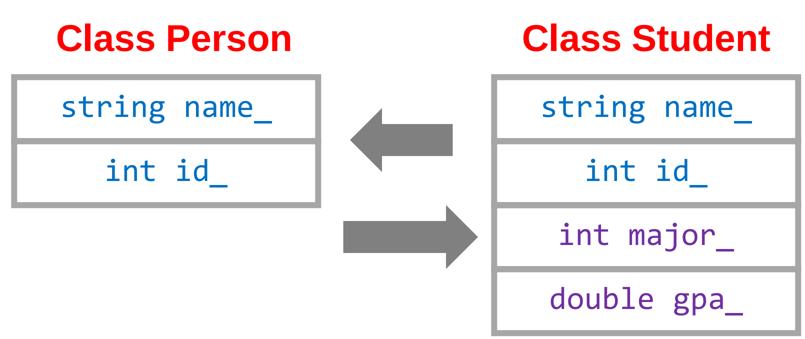
CSCI 104 Polymorphism

Mark Redekopp
David Kempe

Assignment of Base/Derived

- Can we assign a derived object into a base object?
- Can we assign a base object into a derived?
  - p = s; // Base = Derived…_________
  - s = p; // Derived = Base…_________
- Think hierarchy & animal classification?
  - Can any dog be (assigned as) a mammal
  - Can any mammal be (assigned as) a dog
- We can only assign a derived into a base (since the derived has EVERYTHING the base does)
- For pointers and references, a base pointer/reference can point to/reference a derived object!
  - Student s;
  - Person* pptr = &s;

```
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;
}
```



Review Questions 1

- As we call processPerson(&p) what member functions will be called (e.g. Person::print_info, CSStudent::useComputer, etc.)
- As we call processPerson(&s)?
- As we call processPerson(&cs)?
- We use the terms static and dynamic binding when referring to which function will be called when virtual is NOT or IS present.

```
class Person {
 public:
  virtual void print_info() const; // name, ID
  void useComputer(); // stream a show
  string name; int id;
};
class Student : public Person {
 public:
  void print_info() const; // print major
  void useComputer(); // write a paper
  int major; double gpa;
};
class CSStudent : public Student {
 public:
  void print_info() const; // print OH queue pos
  void useComputer(); // fight with Docker
};

void processPerson(Person* p)
{ p->print_info();
  p->useComputer(); }

int main(){
  Person p(...);      processPerson(&p);
  Student s(...);     processPerson(&s);
  CSStudent cs(...); processPerson(&cs);
  // more
}
```

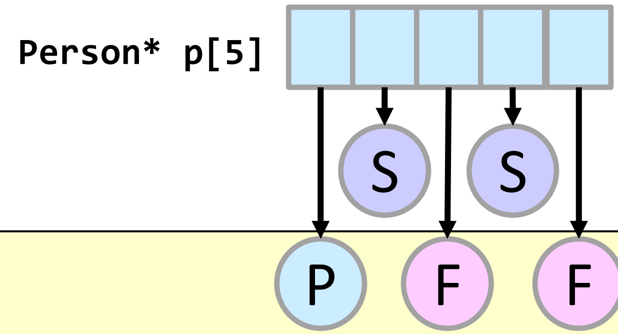
Review Questions 2

- What does "=0;" mean in the declarations to the right?
- What do we call a class with 1 or more of these kind of declarations?
- Is it okay that Student doesn't provide a useComputer() implementation?
- Can we declare Person objects?
- Can we declare pointers or references to Person objects?
- When should a class have a virtual destructor?

```
class Person {
 public:
  virtual void print_info() const = 0;
  virtual void useComputer(); // stream a show
  string name; int id;
};
class Student : public Person {
 public:
  void print_info() const; // print major
  int major; double gpa;
};
class CSStudent : public Student {
 public:
  void print_info() const; // print OH queue pos
  void useComputer(); // fight with Docker
};

void printPerson(Person* p) { p->print_info(); }
void compute(Person& p)     {  p.useComputer(); }

int main(){
  Person p(...);  // Allowed?
  Student s(...);    useComputer(s);
  CSStudent cs(...); printPerson(&cs);
  // more
}
```

Benefits of Polymorphism

- Can we have an array that store multiple types (e.g. an array that stores both ints and doubles)? No!
- Use base pointers to point at different types and have their individual behavior invoked via virtual functions
- Polymorphism via virtual functions allows one set of code to operate appropriately on all derived types of objects
- ***One data structure can now reference many types and the code can perform appropriate behavior on each as you iterate over the structure***




```
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
```

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
```

Derived Implementations

- Consider the List Interface shown to the right
- This abstract class (contains pure virtual functions) allows many possible derived implementations
  - Linked List
  - Static 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;
  ...
};

#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

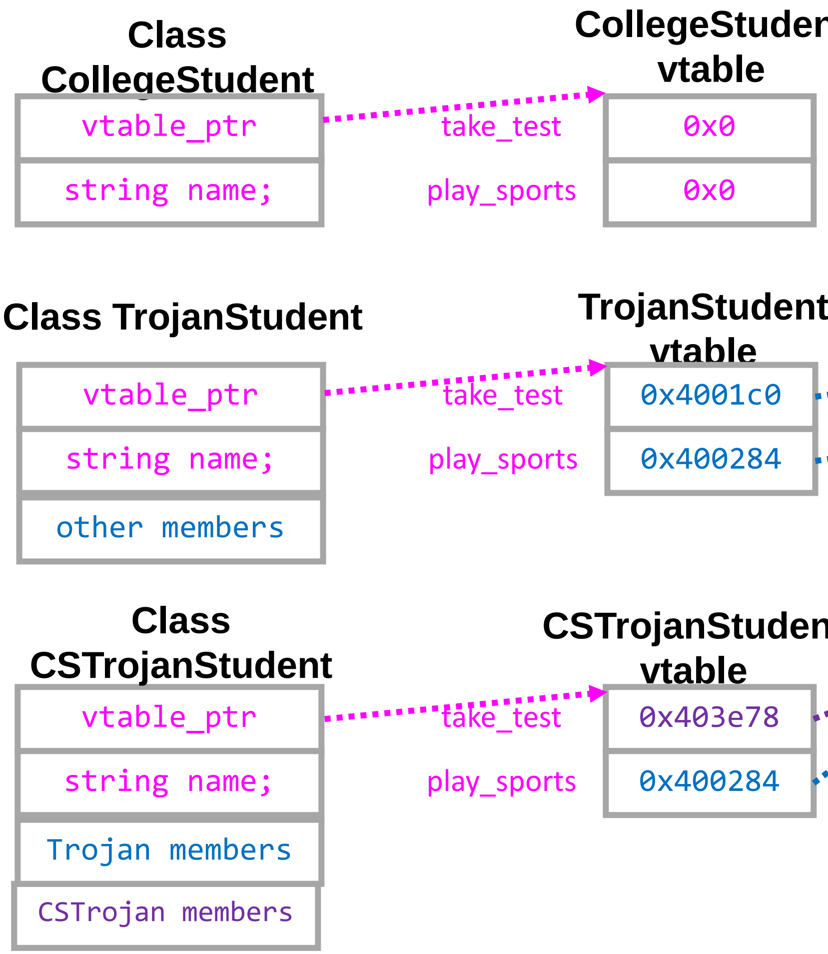
- What's the benefit of this?
  - We can drop in a different implementation WITHOUT changing any other code other than the instantiation!!!
  - Years later I can write a new List implementation that conforms to IList and drop it in and the subsystems [e.g. fill_with_data() and print_data()] should work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

```
#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;
}
```

How polymorphism works under the hood

VTABLES AND VPTRS

VTables

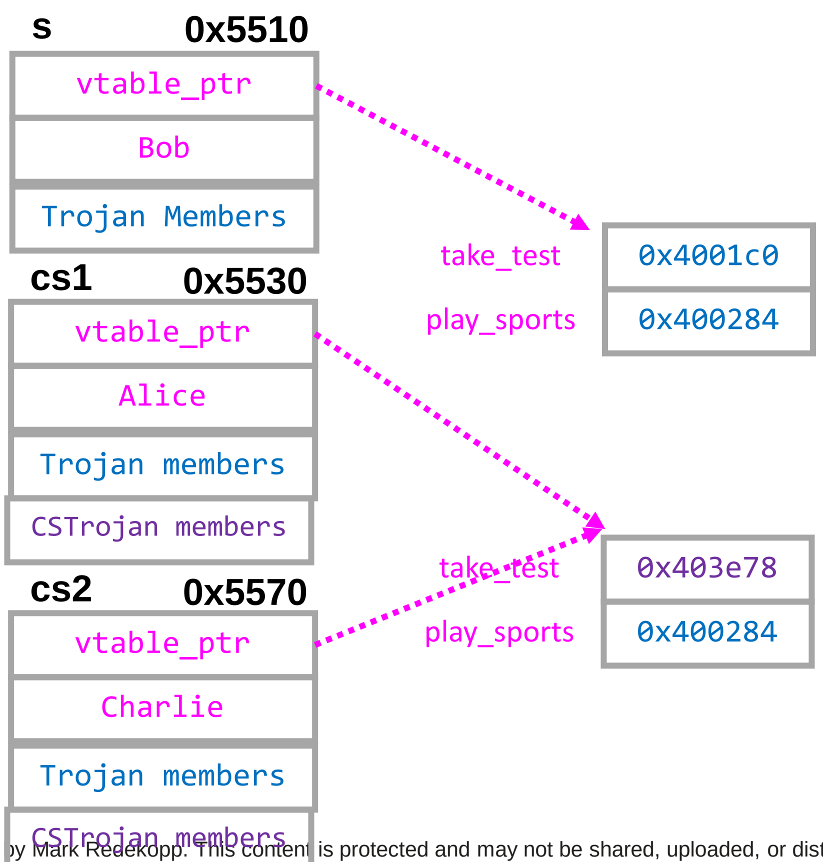



```
class CollegeStudent {
 public:
  string get_name() { return name; }
  virtual void take_test() = 0;
  virtual string play_sports() = 0;
 protected:
  string name;
};
class TrojanStudent : public CollegeStudent {
 public:
  void take_test() { cout << "Got an A."; }
  string play_sports(){return string("WIN!");}
};
class CSTrojanStudent : public TrojanStudent {
  public:
  void take_test() { cout << "A...curved"; }
};
```

- Compiler creates a table for each class with an entry for each virtual function (aka **vtable**).
- Each entry points to the appropriate function code to call
- Each object has an extra data member (vptr) that points to the vtable for its class.

Example of Calling Virtual Functions

- Calling a non-virtual function, always goes to the same code (known **at compile time/statically**)
- Calling a virtual function, requires following the vtable ptr at **runtime (dynamically)** to find the correct function to call




```
class CollegeStudent {
 public:
  string get_name() { return 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 CSTrojanStudent : public TrojanStudent {
  public:
  void take_test() { cout << "A...curved"; }
};

void f1(CollegeStudent* s) {
   cout << s->get_name() << " test result: ";
   s->take_test();
   s->play_sports();
}
int main()
{
  TrojanStudent s("Bob");          f1(&s);
  CSTrojanStudent cs1("Alice");    f1(&cs1);
  CSTrojanStudent cs2("Charlie"); f1(&cs2);
  return 0;
}
```

OO DESIGN PRINCIPLES

General OO Design Goal

- Loose Coupling: A relationship between objects where changes in one component do not require (or reduced the need for) changes in others.
- Examples:
  - A USB device is loosely coupled with your laptop whereas your processor is tightly coupled
  - A car's battery is loosely coupled with the vehicle whereas the engine is tightly coupled
- To achieve loose coupling we have principles that we often try to follow in our software design.

OO Design Principles

- Single-Responsibility
  - A class (or even a function) should generally have only one responsibility (e.g. a product, a user, a search engine, etc.)
- Open/closed rule:
  - A class should be **open to extension** but **closed to modification**. A class should be designed so that its behavior can be changed through inheritance/polymorphism, not modification.
- These are a few principles from what some developers refer to as the 5 SOLID principles
  - Feel free to search online for more readings. There's not one agreed upon set of principles and even how various principles are applied may be a subject of debate.
- For C++ OO implementation guidelines:
  - https://isocpp.org/faq – Scroll to "Classes and Inheritance" section

Class hierarchies with low coupling (if time permits)

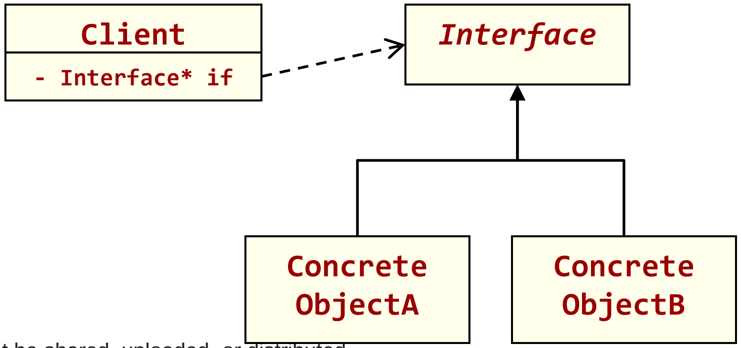
SPECIFIC DESIGN PATTERNS

Design Patterns

- Common software practices to create modular code
  - Often using inheritance and polymorphism
- Researchers studied software development processes and actual code to see if there were common patterns that were often used
  - Most well-known study resulted in a book by four authors affectionately known as the "Gang of Four" (or GoF)
    - <u>Design Patterns: Elements of Reusable Object-Oriented Software</u> by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
- Creational Patterns
  - Singleton, Factory Method, Abstract Factory, Builder, Prototype
- Structural Patterns
  - Adapter, Façade, Decorator, Bridge, Composite, Flyweight, Proxy
- Behavioral Patterns
  - Iterator, Mediator, Chain of Responsibility, Command, State, Memento, Observer, Template Method, Strategy, Visitor, Interpreter

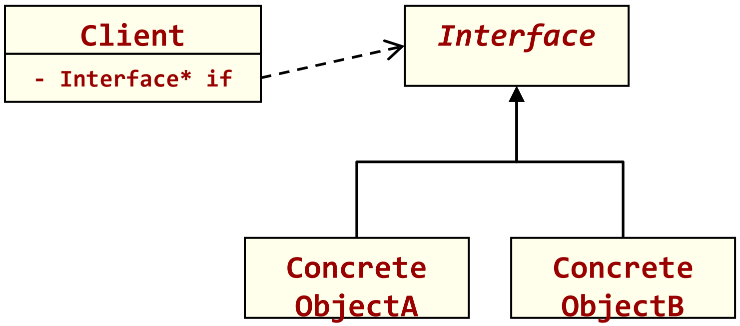
Understanding UML Relationships

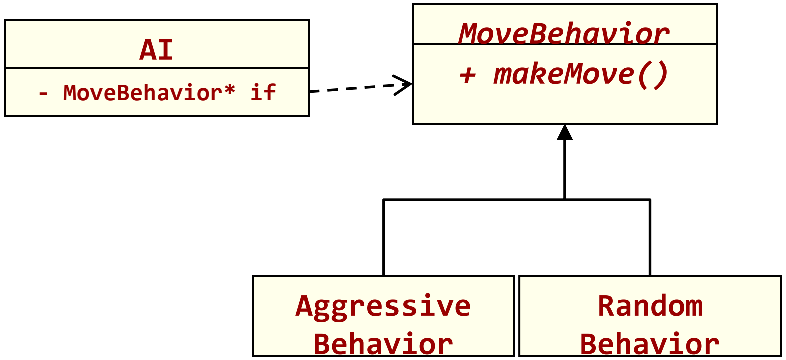
- UML (Unified Modeling Language) is often used to depict software designs and object relationships
  - https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/
  - UML can be very detailed and specific, whereas we may often use a basic subset to communicate our SW design
- We'll generally just use generic inheritance and composition relationships:



Strategy

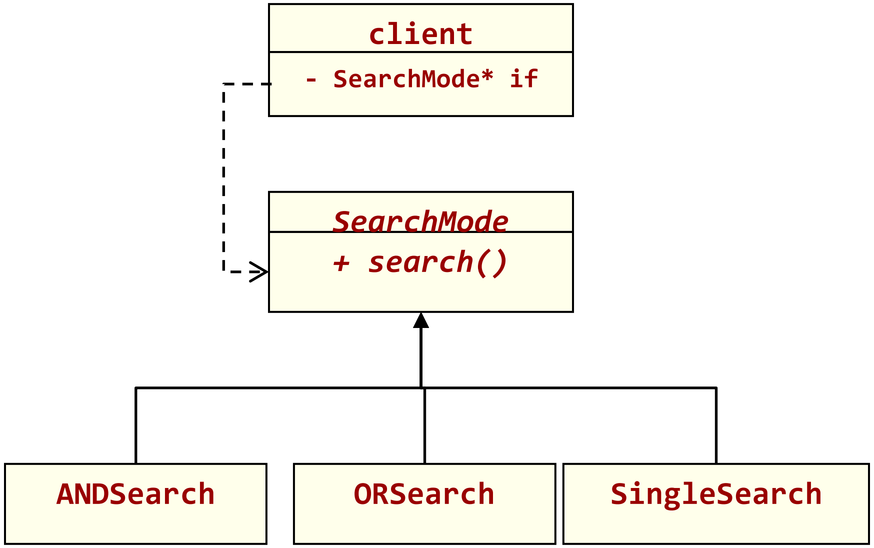
- Abstracting interface to allow alternative approaches
- Fairly classic polymorphism idea
- In a video game the AI may take different strategies
  - Decouples AI logic from how moves are chosen and provides for alternative approaches to determine what move to make
- Recall "Shapes" example/exercise from class/lab
  - Program that dealt with abstract shape class rather than concrete rectangles, circles, etc.
  - The program could now deal with any new shape provided it fit the interface





Your Search Engine

- Think about your class project and where you might be able to use the strategy pattern
- AND, OR, Normal Search

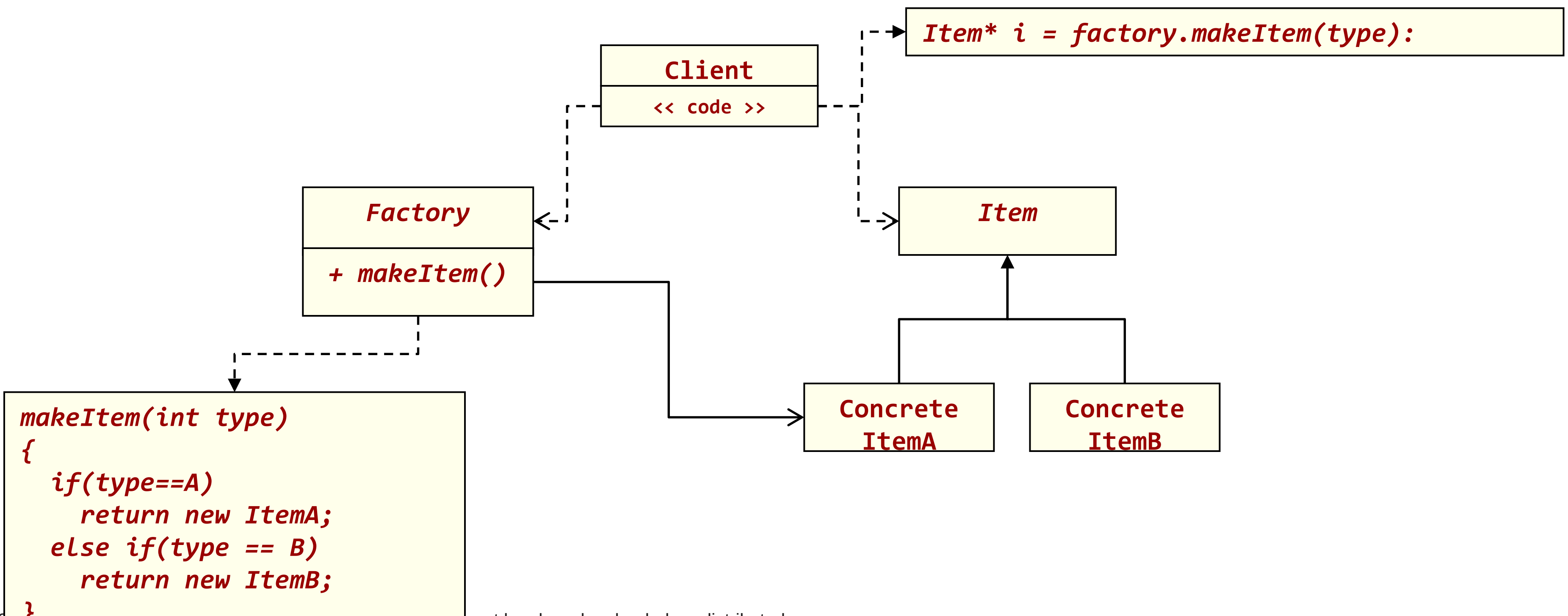



```
string searchType;
string searchWords;

cin >> sType;
SearchMode* s;
if(sType == "AND"){
  s = new ANDSearch;
}
else if(sType == "OR")
{
  s = new ORSearch;
}
else {
  s = new SingleSearch;
}

getline(cin, searchWords);
s->search(searchWords);
```

Client

Factory Pattern

- A function, class, or static function of a class used to abstract creation
- Rather than making your client construct objects (via 'new', etc.), abstract that functionality so that it can be easily extended without affecting the client



Factory Example

- We can pair up our search strategy objects with a factory to allow for easy creation of new approaches

Factory

```
class SearchFactory{
 public:
   static SearchMode* create(string type)
   {
     if(type == "AND")
       return new ANDSearch;
     else if(type == "OR")
       return new ORSearch;
     else
       return new SingleSearch;
   }
};
```

Client

```
string sType;
string searchWords;

cin >> sType;
SearchMode* s = SearchFactory::create(sType);

getline(cin, searchWords);
s->search(searchWords);
```

Search Interface

```
class SearchMode {
  public:
   virtual search(set<string> searchWords) = 0; ...
};
```

Concrete Search

```
class AndSearch : public SearchMode
{
  public:
   search(set<string> searchWords){
     // perform AND search approach
   }
   ...
};
```

Factory Example

- The benefit is now I can add new search modes without the client changing or even recompiling

```
class SearchFactory{
 public:
   static SearchMode* create(string type)
   {
     if(type == "AND")
       return new ANDSearch;
     else if(type == "OR")
       return new ORSearch;
     else if(type == "DIFF")
       return new DIFFSearch;
     else
       return new SingleSearch;
   }
};
```

```
string sType;
string searchWords;

cin >> sType;
SearchMode* s = SearchFactory::create(sType);

getline(cin, searchWords);
s->search(searchWords);
```

```
class DIFFSearch : public SearchMode
{
  public:
   search(set<string> searchWords);
   ...
};
```

Iterator

- Decouples organization of data in a collection from the client who wants to iterate over and access just the data
  - Data could be in a BST, linked list, or array
  - Client just needs to…
    - Allocate an iterator [it = collection.begin()]
    - Dereferences the iterator to access data [*it]
    - Increment/decrement the iterator [++it]

On Your Own

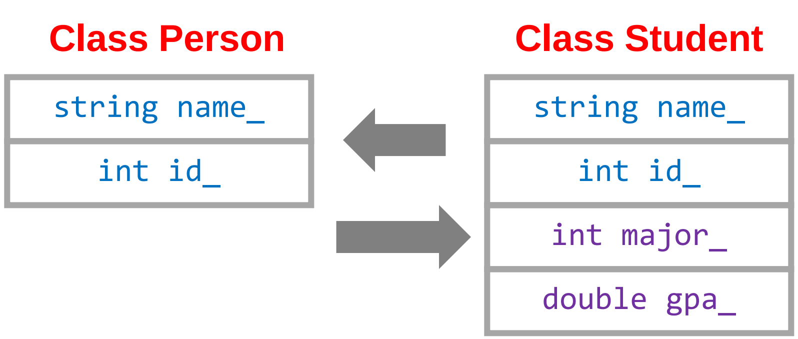
- Design Patterns
  - Observer
  - Proxy
  - Template Method
  - Adapter
- Questions to try to answer
  - How does it make the design more modular (loosely coupled)
  - When/why would you use the pattern
- Resources
  - http://sourcemaking.com/
  - http://www.vincehuston.org/dp/
  - http://www.oodesign.com/

PRE SUMMER 2021 POLYMORPHISM SLIDES

Assignment of Base/Derived

- Can we assign a derived object into a base object?
- Can we assign a base object into a derived?
  - p = s; // Base = Derived…_________
  - s = p; // Derived = Base…_________
- Think hierarchy & animal classification?
  - Can any dog be (assigned as) a mammal
  - Can any mammal be (assigned as) a dog
- We can only assign a derived into a base (since the derived has EVERYTHING the base does)

```
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;
}
```



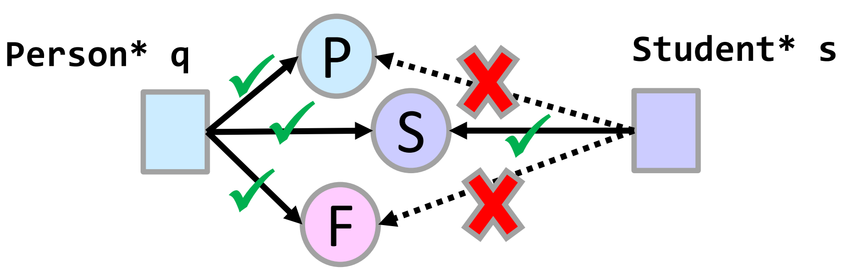
USC Viterbi School of Engineering

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 from the class corresponding to ***the type of the pointer used***

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

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

class Faculty : public Person {
 public:
  void print_info() const; // 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();
} // calls
```

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



Base pointer CAN point at any publicly derived object

Derived pointer CANNOT point at base or "sibling" objects

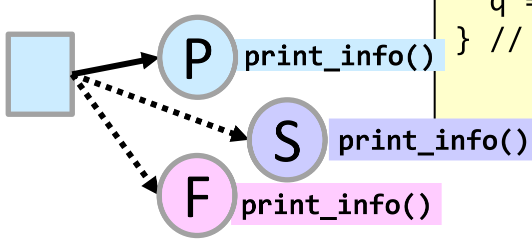
Inheritance

- For second and third call to print_info() we might like to have `Student::print_info()` and `Faculty::print_info()` executed since the actual object pointed to is a Student/Faculty
- BUT…it will call `Person::print_info()`
- This is called '**static binding**' (i.e. the version of the function called is based on the static **type of the pointer** being used)

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

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

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

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

1 **No VIRTUAL declaration…**

Person* **q**



2 **…only functions from the class type of the pointer used can be called**

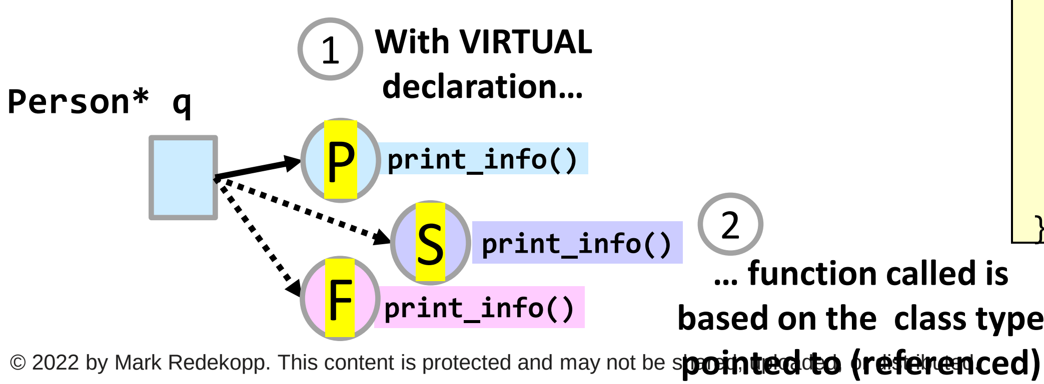
Name=Bill, ID=1
Name=Joe, ID=2
Name=Ken, ID=3

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 **type of object pointed to/referenced** rather than the type of pointer/reference
  - **Note**: You do NOT have to override a virtual function in the derived class…you can just inherit and use the base class version
- This is called 'dynamic binding' (i.e. which version is called is based on the type of object being pointed to)

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

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

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

int main(){
  Person *p = new Person("Bill",1);
  Student *s = new Student("Joe",2,5);
  Faculty *f = new Faculty("Mary",3,1);
  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

}
```

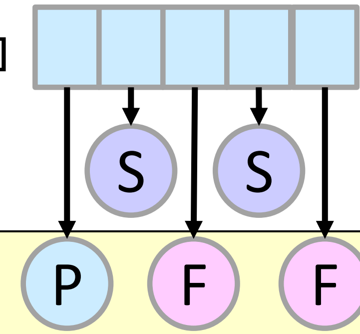
```
Name=Bill, ID=1
Name=Joe, ID=2, Major = 5
Name=Mary, ID=3, Tenured=1
```



Polymorphism

- Can we have an array that store multiple types (e.g. an array that stores both ints and doubles)? No!
- Use base pointers to point at different types and have their individual behavior invoked via virtual functions
- Polymorphism via virtual functions allows one set of code to operate appropriately on all derived types of objects
- ***One data structure can now reference many types and the code can perform appropriate behavior on each as you iterate over the structure***

`Person* p[5]`




```
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
```

Pointers, References, and Objects

- To allow dynamic binding and polymorphism you use a base class
  – **Pointer**
  – **Reference**
- **Copying a derived object** to a base object makes a copy and so no polymorphic behavior is possible

```
void f1(Person* p)
{
   p->print_info();
   // calls Student::print_info()
}

void f2(const Person& p)
{
   p.print_info();
   // calls Student::print_info()
}

void f3(Person p)
{
   p.print_info();
   // calls Person::print_info() on the copy
}

int main(){
  Student s("Joe",2,5);
  f1(&s);
  f2(s);
  f3(s);
  return 0;
}
```

Class Person

string name_
int id_

Class Student

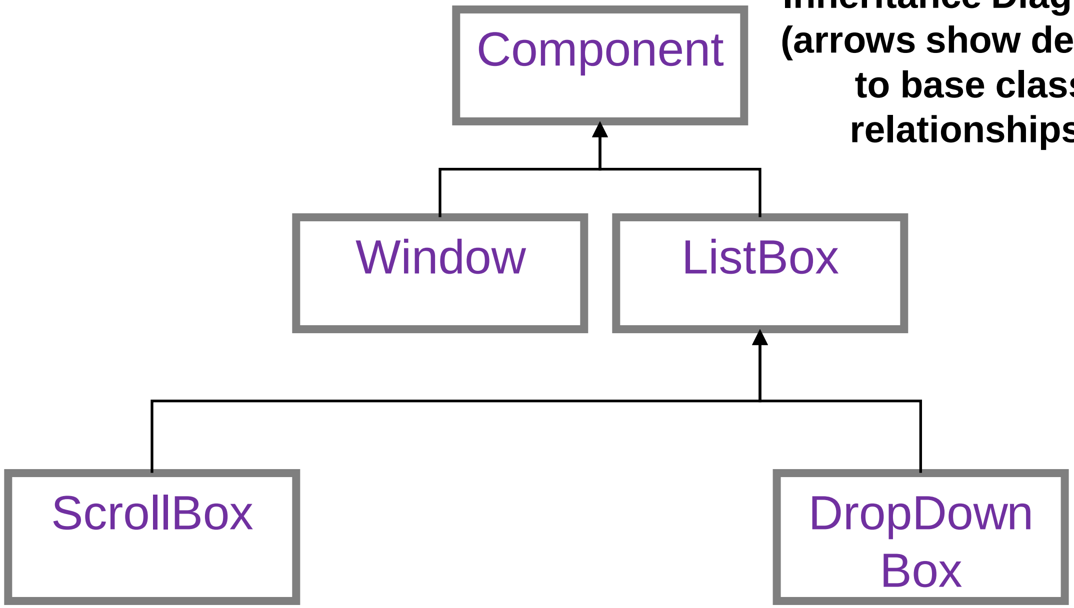
string name_
int id_
int major_
double gpa_

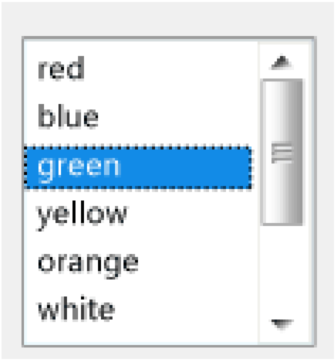
```
Name=Joe, ID=2, Major = 5
Name=Joe, ID=2, Major = 5
Name=Joe, ID=2
```

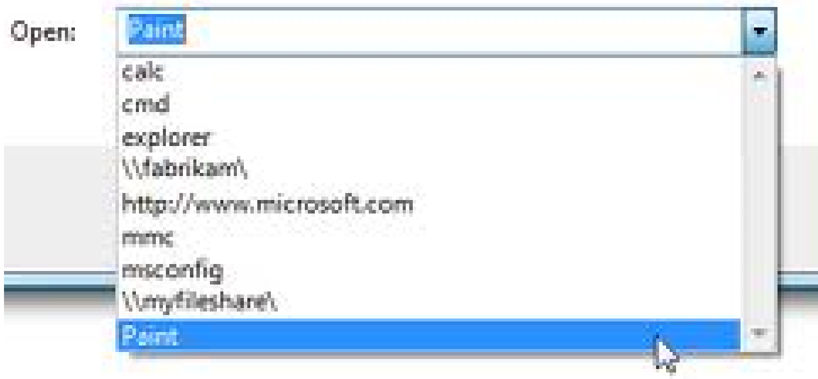
Inheritance/Polymorphism Example

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

Inheritance Diagrams (arrows show derived to base class relationships)







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(...);
  ...
  delete s; // Which destructor gets called?
  return 0;
}
```

Due to static binding (no virtual decl.) ~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(...);
  ...
  delete s; // Which destructor gets called?
  return 0;
}
```

Due to dynamic binding (virtual decl.) ~StudentWithGrades() gets called and does delete grades array

- **Classes that have at least 1 virtual function should have a virtual destructor** (http://www.parashift.com/c++-faq-lite/virtual-functions.html#faq-20.7)

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 & Pure Virtuals

- In software development we may want to create a base class that serves only as a requirement/interface that derived classes must implement and adhere to
- Example:
  - Suppose we want to create a CollegeStudent class and ensure all derived objects implement behavior for the student to take a test and play sports
  - But depending on which college you go to you may do these activities differently. Until we know the university we don't know how to implement take_test() and play_sports()…
  - We can decide to NOT implement them in this class known as "pure" virtual functions (indicated by setting their prototype =0;)
- A class with pure virtuals is called 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 with 2 pure virtual functions. No object of type CollegeStudent will be allowed. It only serves as an interface that derived classes will have to implement.

Abstract Classes & Pure Virtuals

- An abstract base class is one that defines at least 1 or more 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** to be able to create an instance of the derived object
- Objects of the abstract class type MAY NOT be declared/instantiated
  - Doing so would not be safe since some functions are not implemented

```
class CollegeStudent {
 public:
  string get_name() { return 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
```

How Long is a Class Abstract?

- Objects of the abstract class type MAY NOT be declared/instantiated
  - Doing so would not be safe since some functions are not implemented
- Until each pure virtual function has a definition the class stays abstract (see TrojanStudent to the right)

```
class CollegeStudent {
 public:
  string get_name() { return name; }
  virtual void take_test() = 0;
  virtual string play_sports() = 0;
 protected:
  string name;
};
class TrojanStudent : public CollegeStudent {
 public:
   string play_sports(){return string("WIN!");}
};
class CSTrojanStudent : public TrojanStudent {
  public:
  void take_test() { cout << "A...curved"; }
};
int main() {
  CollegeStudent cs1;
      // WON'T COMPILE
      // CollegeStudent is abstract
  TrojanStudent ts1;
      // WON'T COMPILE
      // TrojanStudent is still abstract

  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

- An abstract base class can still 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 << "???"; }
};
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
```

USC Viterbi School of Engineering

Derived Implementations

- Consider the List Interface shown to the right
- This abstract class (contains pure virtual functions) allows many possible derived implementations
  - Linked List
  - Static 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;
  ...
};

#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

- 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 as is.

```
#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;
}
```

Polymorphism & Private Inheritance

- **Warning**: If private or protected inheritance is used, the derived class is no longer type-compatible with base class
  - Can't have a base class pointer reference a derived object
- Example to the right
  - Person* can no longer point at Faculty
- Another example
  - Given: `class FIFO : private List`
  - Can NOT do the following:
  - `List * p = new FIFO();`

```
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;
};
// if we use private inheritance
// for some reason
class Faculty : private Person {
 public:
  void print_info(); // print tenured
  bool tenure;
};
int main()
{
  Person *q;
  Student* s = new Student("Joe",2,5);
  Faculty* f = new Faculty("Ken",3,0);
  q = s; q->print_info(); // works
  q = f; q->print_info(); // won't work!!!
  f->print_info();        // works
}
```

ANOTHER EXAMPLE

A Game of Monsters

- 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 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())
    }
  }
}
```

Is Polymorphism Needed?

- 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 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

- Can you just create different classes?
- Not really, can't store 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

- 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

- Will this work?
- Yes, Dynamic binding!!
- 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;
}
```

Deeper Investigation

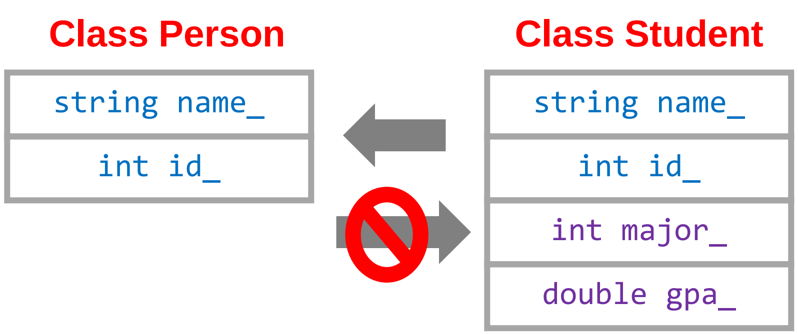
- Design patterns: Common OO structures and uses of inheritance/polymorphism
  - https://sourcemaking.com/design_patterns
- Open/closed rule:
  - http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- General guidelines and FAQ
  - https://isocpp.org/faq – Scroll to "Classes and Inheritance" section

SOLUTIONS

Assignment of Base/Derived

- Can we assign a derived object into a base object?
- Can we assign a base object into a derived?
- Think hierarchy & animal classification?
  - Can any dog be (assigned as) a mammal
  - Can any mammal be (assigned as) a dog
- 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;
}
```



Review Questions 1

- As we call processPerson(&p) what member functions will be called (e.g. Person::print_info, CSStudent::useComputer, etc.)
  - Person::print_info() / Person::useComputer()
- As we call processPerson(&s)?
  - Student::print_info() / Person::useComputer()
- As we call processPerson(&cs)?
  - CSStudent::print_info() / Person::useComputer()
- We use the terms static and dynamic binding when referring to which function will be called when virtual is NOT or IS present.

```
class Person {
 public:
  virtual void print_info() const; // name, ID
  void useComputer(); // stream a show
  string name; int id;
};
class Student : public Person {
 public:
  void print_info() const; // print major
  void useComputer(); // write a paper
  int major; double gpa;
};
class CSStudent : public Person {
 public:
  void print_info() const; // print OH queue pos
  void useComputer(); // fight with Docker
};

void processPerson(Person* p)
{ p->print_info();
  p->useComputer(); }

int main(){
  Person p(...);     processPerson(&p);
  Student s(...);    processPerson(&s);
  CSStudent cs(...); processPerson(&cs);
  // more
}
```

Review Questions 2

- What does "=0;" mean in the declarations to the right?
  - Pure virtual function
- What do we call a class with 1 or more of these kind of declarations?
  - Abstract class
- Is it okay that Student doesn't provide a useComputer() implementation?
  - Yes, it inherits Person::useComputer()
- Can we declare Person objects? No
- Can we declare pointers or references to Person objects? Yes
- When should a class have a virtual destructor?
  - When at least one other virtual function is declared in the class

```
class Person {
 public:
  virtual void print_info() const = 0;
  virtual void useComputer(); // stream a show
  string name; int id;
};
class Student : public Person {
 public:
  void print_info() const; // print major
  int major; double gpa;
};
class CSStudent : public Person {
 public:
  void print_info() const; // print OH queue pos
  void useComputer(); // fight with Docker
};

void printPerson(Person* p) { p->print_info(); }
void compute(Person& p)     {  p.useComputer(); }

int main(){
  Person p(...);  // Allowed?
  Student s(...);    useComputer(s);
  CSStudent cs(...); printPerson(&cs);
  // more
}
```