CSCI 104
Polymorphism

Mark Redekopp
David Kempe

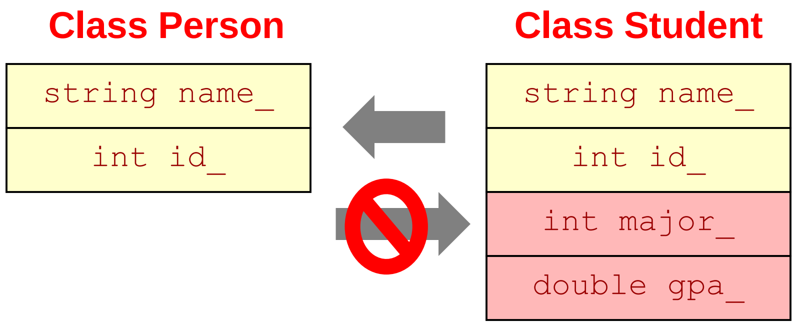
Virtual functions, Abstract classes, and Interfaces

POLYMORPHISM

Assignment of Base/Declared

- 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;
}
```



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
- BUT…it will call Person::print_info()
- 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 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 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 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
```

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

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** (http://www.parashift.com/c++-faq-lite/virtual-functions.html#faq-20.7)

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

- 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() { 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
```

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 << "???"; }
};
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
  - 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 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;
}
```

Polymorphism & Private Inheritance

- If for some reason you use private inheritance, then the derived class is no longer type compatible with the base class
  – Can't have a base class pointer reference a derived object

```
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(){
Student *s = new Student("Joe",2,5);
  Faculty *f = new Faculty("Ken",3,0);
  Person *q;
  q = s; q->print_info();
  q = f; q->print_info(); // won't work!!!
  f->print_info();        // works
}
```

Exercise

- Download the skeleton file
- $ wget http://ee.usc.edu/~redekopp/cs104/shapes.cpp
- Examine the given code:
  - Examine the abstract base class declaration of Shape
  - Examine the derived implementation of a RightTriangle class
  - Examine the main() function that implements an iterative menu selection process for users to create Shapes and add them to a list (vector) of Shape *'s
  - After exiting the menu, the Shapes in the list will be printed out along with their perimeter and area.
- Write classes to model a Rectangle, Square, and Circle taking in appropriate parameters in the constructor [see the menu output and comments to infer what those parameters/data member should be] and implementing appropriate member functions.
- Add code in the main()'s if…else if statements to allocate appropriate objects and enter them into the shapeList
- Compile, run and test the program (sample input and output is shown below).
- Debug any errors.

Exercise

- Sample input and output:

  Inputs:

  1 3 4 (Right triangle with b=3, h=4)

  2 3 4 (Rectangle with b=3, h=4)

  3 4 (Square with side = 4)

  4 2 (Circle with radius = 2)

  0 (Quit and print)

  Outputs:

  Right Triangle: Area=6 Perim=12

  Rectangle: Area=12 Perim=14

  Square: Area=16 Perim=16

  Circle: Area=12.5664 Perim=12.5664

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

- 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

- 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

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

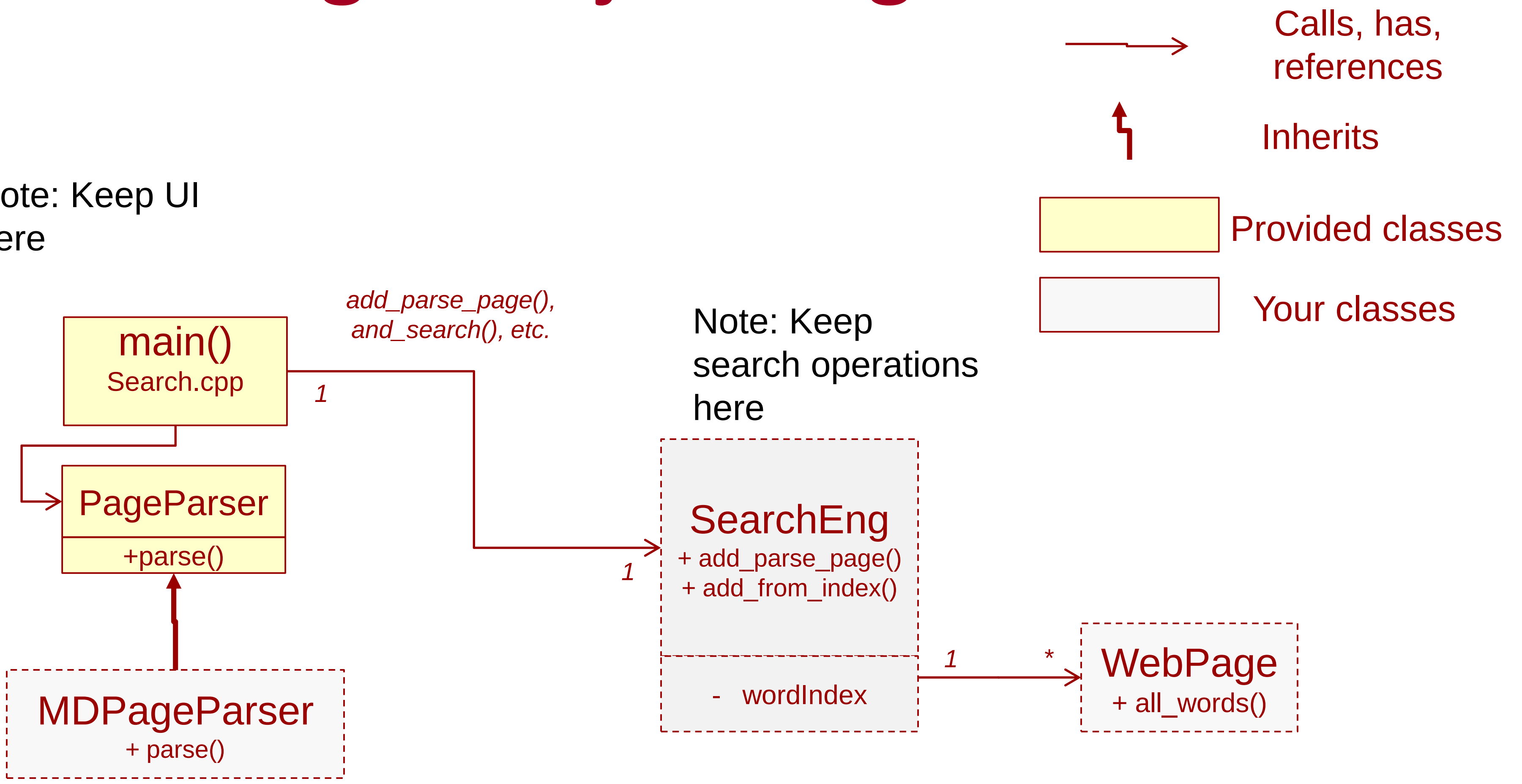
```
class Monster {
  int getValue()
  {
    // generic code
  }
};
class MonsterA : public Monster {
 public:
  bool isDead();
  int getValue()
  {
   // code for Monster A with multipliers & head shots
  }
};

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

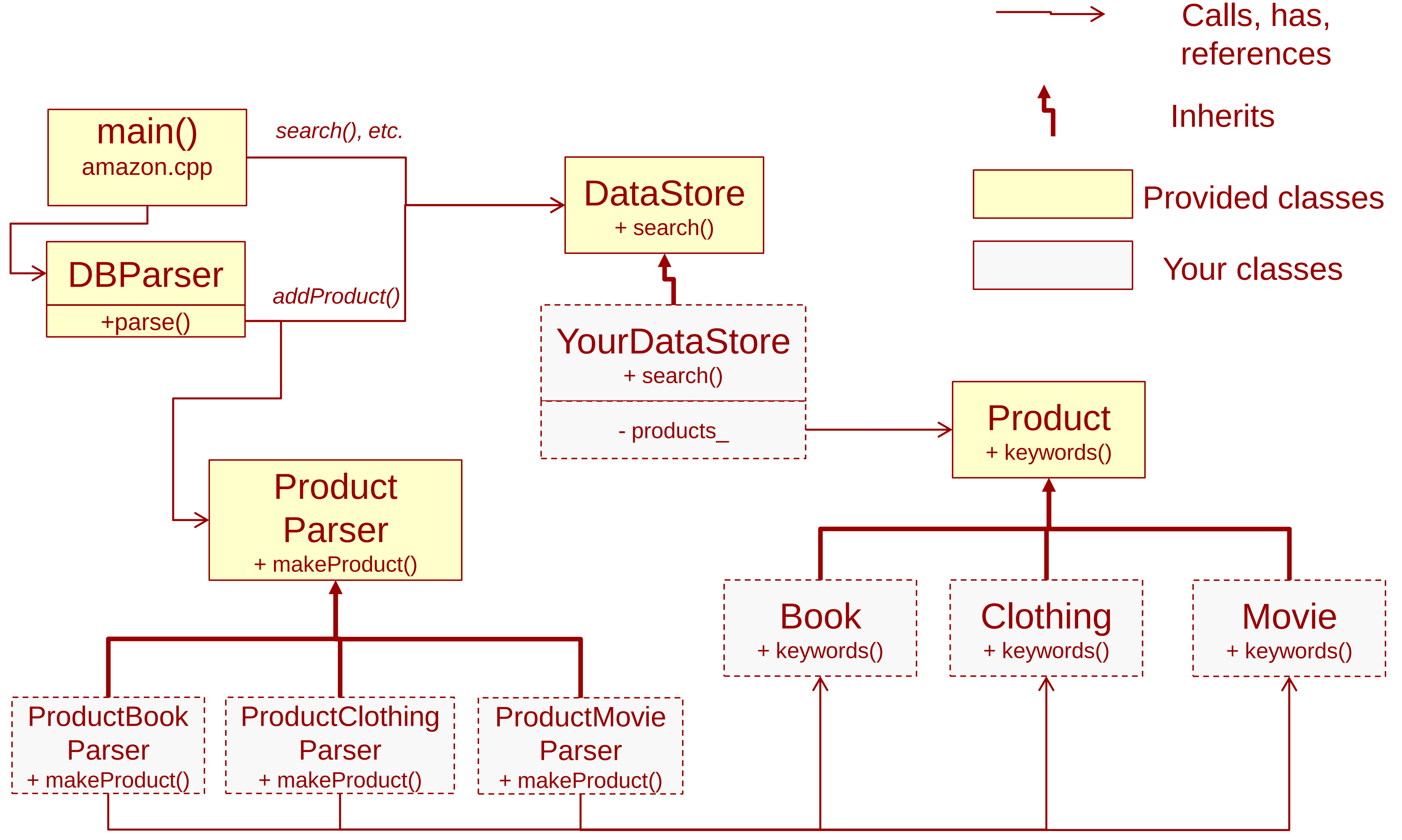
- Will this work?
- Yes!!
- Now I can add new Monster types w/o changing any Monster classes
- Only the creation code need change

```
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;
}
```

Google Project Organization



Amazon Project Organization



Calls, has, references
Inherits
Provided classes
Your classes
main()
amazon.cpp
search(), etc.
DataStore
+ search()
DBParser
+parse()
addProduct()
YourDataStore
+ search()
- products_
Product
+ keywords()
Product Parser
+ makeProduct()
Book
+ keywords()
Clothing
+ keywords()
Movie
+ keywords()
ProductBook Parser
+ makeProduct()
ProductClothing Parser
+ makeProduct()
ProductMovie Parser
+ makeProduct()