square inherits from Rectangle which inherits from Shape
  - Similar to classification of organisms:
    - Animal -> Vertebrate -> Mammals -> Primates



Base and Derived Classes

- Derived classes inherit all data members and functions of base class
- Student class inherits:
  - `get_name()` and `get_id()`
  - `name_` and `id_` member variables

```
class Person {
 public:
  Person(string n, int ident);
  string get_name();
  int get_id();
 private:
  string name_; int id_;
};
class Student : public Person {
 public:
  Student(string n, int ident, int mjr);
  int get_major();
  double get_gpa();
  void set_gpa(double new_gpa);
 private:
  int major_; double gpa_;
};
```

class Person

string name_
int id_

class Student

string name_
int id_
int major_
double gpa_

Base and Derived Classes

- Derived classes inherit all data members and functions of base class
- Student class inherits:
  - get_name() and get_id()
  - name_ and id_ member variables

class Person

string name_
int id_

class Student

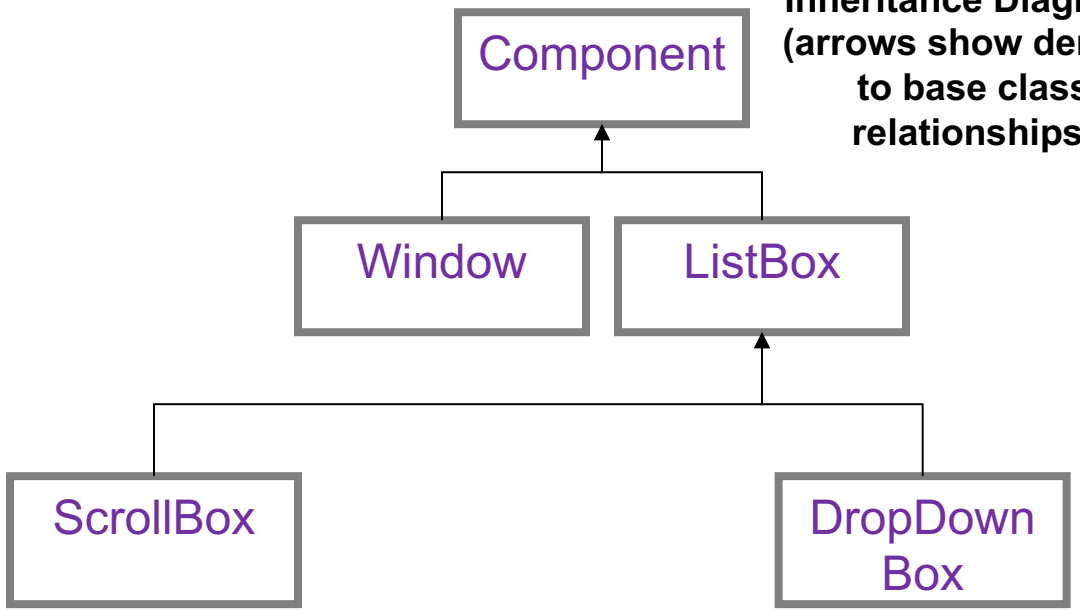
string name_
int id_
int major_
double gpa_

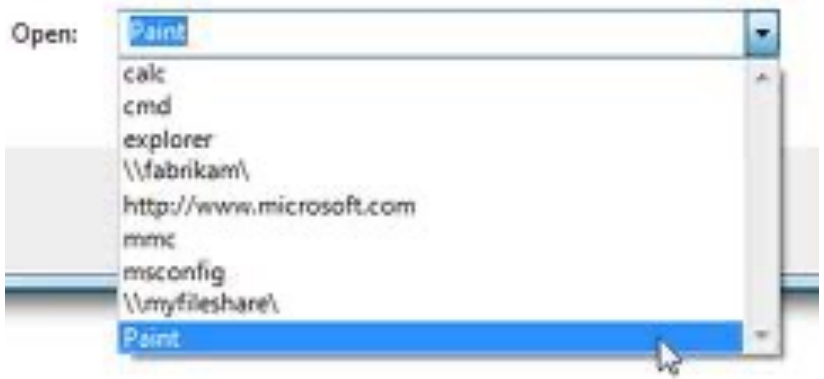
```
class Person {
 public:
  Person(string n, int ident);
  string get_name();
  int get_id();
 private:
  string name_; int id_;
};
class Student : public Person {
 public:
  Student(string n, int ident, int mjr);
  int get_major();
  double get_gpa();
  void set_gpa(double new_gpa);
 private:
  int major_; double gpa_;
};

int main()
{
  Student s1("Tommy", 1, 9);
  // Student has Person functionality
  // as if it was written as part of
  // Student
  cout << s1.get_name() << endl;

}
```

Inheritance Example

- Component
  - Draw()
  - onClick()
- Window
  - Minimize()
  - Maximize()
- ListBox
  - Get_Selection()
- ScrollBox
  - onScroll()
- DropDownBox
  - onDropDown()

Inheritance Diagrams (arrows show derived to base class relationships)





CONSTRUCTORS AND INHERITANCE

Constructors and Inheritance

- How do we initialize base class data members?
- Can't assign base class members if they are private

```
class Person {
 public:
  Person(string n, int ident);
  ...
 private:
  string name_;
  int id_;
};
class Student : public Person {
 public:
  Student(string n, int ident, int mjr);
  ...
 private:
  int major_;
  double gpa_;
};

Student::Student(string n, int ident, int mjr)
{
   name_ = n;   // can we access name_ and id_?
   id_ = ident;
   major_ = mjr;
}
```

Constructors and Inheritance

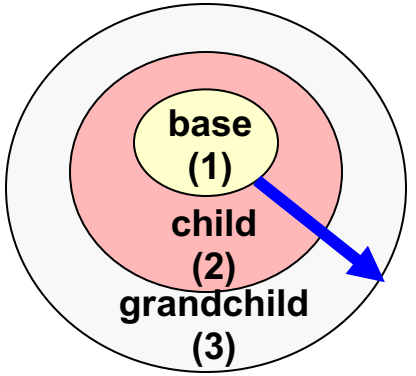
- Constructors are only called when a variable is created and cannot be called directly from another constructor
  - How to deal with base constructors?
- Also want/need base class or other members to be initialized before we perform this object's constructor code
- Use initializer format instead
  - See example below

```
class Person {
 public:
  Person(string n, int ident);
  ...
 private:
  string name_;
  int id_;
};
class Student : public Person {
 public:
  Student(string n, int ident, int mjr);
  ...
 private:
  int major_;
  double gpa_;
};

Student::Student(string n, int ident, int mjr)
{
 // How to initialize Base class members?
  Person(n, ident); // No! can't call Construc.
                    //   as a function

}
```

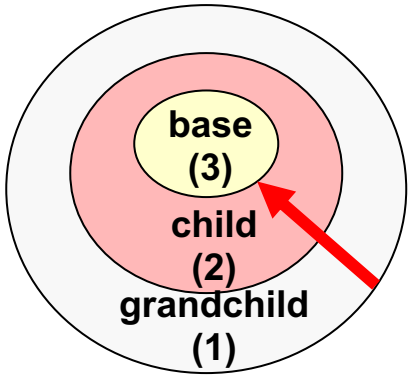
```
Student::Student(string n, int ident, int mjr) :
   Person(n, ident)
{
  cout << "Constructing student: " << name_ << endl;
  major_ = mjr;   gpa_ = 0.0;
}
```

Constructors & Destructors

- Constructors
  - A Derived class will automatically call its Base class constructor **BEFORE** it's own constructor executes, either:
    - Explicitly calling a specified base class constructor in the initialization list
    - Implicitly calling the default base class constructor if no base class constructor is called in the initialization list
- Destructors
  - The derived class will call the Base class destructor automatically **AFTER** it's own destructor executes
- General idea
  - Constructors get called from base->derived (smaller to larger)
  - Destructors get called from derived->base (larger to smaller)



Constructor call ordering



Destructor call ordering

Constructor & Destructor Ordering

```
class A {
  int a;
public:
  A()  { a=0; cout << "A:" << a << endl; }
  ~A() { cout << "~A" << endl; }
  A(int mya) { a = mya;
               cout << "A:" << a << endl; }
};

class B : public A {
  int b;
public:
  B()  { b = 0; cout << "B:" << b << endl; }
  ~B() { cout << "~B "; }
  B(int myb) { b = myb;
               cout << "B:" << b << endl; }
};

class C : public B {
  int c;
public:
  C()  { c = 0; cout << "C:" << c << endl; }
  ~C() { cout << "~C "; }
  C(int myb, int myc) : B(myb) {
     c = myc;
     cout << "C:" << c << endl; }
};
```

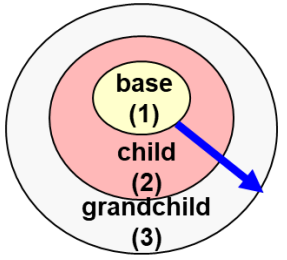
Sample Classes

```
int main()
{
  cout << "Allocating a B object" << endl;
  B b1;
  cout << "Allocating 1st C object" << endl;
  C* c1 = new C;
  cout << "Allocating 2nd C object" << endl;
  C c2(4,5);
  cout << "Deleting c1 object" << endl;
  delete c1;
  cout << "Quitting" << endl;
  return 0;
}
```

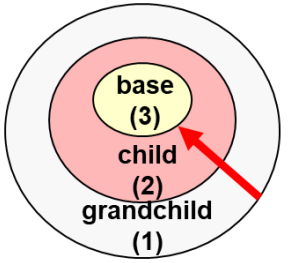
Test Program

```
Allocating a B object
A:0
B:0
Allocating 1st C object
A:0
B:0
C:0
Allocating 2nd C object
A:0
B:4
C:5
Deleting c1 object
~C ~B ~A
Quitting
~C ~B ~A
~B ~A
```

Output



Constructor call ordering



Destructor call ordering

PUBLIC, PRIVATE, PROTECTED

Protected Members

- Private members of a base class can not be accessed directly by a derived class member function
  - Code for print_grade_report() would not compile since 'name_' is private to class Person
- Base class can declare variables with protected storage class which means:
  - Private to any object or code not inheriting from the base (i.e. private to any 3rd party)
  - Public to any derived (child) class can access directly

```
class Person {
 public:
  ...
 private:
  string name_; int id_;
};

class Student : public Person {
 public:
   void print_grade_report();
 private:
   int major_; double gpa_;
};
```

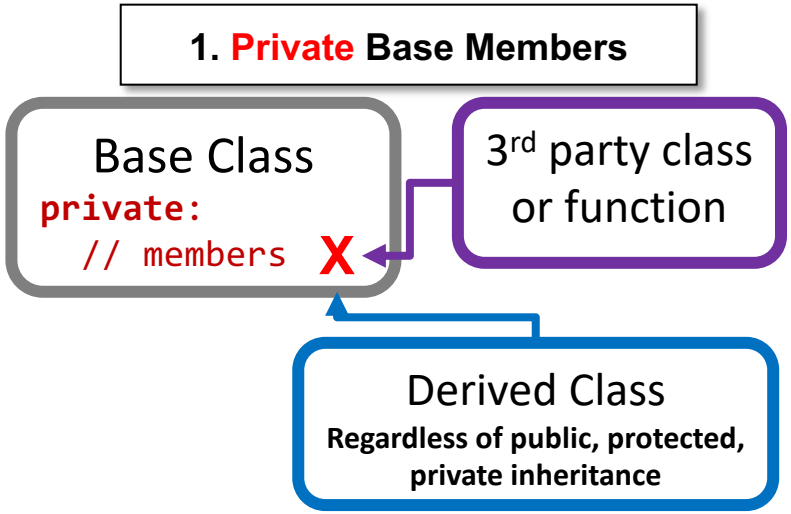
```
void Student::print_grade_report()
{
  cout << "Student " << name_ << ...
}
```

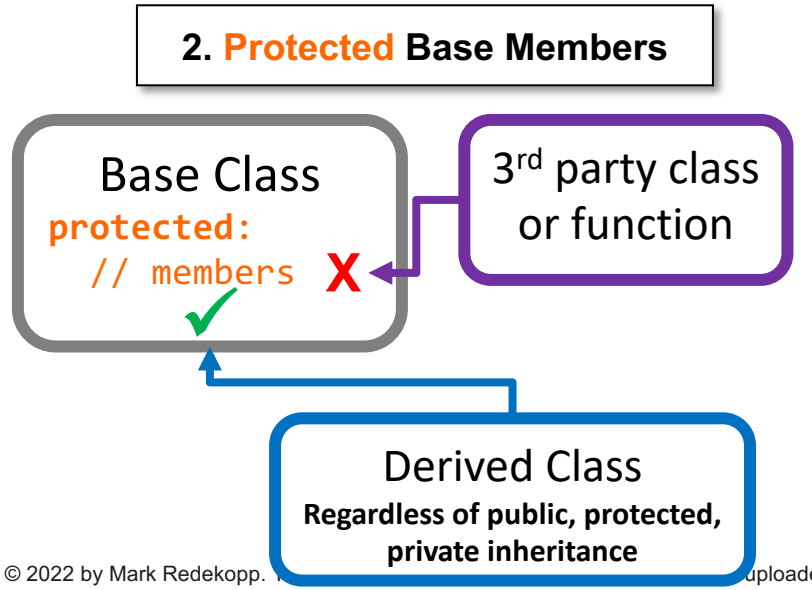
X

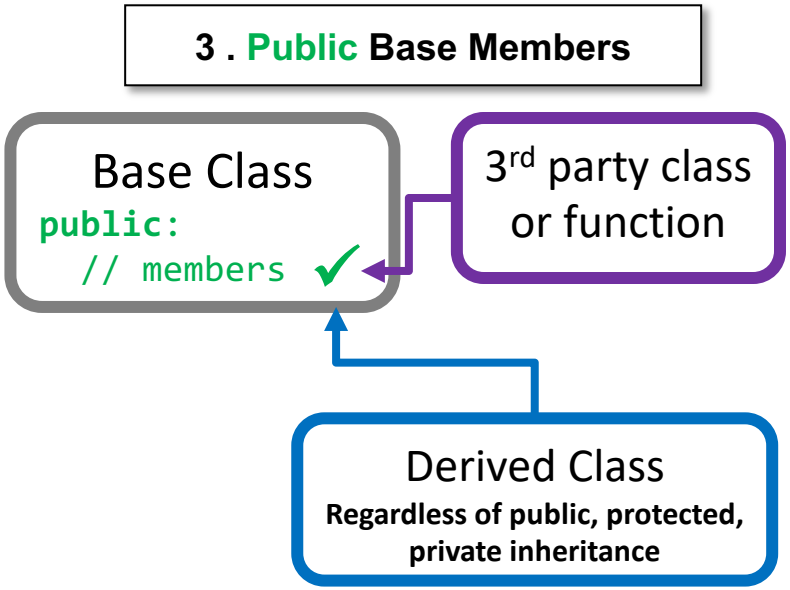
```
class Person {
 public:
  ...
 protected:
  string name_; int id_;
};
```

Public, Protected, & Private Access

- Derived class sees base class members using the base class' specification
  - If Base class said it was **public** or **protected**, the derived class **can** access it directly
  - If Base class said it was **private**, the derived class **cannot** access it directly







Public/Private/Protected Inheritance

- public/protected/private inheritance before base class indicates HOW the public base class members are viewed by clients (those outside) of the derived class
- public
  - public and protected base class members are accessible to the child class and grandchild classes
  - Only public base class members are accessible to 3rd party clients
- protected
  - public and protected base class members are accessible to the child class and grandchild classes
  - no base class members are accessible to 3rd parties
- private
  - public and protected base class members are accessible to the child class
  - No base class members are accessible to grandchild classes or 3rd party clients

Base Class

```
class Person {
 public:
  Person(string n, int ident);
  string get_name();
  int get_id();
 private: // INACCESSIBLE TO DERIVED
  string name_; int id_;
};
```

```
class Student : public Person {
 public:
  Student(string n, int ident, int mjr);
  int get_major();
  double get_gpa();
  void set_gpa(double new_gpa);
 private:
  int major_; double gpa_;
};
class Faculty : private Person {
 public:
  Faculty(string n, int ident, bool tnr);
  bool get_tenure();
 private:
  bool tenure_;
};
```

```
int main(){
  Student s1("Tommy", 73412, 1);
  Faculty f1("Mark", 53201, 2);
  cout << s1.get_name() << endl; // works
  cout << f1.get_name() << endl; // fails
}
```

Inheritance Access

- Derive as public if…
  - You want users of your derived class to be able to call base class functions/methods
- Derive as private if…
  - You only want your internal workings to call base class functions/methods
- Derive as protected more rearely
  - Same reasons as private inheritance but also allow grandchild classes to use Base class methods

Inherited Base	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Private	Private	Private

External client access to Base class members is always the more restrictive of either the base declaration or how the base is inherited.

```
class Person {
 public:
  Person(string n, int ident);
  string get_name();
  int get_id();
 private: // INACCESSIBLE TO DERIVED
  string name_; int id_;
};
```

```
class Student : public Person {
 public:
  Student(string n, int ident, int mjr);
  int get_major();
  double get_gpa();
  void set_gpa(double new_gpa);
 private:
  int major_; double gpa_;
};
class Faculty : private Person {
 public:
  Faculty(string n, int ident, bool tnr);
  bool get_tenure();
 private:
  bool tenure_;
};
```

```
int main(){
  Student s1("Tommy", 73412, 1);
  Faculty f1("Mark", 53201, 2);
  cout << s1.get_name() << endl; // works
  cout << f1.get_name() << endl; // fails
}
```

Public/Private/Protected Cases

Base Class

```
public: void f1();
protected: void f2();
private: void f3();
```

How a grandchild class or 3rd party sees what is inherited is the MORE restrictive of the how the base class declared it or how the derived class inherited.

Child

```
class ChildA :
 public Base
{ /* . . . */  };
```

```
class ChildB :
 protected Base
{ /* . . . */  };
```

```
class ChildC :
 private Base
{ /* . . . */  };
```

Grandchild

```
class GCA :
  public ChildA
{ public:
 void g1()
 { f1(); f2(); f3();}
}
```
✓ ✓ X

```
class GCB :
  public ChildB
{ public:
  void g1()
  { f1(); f2(); f3(); }
}
```
✓ ✓ X

```
class GCC :
  public ChildC
{ public:
  void g1()
  { f1(); f2(); f3(); }
}
```
X X X

3rd Party

```
int main()
{ ChildA a;
  a.f1(); a.f2();a.f3();
}
```
✓ X X

```
int main()
{ ChildB b;
  b.f1(); b.f2(); b.f3();
}
```
X X X

```
int main()
{ ChildC c;
  b.f1(); b.f2(); b.f3();
}
```
X X X

When to Inherit Privately

- If public: Outside user can call the base List functions and break the Queue order
- If private: hide the base class public function, so users can only call derived class interface
- If protected: hide the base class public and protected functions except to derived and friend classes
- For protected or private inheritance, "(implemented) as-a" relationship
  - Queue "implemented as-a" List

```
class List{
 public:
  List();
  void insert(int loc, int val);
  int size();
  int get(int loc); // get value at loc
  void erase(int loc;)
 private:
  Item* _head;
};
```

Base Class

```
class Queue : public List // or private List
{ public:
  Queue();
  push_back(int val)
    { insert(size(), val); }
  int front();
    { return get(0); }
  void pop_front();
    { erase(0); }
};
```

Derived Class

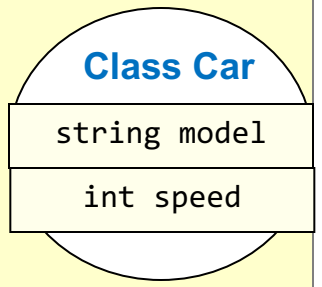
```
Queue q1;
q1.push_back(7); q1.push_back(8);
q1.insert(0,9) // is it good this is allowed?
```

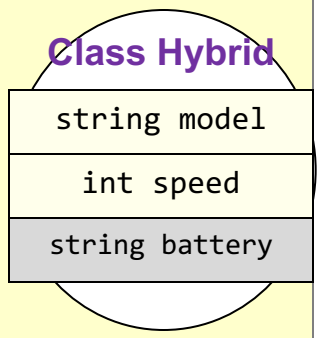
ODDS AND ENDS OF INHERITANCE

Overloading Base Functions

- A derived class may want to redefined the behavior of a member function of the base class
- A base member function can be overloaded in the derived class
- When derived objects call that function the derived version will be executed
- When a base objects call that function the base version will be executed

```
class Car{
 public:
  double compute_mpg();
 private:
  string model; int speed;
};

double Car::compute_mpg()
{
  if(speed > 55) return 30.0;
  else return 20.0;
}


class Hybrid : public Car {
 public:
  void drive_w_battery();
  double compute_mpg();
 private:
  string batteryType;
};

double Hybrid::compute_mpg()
{
  if(speed <= 15) return 45; // hybrid mode
  else if(speed > 55) return 30.0;
  else return 20.0;
}
```





Scoping Base Functions

- We can still call the base function version by using the scope operator (::)
  - `base_class_name::function_name()`

```
class Car{
 public:
  double compute_mpg();
 private:
  string make; string model;
};

double Car::compute_mpg()
{
  if(speed > 55) return 30.0;
  else return 20.0;
}


class Hybrid : public Car {
 public:
  void drive_w_battery();
  double compute_mpg();
 private:
  string batteryType;
};

double Hybrid::compute_mpg()
{
  if(speed <= 15) return 45; // hybrid mode
  else return Car::compute_mpg();
}
```

COMPOSITION VS. INHERITANCE

Composition

- Code reuse is a common need in (object-oriented) programming
  - We could use a pre-written List class to make a Queue class
- An easy and often preferable way is to simply use the existing class as a data member
- ***Composition defines a "has-a" relationship***
  - A Queue "has-a" List in its implementation
- But could we inherit?
  - Public inheritance would mean a Queue "is-a" List and a Queue should be able to do anything a List can do, but that's not the case
  - Private inheritance could be used but is not a universal approach supported by other languages
  - Often programmers say "**prefer composition rather than inheritance**" when the goal is code reuse

```
class List{
 public:
  List();
  void insert(int loc, int val);
  int size();
  int get(int loc);
  void erase(int loc;)
 private:
  Item* head_;
};
```

Base Class

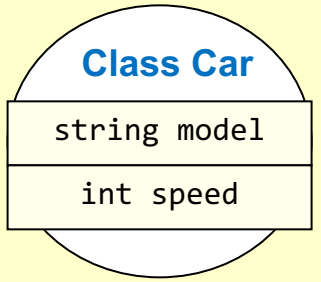
```
class Queue
{ private:
    List mylist;
  public:
   Queue();
   push_back(int val)
    { mylist.insert(size(), val); }
   int front();
    { return mylist.get(0); }
   void pop_front();
    { mylist.erase(0); }
   int size() // need to create wrapper
     { return mylist.size(); }
};
```

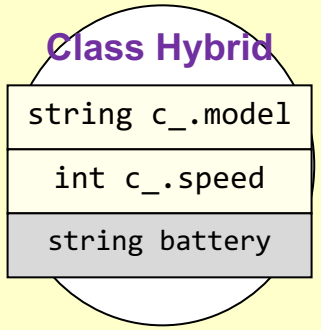
Queue via Composition

Inheritance vs. Composition

- Software engineers debate about using ***inheritance (is-a)*** vs. ***composition (has-a)***
- Rather than a Hybrid "is-a" Car we might say Hybrid "has-a" car in it, plus other stuff
- While it might not make complete sense verbally, we could re-factor our code the following ways…
- Interesting article I'd recommend you read at least once:
  - https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose

```
class Car{
 public:
  double compute_mpg();
 private:
  string model; int speed;
};

double Car::compute_mpg()
{
  if(speed > 55) return 30.0;
  else return 20.0;
}

class Hybrid {
 public:
  double compute_mpg();
 private:
  Car c_; // has-a relationship
  string batteryType;
};
double Hybrid::compute_mpg()
{
  if(speed <= 15) return 45; // hybrid mode
  else return c_.compute_mpg();
}
```





Inheritance vs. Composition

- Suppose we wanted to create a variation of the `std::string` class that only allows a fixed size specified at creation (no size alteration after creation)
  - What is the best way to enforce this?

```
class FixedString : public string
{ public:
   FixedString(size_t fixedSize) :
     string(' ', fixedSize)
   { }
};
```

Using Public Inheritance

```
class FixedString : private std::string
{ public:
   FixedString(size_t fixedSize) :
     std::string(' ', fixedSize)
   { }
   size_t size() const { return string::size(); }
   char const & operator[](size_t idx) const
    { return string::operator[idx]; }
   ...
};
```

Using Private Inheritance

```
class FixedString
{ private:
   string str_;
  public:
   FixedString(size_t fixedSize) :
     str_(' ', fixedSize)
   { }
   size_t size() const { return str_.size(); }
   char const & operator[](size_t idx) const
     { return str_[idx]; }
   ...
};
```

Using Composition

Which is/are reasonable choices? Consider the code to the right in making your decision?

```
FixedString s1(10);
s1[0] = 'a';
// will the compiler allow these
s1 = "abcdefghijklmnopqrstuvwxyz";
s1 += "abc";
```

Summary

- Summary:
  - **Public Inheritance => "is-a" relationship**
  - **Composition => "has-a" relationship**
  - **Private/Protected Inheritance => "as-a" relationship or "implemented-as" or "implemented-in-terms-of"**
- Public inheritance mainly when
  - We want to add or specialize behavior
  - A true "is-a" relationship holds for the relationship of base and derived
- Composition or private inheritance
  - When reuse is the main desire

```
class List{
 public:
  List();
  void insert(int loc, const int& val);
  int size();
  int& get(int loc);
  void pop(int loc;)
 private:
  IntItem* _head;
};
```

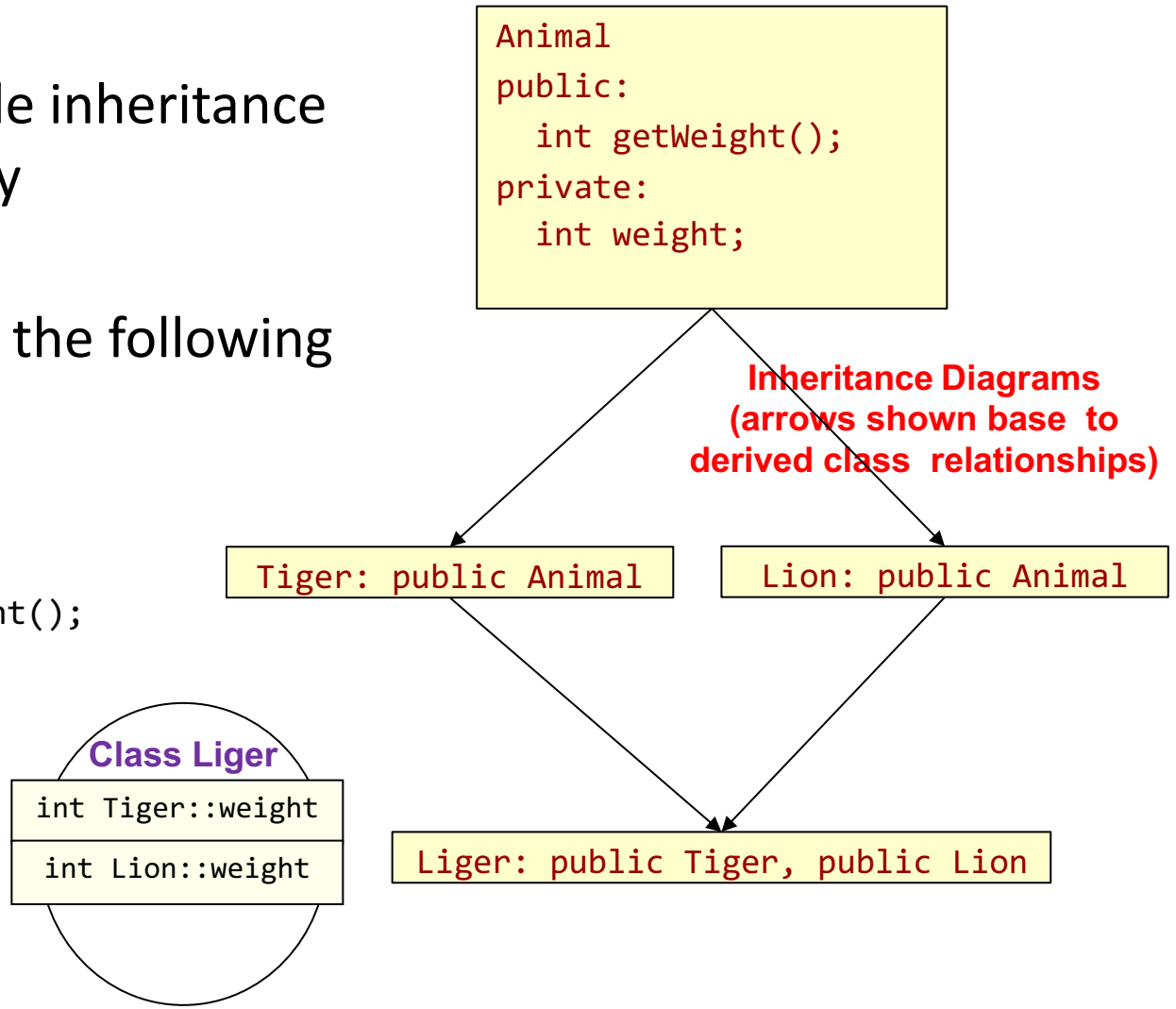
Base Class

```
class Queue
{ private:
   List mylist;
  public:
   Queue();
   push_back(const int& val)
    { mylist.insert(size(), val); }
   int& front();
    { return mylist.get(0); }
   void pop_front();
    { mylist.pop(0); }
   int size() // need to create wrapper
     { return mylist.size(); }
};
```

Queue via Composition

Warning: Multiple Inheritance

- C++ allows multiple inheritance but it is not usually recommended
- What happens for the following code?
- Suppose in main()
  - `Liger x;`
  - `int wt = x.getWeight();`



Example source: https://www.programmerinterview.com/index.php/c-cplusplus/diamond-problem

REVIEW QUESTIONS

Inheritance Review 1

- **T/F**: A student object has a name_ and id_ member
- **T/F**: Code from the Student class can access name_ and id_
  - What could you change to flip the T/F answer?
- What would change if Student inherited Person through private inheritance?

```
class Person {
 public:
  Person(string n, int ident);
  string get_name();
  int get_id();
 private:
  string name_; int id_;
};

class Student : public Person {
 public:
  Student(string n, int ident, int mjr);
  int get_major();
  double get_gpa();
  void set_gpa(double new_gpa);
 private:
  int major_; double gpa_;
};
int main()
{
  Student s1("Amanda", 12345, 1);
  cout << s1.get_name() << endl;
  return 0;
}
```

Inheritance Review 2

- Inheritance defines an _____ relationship between classes
- Composition defines a _____ relationship between two objects
- Protected access makes members accessible to _____ but still not to _____

Constructor & Destructor Ordering

```
class A {
  int a;
public:
  A()  { a=0; cout << "A:" << a << endl; }
  ~A() { cout << "~A" << endl; }
  A(int mya) { a = mya;
               cout << "A:" << a << endl; }
};

class B : public A {
  int b;
public:
  B()  { b = 0; cout << "B:" << b << endl; }
  ~B() { cout << "~B "; }
  B(int myb) { b = myb;
               cout << "B:" << b << endl; }
};

class C : public B {
  int c;
public:
  C()  { c = 0; cout << "C:" << c << endl; }
  ~C() { cout << "~C "; }
  C(int myb, int myc) : B(myb) {
     c = myc;
     cout << "C:" << c << endl; }
};
```

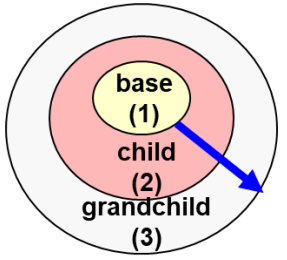
Sample Classes

```
int main()
{
  cout << "Allocating a B object" << endl;
  B b1;
  cout << "Allocating 1st C object" << endl;
  C* c1 = new C;
  cout << "Allocating 2nd C object" << endl;
  C c2(4,5);
  cout << "Deleting c1 object" << endl;
  delete c1;
  cout << "Quitting" << endl;
  return 0;
}
```

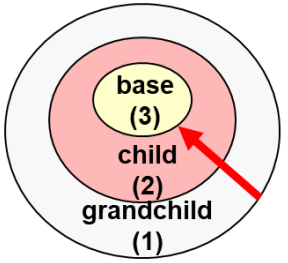
Test Program

```
Allocating a B object
A:0
B:0
Allocating 1st C object
A:0
B:0
C:0
Allocating 2nd C object
A:0
B:4
C:5
Deleting c1 object
~C ~B ~A
Quitting
~C ~B ~A
~B ~A
```

Output



Constructor call ordering



Destructor call ordering

PRE-SUMMER 2021 INHERITANCE SLIDES

Inheritance vs. Composition

- Suppose we wanted to create a variation of the `std::string` class that only allows a fixed size specified at creation (no size alteration after creation)
  - What is the best way to enforce this?

```
class FixedString : public string
{ public:
   FixedString(size_t fixedSize) :
     string(' ', fixedSize)
   { }
};
```

Using Public Inheritance

```
class FixedString : private std::string
{ public:
   FixedString(size_t fixedSize) :
     std::string(' ', fixedSize)
   { }
   size_t size() const { return string::size(); }
   char const & operator[](size_t idx) const
    { return string::operator[idx]; }
   ...
};
```

Using Private Inheritance

```
class FixedString
{ private:
   string str_;
  public:
   FixedString(size_t fixedSize) :
     str_(' ', fixedSize)
   { }
   size_t size() const { return str_.size(); }
   char const & operator[](size_t idx) const
     { return str_[idx]; }
   ...
};
```

Using Composition

Which is/are reasonable choices? Consider the code to the right in making your decision?

```
FixedString s1(10);
s1[0] = 'a';
S1 += "abc"; // will the compiler allow this
```

SOLUTIONS

Inheritance Review 1

- **T/F**: A student object has a name_ and id_ member
- **T/F**: Code from the Student class can access name_ and id_
  - What could you change to flip the T/F answer? Changing Person's access specifier to protected or public. Regardless of how Student inherits, name_ and id_ will be private to the Student class.
- What would change if Student inherited Person through private inheritance?
  - External clients (like main) would not be able to access the inherited members (from Person) of a Student object.

```
class Person {
 public:
  Person(string n, int ident);
  string get_name();
  int get_id();
 private:
  string name_; int id_;
};

class Student : public Person {
 public:
  Student(string n, int ident, int mjr);
  int get_major();
  double get_gpa();
  void set_gpa(double new_gpa);
 private:
  int major_; double gpa_;
};
int main()
{
  Student s1("Amanda", 12345, 1);
  cout << s1.get_name() << endl;
  return 0;
}
```

Inheritance Review 2

- Inheritance defines an is-a relationship between classes
- Composition defines a has-a relationship between two objects
- Protected access makes members accessible to a derived/child class but still not to external/3rd-party clients