University of Southern California

Viterbi School of Engineering

Software Design

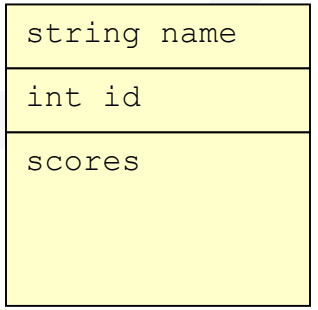
OOP: Inheritance, Composition, Interface, …

Reference: Professor Redekopp' EE355 slide units, online Resources

Consider this Struct/Class

- **Examine this struct/class definition...**

```
#include <string>
#include <vector>
using namespace std;

struct Student
{
  string name;
  int id;
  vector<double> scores;
   // say I want 10 test scores per student

};

int main()
{
  Student s1;
}
```

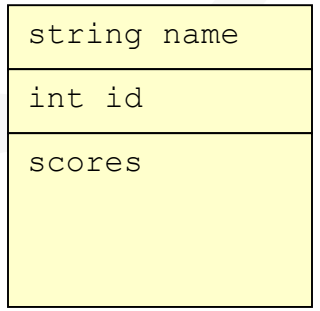


Composite Objects

- **Fun Fact: Memory for an object comes alive before the code for the constructor starts at the first curly brace '{'**

```
#include <string>
#include <vector>
using namespace std;

struct Student
{
  string name;
  int id;
  vector<double> scores;
   // say I want 10 test scores per student

  Student() /* mem allocated here */
  {
    // Can I do this to init. members?
    name("Tommy Trojan");
    id = 12313;
    scores(10);
  }
};

int main()
{
  Student s1;
}
```

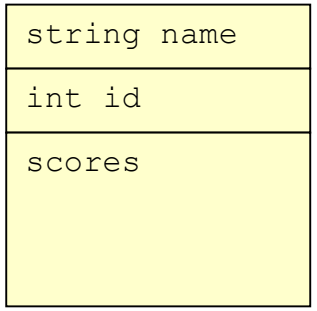
This would be "constructing" name twice. It's too late to do it in the {…}



Composite Objects (cont.)

- You cannot call constructors on data members once the constructor has started (i.e., passed the open curly '{')
  - So what can we do??? Use assignment operators (less efficient) or use constructor initialization lists!

```
#include <string>
#include <vector>
using namespace std;

struct Student
{
  string name;
  int id;
  vector<double> scores;
   // say I want 10 test scores per student

  Student() /* mem allocated here */
  {
    // Can I do this to init. members?
    name = "Tommy Trojan";
    id = 12313;
    scores = 10;
  }
};
int main()
{
  Student s1;
}
```



Constructor Initialization Lists

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

If you write this…

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

The compiler will still generate this.

- Though you do not see it, realize that the default constructors are implicitly called for each data member before entering the {…}
- You can then assign values but this is a 2-step process

Constructor Initialization Lists (cont.)

```
Student:: Student() /* mem allocated here */
{
  name("Tommy Trojan");
  id = 12313;
  scores(10);
}
```

You can't call member constructors in the {…}

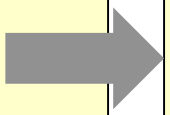
```
Student::Student() :
    name("Tommy"), id(12313), scores(10)
{   }
```

You would have to call the member constructors in the initialization list context

- Rather than writing many assignment statements we can use a special initialization list technique for C++ constructors
  - Constructor(param_list) : member1(param/val), ..., memberN(param/val)
    { ... }
- We are really calling the respective constructors for each data member

Constructor Initialization Lists (cont.)

```
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);
}
```

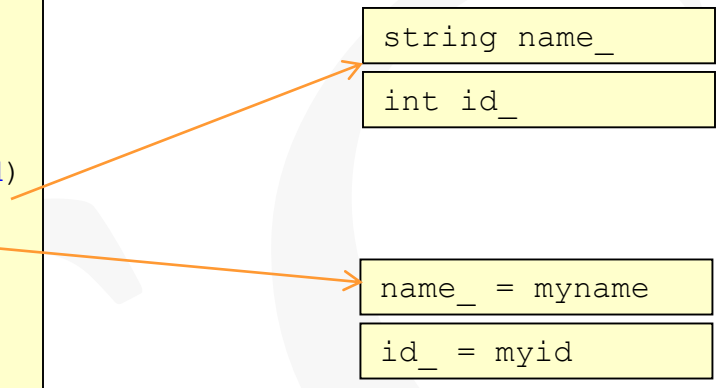
You can still assign data members in the {…}

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

- You can still assign values (which triggers operator=) 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 to avoid the extra time of the default constructor executing

Constructor Initialization Lists (cont.)

```
Student::Student() { }
Student::Student(string myname)
{ name_ = myname;
  id_ = -1;
}
Student::Student(string myname, int myid)
{ name_ = myname;
  id_ = myid;
}
...
```

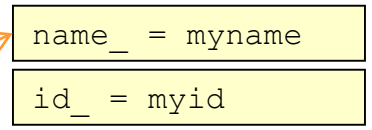


Memory is allocated before the '{' with the default constructor being called…

…then values copied in when assignment performed using operator=()

Initialization using assignment

```
Student::Student() { }
Student::Student(string myname) :
                 name_(myname), id_(-1)
{  }
Student::Student(string myname, int myid) :
                 name_(myname), id_(myid)
{   }
...
```



Memory is allocated and filled in "one-step" by calling the copy constructor

Initialization List approach

INHERITANCE

Exercise:

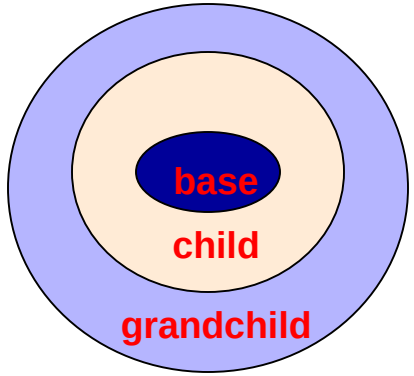
inh package

Object Oriented Design

- Encapsulation
  - Combine data and operations on that data into a single unit (e.g. a class w/ public and private aspects)
- Inheritance
  - Creating new objects (classes) from existing ones
- Polymorphism
  - Using the same expression to denote different operations

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

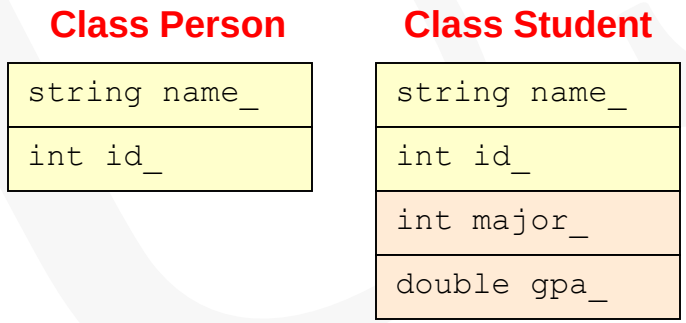
string name_
int id_

Class Student

string name_
int id_
int major_
double gpa_

Base and Derived Classes (cont.)

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

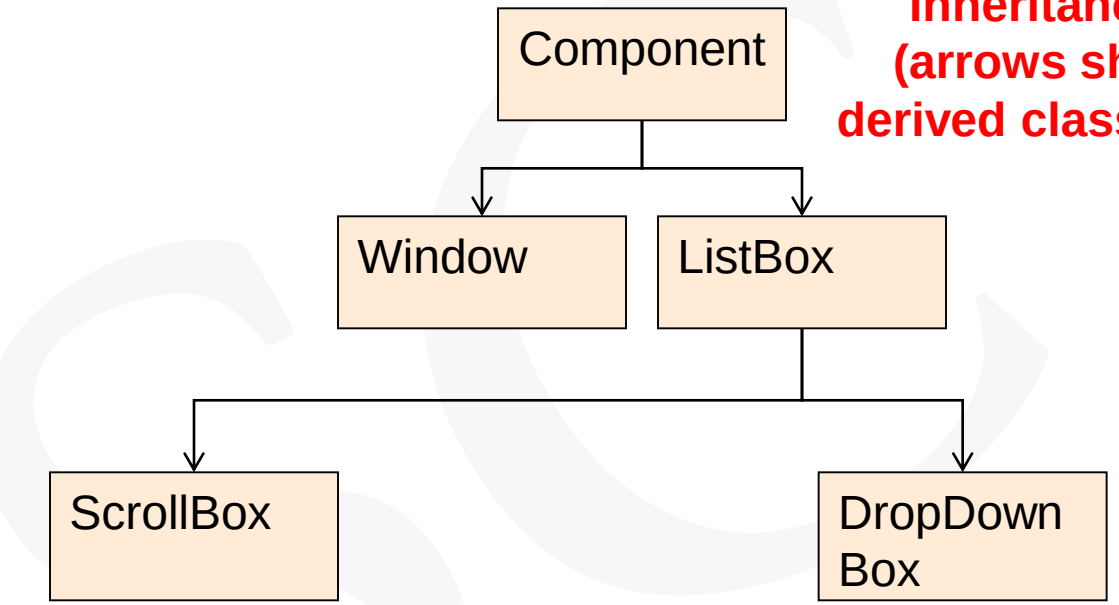


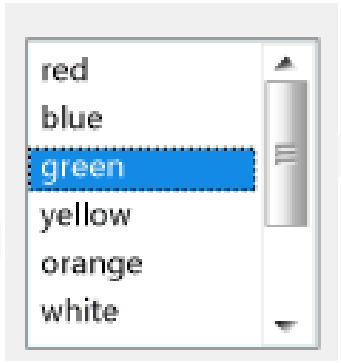

```
class Person {
 public:
  Person(string n, int ident);
  string get_name();
  int get_id();
 private:
  string name_; int id_;
};
class Student : public Person {
 public:
  Student(string n, int ident, int mjr);
  int get_major();
  double get_gpa();
  void set_gpa(double new_gpa);
 private:
  int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

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;

}
```

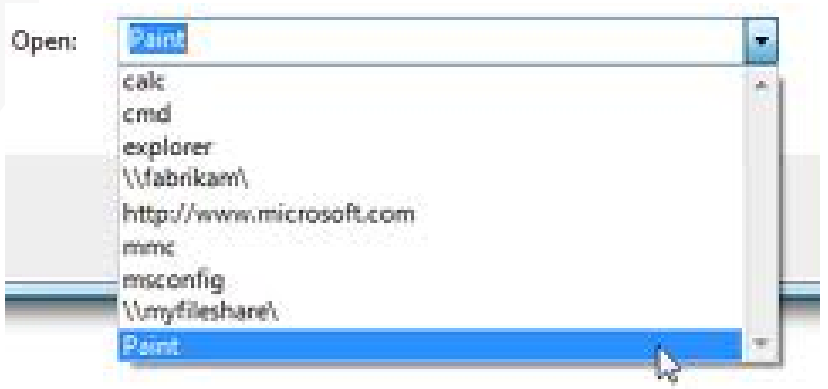
Inheritance Example

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

Inheritance Diagrams (arrows shown base to derived class relationships)







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't access name_ in Student
   id_ = ident;
   major_ = mjr;
}
```

Constructors and Inheritance (cont.)

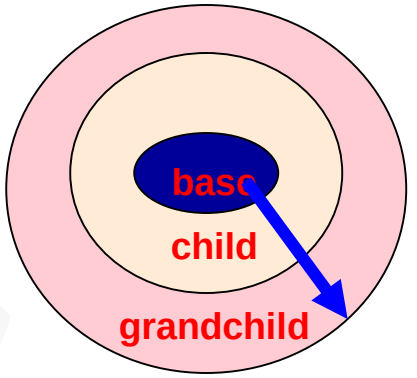
- Constructors are only called when a variable ‘enters scope’ (i.e., is created) and cannot be called directly
  - How to deal with base constructors?
- Also want/need base class or other members to be initialized before we perform this object's constructor code
- Use initializer format instead
  - See 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
}
```

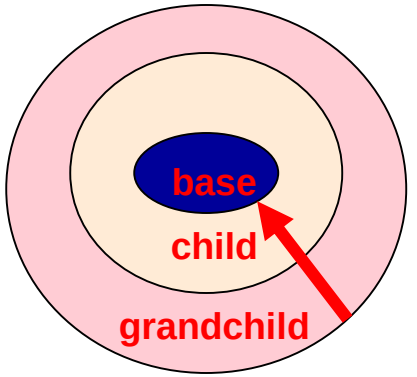
```
Student::Student(string n, int ident, int mjr) : Person(n, ident)
{
  cout << "Constructing student: " << name_ << endl;
  major_ = mjr;   gpa_ = 0.0;
}
```

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; }
};
```

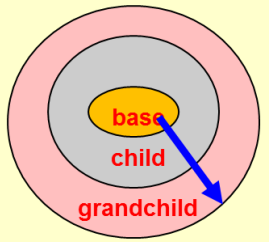
Sample Classes

```
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;
}
```

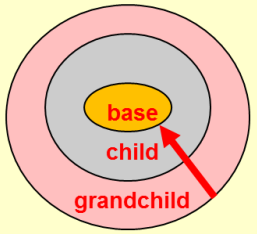
Test Program

```
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

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
  - Private to anyone not inheriting from the base
  - Derived classes 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/Private/Protected 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 identifier before base class indicates HOW the public base class members are viewed by clients (those outside) of the derived class
  - public => public base class members are public to clients (others can access)
  - private => public & protected base class members are private to clients (not accessible to the outside world)

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

Base Class

```
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_;
};
```

Derived Classes

Inheritance Access Summary

- Base class
  - Declare as protected if you want to allow a member to be directly accessed/modified by derived classes
- 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

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 inheritance level

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

Base Class

```
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
}
```

When to Inherit Privately

- Suppose I want to create a FIFO (First-in, First-Out) data structure where you can only
  - Push in the back
  - Pop from the front
- FIFO is-a special List
- Do I want to inherit publicly from List
- NO!!! Because now the outside user can call the base List functions and break my FIFO order
- Inherit privately to hide the base class public function and make users go through the derived class' interface
  - Private inheritance defines an "as-a" relationship

```
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 FIFO : public List // or private List
{ public:
  FIFO();
  push_back(const int& val)
    { insert(size(), val); }
  int& front();
    { return get(0); }
  void pop_front();
    { pop(0); }
};
```

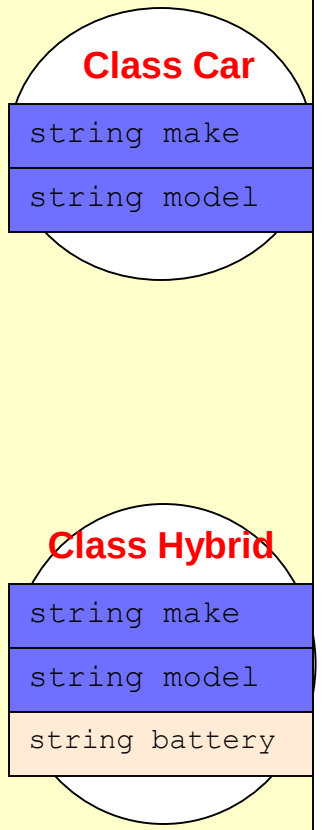
Derived Class

```
FIFO f1;
f1.push_back(7); f1.push_back(8);
f1.insert(0,9)
```

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 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 if(speed > 55) return 30.0;
  else return 20.0;
}
```



Scoping Base Functions

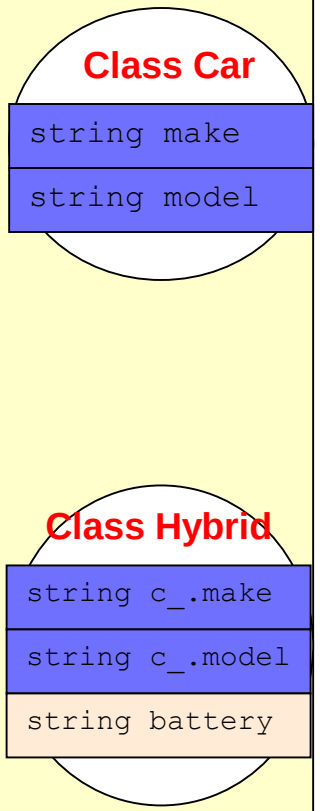
- We can still call the base function version by using the scope operator (::)
  - base_class_name::function_name()

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


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

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


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

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
  - Better example when we get to Lists, Queues and Stacks
- While it might not make complete sense verbally, we could re-factor our code the following ways...

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


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();
}
```



Another Composition

- We can create a FIFO that "has-a" a List as the underlying structure
- Summary:
  - Public Inheritance => "is-a" relationship
  - Composition => "has-a" relationship
  - Private Inheritance => "as-a" relationship
    "implemented-as"

```
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 FIFO
{ private:
   List mylist;
  public:
   FIFO();
   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(); }
};
```

FIFO via Composition

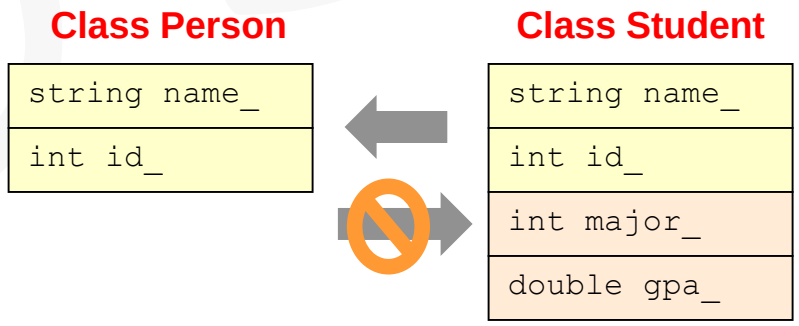
Virtual functions, Abstract classes, and Interfaces

POLYMORPHISM

Assignment of Base/Declared

- Can we assign a derived object into a base object and vice versa?
- To assign a = b, b must have everything a has
- Think hierarchy & animal classification (e.g., a Dog is a Mammal)
  - Does a dog nurse their young?
  - Does a mammal bark?
- We can only assign a derived into a base (since the derived has EVERYTHING the base does)
  - p = s; // Base = Derived...GOOD
  - s = p; // Derived = Base...BAD

```
class Person {
 public:
  void print_info(); // print name, ID
  string name; int id;
};

class Student : public Person {
 public:
  void print_info(); // print major too
  int major; double gpa;
};

int main(){
  Person p("Bill",1);
  Student s("Joe",2,5);
  // Which assignment is plausible
  p = s;  // or
  s = p;
}
```



Inheritance

- A pointer or reference to a derived class object is type-compatible with (can be assigned to) a base-class type pointer/reference
  - Person pointer or reference can also point to Student or Faculty object (i.e., a Student is a person)
  - All methods known to Person are supported by a Student object because it was derived from Person
  - Will apply the function corresponding to *the type of the pointer*
- For second and third call to print_info() we would like to have Student::print_info() and Faculty::print_info() executed since the actual object pointed to is a Student/Faculty
- This is called 'static binding'
  - Which version is called is based on the static type of the pointer being used

```
class Person {
 public:
  void print_info(); // print name, ID
  string name; int id;
};

class Student : public Person {
 public:
  void print_info(); // print major too
  int major; double gpa;
};

class Faculty : public Person {
 public:
  void print_info(); // print tenured
  bool tenure;
};

int main(){
  Person *p = new Person("Bill",1);
  Student *s = new Student("Joe",2,5);
  Faculty *f = new Faculty("Ken",3,0);
  Person *q;
  q = p; q->print_info();
  q = s; q->print_info();
  q = f; q->print_info();
}
```

Name=Bill, ID=1
Name=Joe, ID=2
Name=Ken, ID=3

Virtual Functions & Dynamic Binding

- Member functions can be declared 'virtual'
- 'Virtual' declaration allows derived classes to redefine the function *and* which version is called is determined by the most specific, i.e., the type of object pointed to/referenced rather than the type of pointer/reference
  - This is known as dynamic binding

```
class Person {
 public:
  virtual void print_info();
  string name; int id;
};

class Student : public Person {
 public:
  void print_info(); // print major too
  int major; double gpa;
};

class Faculty : public Person {
 public:
  void print_info(); // print tenured
  bool tenured;
};

int main(){
  Person *p = new Person("Bill",1);
  Student *s = new Student("Joe",2,5);
  Faculty *f = new Faculty("Ken",3,0);
  Person *q;
  q = p; q->print_info();
  q = s; q->print_info();
  q = f; q->print_info();
  // calls print_info
  // for objected pointed to, not type of q
}
```

Name=Bill, ID=1
Name=Joe, ID=2, Major = 5
?

Polymorphism

- **Idea of polymorphism says that one set of code should operate appropriately (call appropriate functions of derived classes) on all derived types of objects**

```
int main()
{
  Person* p[5];
  p[0] = new Person("Bill",1);
  p[1] = new Student("Joe",2,5);
  p[2] = new Faculty("Ken",3,0);
  p[3] = new Student("Mary",4,2);
  p[4] = new Faculty("Jen",5,1);
  for(int i=0; i < 5; i++){
    p[i]->print_info();
    // should print most specific info
    // based on type of object
  }
}
```

Name=Bill, ID=1
Name=Joe, ID=2, Major = 5
Name = Ken, ID=3, Tenured=0
Name = Mary, ID=4, Major=2
Name = Jen, ID=5, Tenured=1

Virtual Destructors

```
class Student{
 ~Student() {  }
 string major();
 ...
}

class StudentWithGrades : public Student
{
 public:
  StudentWithGrades(...)
  { grades = new int[10]; }
  ~StudentWithGrades { delete [] grades; }
  int *grades;
}

int main()
{
  Student *s = new StudentWithGrades(...);
  cout << s->major();
  delete s; // What destructor gets called?
  return 0;
}
```

~Student() gets called and doesn't delete grades array

```
class Student{
 virtual ~Student() {  }
 string major();
 ...
}

class StudentWithGrades : public Student
{
 public:
  StudentWithGrades(...)
  { grades = new int[10]; }
  ~StudentWithGrades { delete [] grades; }
  int *grades;
}

int main()
{
  Student *s = new StudentWithGrades(...);
  cout << s->major();
  delete s; // What destructor gets called?
  return 0;
}
```

~StudentWithGrades() gets called and does delete grades array

- **Classes that will be used as a base class should have a virtual destructor**

Summary

- **No virtual declaration:**
  - **Member function that is called is based on the _____**
  - **Static binding**
- **With virtual declaration:**
  - **Member function that is called is based on the _____**
  - **Dynamic Binding**

Summary

- **No virtual declaration:**
  - **Member function that is called is based on the *type of the pointer/reference***
  - **Static binding**
- **With virtual declaration:**
  - **Member function that is called is based on the *type of the object pointed at (referenced)***
  - **Dynamic Binding**

Abstract Classes

- In software development we may want to create a base class that serves only as a requirement/interface that derived classes must implement/adhere to
- College students take tests and play sports so it makes sense to ensure that is defined for any type of CollegeStudent
  - But depending on which college you go to you may do these activities differently
  - But…until we know the university we don't know how to write take_test() and play_sports()…these are abstract
- Make this an abstract base class (i.e., interface for future derived classes)

```
class CollegeStudent {
 public:
  string get_name();
  virtual void take_test();
  virtual string play_sports();
 protected:
  string name;
};
```

Valid class. Objects of type CollegeStudent can be declared.

```
class CollegeStudent {
 public:
  string get_name();
  virtual void take_test() = 0;
  virtual string play_sports() = 0;
 protected:
  string name;
};
```

Abstract Base Class…No object of type CollegeStudent will be allowed. It only serves as an interface that derived classes will have to implement.

Abstract Classes (cont.)

- An abstract class is one that defined pure virtual functions
  - Prototype only
  - Make function body " = 0; "
  - Functions that are not implemented by the base class but must be implemented by the derived class
- No objects of the abstract type are allowed to be instantiated

```
class CollegeStudent {
 public:
  string get_name();
  virtual void take_test() = 0;
  virtual string play_sports() = 0;
 protected:
  string name;
};

class TrojanStudent : public CollegeStudent {
 public:
  void take_test() { cout << "Got an A."; }
  string play_sports(){return string("WIN!");}
};

class BruinStudent : public CollegeStudent {
  public:
  void take_test() { cout << "Uh..uh..C-."; }
  string play_sports(){return string("LOSE");}
};

int main() {
  vector<CollegeStudent *> mylist;
  mylist.push_back(new TrojanStudent());
  mylist.push_back(new BruinStudent());
  for(int i=0; i < 2; i++){
    mylist[i]->take_test();
    cout << mylist[i]->play_sports() << endl;
  }
  return 0;
}
```

Output:
Got an A. WIN!
Uh..uh..C-. LOSE

When to Use Inheritance

- **Main use of inheritance is to setup interfaces (abstract classes) that allow for new, derived classes to be written in the future that provide additional functionality but still works seamlessly with original code**

```
#include "student.h"
void sports_simulator(CollegeStudent *stu){
  ...
  stu->play_sports();
};
```

g++ -c sportsim.cpp
outputs sportsim.o (10 years ago)

```
#include "student.h"
class MITStudent : public CollegeStudent {
 public:
  void take_test() { cout << "Got an A+."; }
  string play_sports()
   { return string("What are sports?!?"); }
};

int main() {
  vector<CollegeStudent *> mylist;
  mylist.push_back(new TrojanStudent());
  mylist.push_back(new MITStudent());
  for(int i=0; i < 2; i++){
    sports_simulator(mylist[i]);
  }
  return 0;
}
```

g++ main.cpp sportsim.o
program will run fine today with new MITStudent

Abstract Classes

- No objects of the abstract type are allowed to be instantiated
- But the abstract base class can define common functions, have data members, etc. that all derived classes can use via inheritance
  - Ex. 'color' of the Animal

```
class Animal {
 public:
  Animal(string c) : color(c) { }
  virtual ~Animal()
  string get_color() { return c; }
  virtual void make_sound() = 0;
 protected:
  string color;
};
class Dog : public Animal {
 public:
  void make_sound() { cout << "Bark"; }
};
class Cat : public Animal {
  public:
  void make_sound() { cout << "Meow"; }
};
class Fox : public Animal {
  public:
  void make_sound() { cout << "???"; }
};  // derived class must define pure virtual
    // (even if you don't quite know what to do)
int main(){
  Animal* a[3];
  a[0] = new Animal;
     // WON'T COMPILE...abstract class
  a[1] = new Dog("brown");
  a[2] = new Cat("calico");
  cout << a[1]->get_color() << endl;
  cout << a[2]->make_sound() << endl;
}
```

Output:
brown
meow

A List Interface

- Consider the List Interface shown to the right
- This abstract class (contains pure virtual functions) allows many possible derived implementations
  - Linked List
  - Bounded Dynamic Array
  - Unbounded Dynamic Array
- Any derived implementation will have to conform to these public member functions

```
#ifndef ILISTINT_H
#define ILISTINT_H

class IListInt {
 public:
  virtual bool empty() const = 0;
  virtual int  size() const = 0;
  virtual void push_back(const int& new_val) = 0;
  virtual void insert(int newPosition,
                      const int& new_val) = 0;
  virtual void remove(int loc) = 0;
  virtual int const &   get(int loc) const = 0;
  virtual int&   get(int loc) = 0;
};

#endif
```

g++ main.cpp sportsim.o
program will run fine today with new MITStudent

Derived Implementations

- Consider the List Interface shown to the right
- This abstract class (contains pure virtual functions) allows many possible derived implementations
  - Linked List
  - Array
- Any derived implementation will have to conform to these public member functions

```
#ifndef ILISTINT_H
#define ILISTINT_H

class IListInt {
 public:
  virtual bool empty() const = 0;
  virtual int  size() const = 0;
  ...
};

#endif
```

ilistint.h

```
#include "ilistint.h"

class LListInt : public IListInt {
 public:
  bool empty() const { return head_ == NULL; }
  int  size() const { ... }
  ...
};
```

llistint.h

```
#include "ilistint.h"

class ArrayList : public IListInt {
 public:
  bool empty() const { return size_ == 0; }
  int  size() const { return size_; }
  ...
};
```

alistint.h

Usage

- Recall that to take advantage of dynamic binding you must use a base-class pointer or reference that points-to or references a derived object
- What's the benefit of this?

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

void fill_with_data(IListInt* mylist)
{
  for(int i=0; i < 10; i++){ mylist->push_back(i); }
}

void print_data(const IListInt& mylist)
{
  for(int i=0; i < mylist.size(); i++){
    cout << mylist.get(i) << endl;
  }
}

int main()
{
  IListInt* thelist = new AListInt();

  fill_with_data(thelist);

  print_data(*thelist);

  return 0;
}
```

Usage (cont.)

- What's the benefit of this?
  - We can drop in a different implementation WITHOUT changing any other code other than the instantiation!!!
  - Years later I can write a new List implementation that conforms to iList and drop it in and the subsystems [e.g., fill_with_data() and print_data()] should work just fine

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

void fill_with_data(IListInt* mylist)
{
  for(int i=0; i < 10; i++){ mylist->push_back(i); }
}

void print_data(const IListInt& mylist)
{
  for(int i=0; i < mylist.size(); i++){
    cout << mylist.get(i) << endl;
  }
}

int main()
{
//  IListInt* thelist = new AListInt();
  IListInt* thelist = new LListInt();

  fill_with_data(thelist);

  print_data(*thelist);

  return 0;
}
```

Another Exercise

- Consider a video game with a heroine who has a score and fights 3 different types of monsters {A, B, C}
- Upon slaying a monster you get a different point value:
  - 10 pts. = monster A
  - 20 pts. = monster B
  - 30 pts. = monster C
- You can check if you've slayed a monster via an 'isDead()' call on a monster and then get the value to be added to the heroine's score via 'getScore()'
- The game keeps objects for the heroine and the monsters
- How would you organize your Monster class(es) and its data members?

Using Type Data Member

- Can use a 'type' data member and code
- Con: Adding new monster types requires modifying Monster class code as does changing point total

```
class Player {
  public:
    int addToScore(int val) { _score += val; }
  private:
    int _score;
};

class Monster {
 public:
  Monster(int type) : _type(type) {}
  bool isDead();  // returns true if the monster is dead
  int getValue() {
    if(_type == 0) return 10;
    else if(_type == 1) return 20;
    else return 30;
  }
 private:
  int _type; // 0 = A, 1 = B, 2 = C
};

int main()
{
  Player p;
  int numMonsters = 10;
  Monster** monsters = new Monster*[numMonsters];
  // init monsters of various types
  ...
  while(1){
    // Player action occurs here
    for(int i=0; i < numMonsters; i++){
      if(monsters[i]->isDead())
        p.addToScore(monserts[i]->getValue())
    }
  }
}
```

Using Score Data Member

- Can use a 'value' data member and code
- Pro: Monster class is now decoupled from new types or changes to point values

```
class Player {
  public:
    int addToScore(int val) { _score += val; }
  private:
    int _score;
};

class Monster {
 public:
  Monster(int val) : _value(val) { }
  bool isDead();
  int getValue() {
    return _value;
 }
 private:
  int _value;
};

int main()
{
  Player p;
  int numMonsters = 10;
  Monster** monsters = new Monster*[numMonsters];
  monsters[0] = new Monster(10); // Type A Monster
  monsters[1] = new Monster(20); // Type B Monster
  ...
  while(1){
    // Player action occurs here
    for(int i=0; i < numMonsters; i++){
      if(monsters[i]->isDead())
        p.addToScore(monserts[i]->getValue())
    }
  }
}
```

Using Inheritance

- Go back to the requirements:
  - "Consider a video game with a heroine who has a score and fights 3 different types of monsters {A, B, C}"
  - Anytime you see 'types', 'kinds', etc. an inheritance hierarchy is probably a viable and good solution
  - Anytime you find yourself writing big if..elseif...else statement to determine the type of something, inheritance hierarchy is probably a good solution
- Usually prefer to distinguish types at creation and not in the class itself

```
class Player {
  public:
    int addToScore(int val) { _score += val; }
  private:
    int _score;
};

class Monster {
 public:
  Monster(int val) : _value(val) { }
  bool isDead();
  int getValue() {
    return _value;
 }
 private:
  int _value;
};

int main()
{
  Player p;
  int numMonsters = 10;
  Monster** monsters = new Monster*[numMonsters];
  monsters[0] = new Monster(10); // Type A Monster
  monsters[1] = new Monster(20); // Type B Monster
  ...
  while(1){
    // Player action occurs here
    for(int i=0; i < numMonsters; i++){
      if(monsters[i]->isDead())
        p.addToScore(monserts[i]->getValue())
    }
  }
}
```

Using Polymorphism (cont.)

- So sometimes seeding an object with different data values allows the polymorphic behavior
- Other times, data is not enough...code is needed
- Consider if the score of a monster is not just hard coded based on type but type and other data attributes
  - If Monster type A is slain with a single shot your points are multiplied by the base score and their amount of time they are running around on the screen
  - However, Monster type B alternates between berserk mode and normal mode and you get different points based on what mode they are in when you slay them

```
class Player {
  public:
    int addToScore(int val) { _score += val; }
  private:
    int _score;
};

class Monster {
 public:
  Monster(int val) : _value(val) { }
  bool isDead();
  int getValue() {
    return _value;
 }
 private:
  int _value;
};

int main()
{
  Player p;
  int numMonsters = 10;
  Monster** monsters = new Monster*[numMonsters];
  monsters[0] = new Monster(10); // Type A Monster
  monsters[1] = new Monster(20); // Type B Monster
  ...
  while(1){
    // Player action occurs here
    for(int i=0; i < numMonsters; i++){
      if(monsters[i]->isDead())
        p.addToScore(monserts[i]->getValue())
    }
  }
}
```

Using Polymorphism (cont.)

- Can you just create different classes?
- Not really, can't carry them around in a single container/array

```
class MonsterA {
 public:
  bool isDead();
  int getValue()
  {
    // code for Monster A with multipliers & head shots
  }
};

class MonsterB {
 public:
  bool isDead();
  int getValue()
  {
    // code for Monster B with berserker mode, etc.
  }
};

int main()
{
  Player p;
  int numMonsters = 10;
  // can't have a single array of "Monsters"
  // Monster** monsters = new Monster*[numMonsters];

  // Need separate arrays:
  MonsterA* monsterAs = new MonsterA*[numMonsters];
  MonsterB* monsterBs = new MonsterB*[numMonsters];
```

Using Polymorphism (cont.)

- Will this work?
- No, static binding!!
  - Will only call Monster::getValue() and never MonsterA::getValue() or MonsterB::getValue()

```
class Monster {
  int getValue()
  {
    // generic code
  }
};
class MonsterA : public Monster {
 public:
  bool isDead();
  int getValue()
  {
   // code for Monster A with multipliers & head shots
  }
};

class MonsterB : public Monster {
 public:
  bool isDead();
  int getValue()
  {
    // code for Monster B with berserker mode, etc.
  }
};

int main()
{
  Player p;
  int numMonsters = 10;

  Monster** monsters = new Monster*[numMonsters];
  // now try to create and store MonsterA's and B's in this
  // array
};
```

Using Polymorphism (cont.)

- Will this work?
- Yes!!
- Now I can add new Monster types w/o changing any Monster classes
- Only the creation code need change

```
class Monster {
  bool isDead(); // could be defined once for all monsters
  virtual int getValue() = 0;
};

class MonsterA : public Monster {
 public:
  int getValue()
  {
    // code for Monster A with multipliers & head shots
  }
};

class MonsterB : public Monster {
 public:
  int getValue()
  {
    // code for Monster B with berserker mode, etc.
  }
};

int main()
{
  Monster** monsters = new Monster*[numMonsters];
  monsters[0] = new MonsterA; // Type A Monster
  monsters[1] = new MonsterB; // Type B Monster
  ...
  while(1){
    // Player action occurs here
    for(int i=0; i < numMonsters; i++){
      if(monsters[i]->isDead())
        p.addToScore(monserts[i]->getValue())
  } }
  return 0;
}
```