Lecture 9

Polymorphism & Virtual Functions

Topics covered in last lecture

- Composition
- Inheritance
  - Class declaration
  - Inheritance type
  - Inherited access
  - Constructor and destructor
- Type conversion
- Class hierarchies

Inheritance

An object of Player represents only one player!

```
class Player {
private:
  char playerSymbol;
public:
   Player(char s) { playerSymbol = s; }
   virtual char getMove(Board b, int& x, int& y) = 0;
   char getPlayer() {return playerSymbol;}
};
```

```
class HumanPlayer : public Player {
public:
   HumanPlayer(char s): Player(s) {}
   char getMove(Board b, int& x, int& y);
};
```

```
class RandomPlayer : public Player {
public:
   RandomPlayer(char s): Player(s) {}
   char getMove(Board b, int& x, int& y);
};
```

```
class SmartPlayer : public Player {
public:
  SmartPlayer(char s): Player(s) {}
  char getMove(Board b, int& x, int& y);
};
```

Any specific player *is a* player. All these classes are almost the same except the implementation of *getMove* function

Polymorphism

Bad solution

```
class Board {
    char grid[BOARDSIZE][BOARDSIZE];
public:
   bool addMove(int x1,int y1,int x2,int y2);
   bool checkWin();
   bool validInput(int, int);
   void printBoard();
};
```

```
class Game {
  Board *board;
  RandomPlayer *player1; //hard coded
  SmartPlayer *player2; //hard coded
public:
  Game(Board* b,Player* p1,Player* p2);
  void play();
};
```

```
class Game {
  Board *board;
  Player *player1, *player2;
public:
  Game(Board* b, Player* p1, Player* p2);
  void play();
};
```

Right solution

```
int main() {
 Board* board = new Board(10);
 Player* p1 =  new HumanPlayer('C');
 //Player* p1 =  new RandomPlayer('C');
 Player* p2 =  new SmartPlayer('B');
public:
  Game game(p1, p2);
  game.play();
};
```

How are different players' moves passed to Game class?

Topics covered in the lecture

- Function calls and binding
- Static binding
- Function overriding and dynamic binding
- Polymorphism
- Virtual functions
- Abstract classes

Difficulty factor:
*****

composition

abstraction | encapsulation | data hiding | inheritance | polymorphism

Object oriented analysis, design and programming

Function calls and binding

```
class Binding {
public:
  void print(int value) { cout << value << endl; }
};

Void print(int value) {
        cout << value << endl;
}

int main()
{
   Binding b;
   b.print(10);
   print(10);
   return 0;
}
```

binding.cpp

binding

binding

Find the code that implement the function.

Static binding

- Connecting a function call to a function body (the code) is called binding.
- Static binding: binding is performed at compiling before the program is run.

```
void testOverloading( int numerator, int denominator) {
        int fraction = numerator / denominator;
        cout << "Fraction1 = " << fraction << endl;
}
void testOverloading(double numerator, double denominator)
{
        double fraction = numerator / denominator;
        cout <<  "Fraction2 = " << fraction << endl;
}
void testOverloading( int numerator, double denominator) {
        double fraction = numerator / denominator;
        cout <<  "Fraction3 = " << fraction << endl;
}
```

```
testOverloading(3, 7.0);
```

Function overriding and binding

- Function overriding (method overriding) allows a derived class to provide a new implementation of a method that is already implemented in its base class.

staticBinding.cpp

- With static binding, a function call from an object of a class is bound to the implementation of the function in that class.
- Do not mix up function overloading and function overriding.

Function overloading vs Function overriding

Same function name but different (or different number of) parameters
f1(double) & f1(int)

Same function name and parameters but defined in different classes in a hierarchy
classOne::f1(int) & classTwo:: f1(int)

Static binding for overriding functions

- Static binding: *Call own implementation if any; otherwise call base implementation.*

```
class One {
public:
  double f1(double);
  double f2(double);
};

double One::f1(double num)
{
    return num+1;
}
double One::f2(double num)
{
  return f1(num)*f1(num);
}
```

```
class Two: public One {
public:
    double f1(double);
};
double Two::f1(double num)
{
   return num+2;
}
```

Class One:

$f1(n) = n+1;$

$f2(n) = f1(n)^2;$

Class Two:

$f1(n) = n+2;$

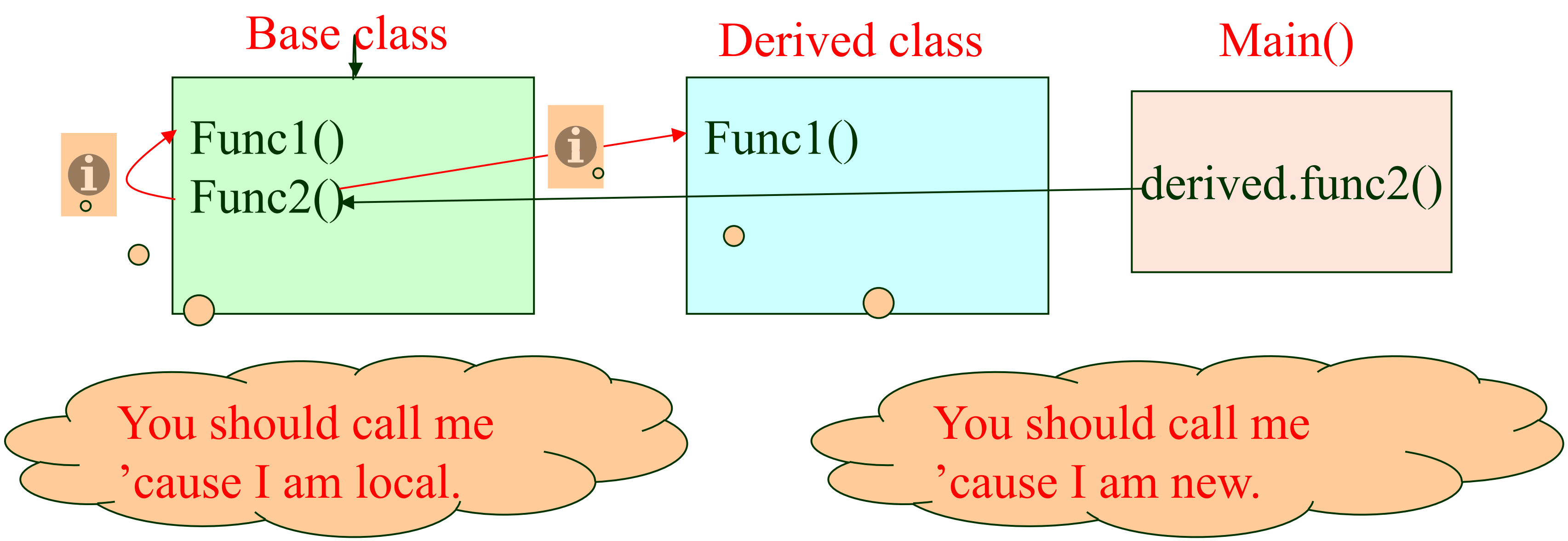
$f2(n) = f1(n)^2;$

Question: *If an object in class Two call f2, which f1 will be called?*

See example access1.cpp

Polymorphism

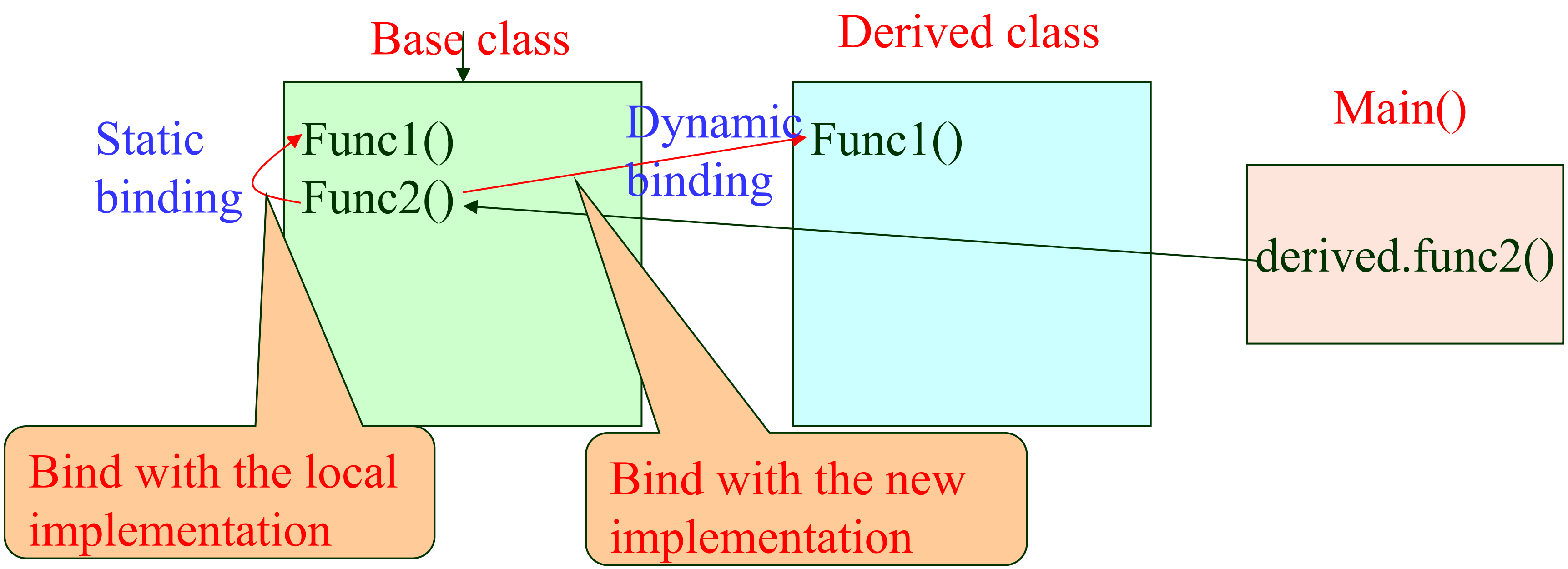
- Unlike function overloading where different parameters determine which version of the function to execute, for function overriding, the name and its signature of a function are exactly the same in both base class and derived class, which one should be executed when it is invoked?



Polymorphism

Why should we allow polymorphism?

- *Polymorphism* permits the same function name to invoke one response in objects of a base class and another response in objects of a derived class.
- The way that C++ determines which function to call is through two types of binding, *static* and *dynamic*.



Polymorphism

- If we want a binding method that is capable of determining which function should be invoked at run time. This type of binding is refereed to as **dynamic binding.** To allow this happening, we use the following keyword in the base class:

```
virtual double f1(double);
```

- A polymorphic function that is dynamically bound is called a **virtual function**.

See example access2.cpp

Polymorphism

- The use of virtual functions allows dynamic binding. This means that depending upon the object which calls an overridden function, the appropriate function will be used. A virtual function effectively creates a pointer to that function, but does not assign an actual value to the pointer until run-time.

Why can't determine it during compiling?

```
class Game {
  Board* board;
  Player* player[2];
public:
  Game(Board* b, Player* p[]);
};
```

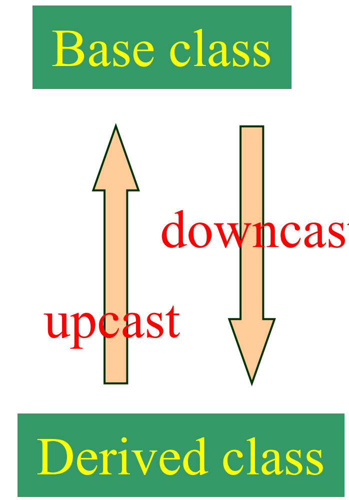
```
class Player {
protected:
  char playerSymbol;
public:
  virtual void getMove(Board, int&, int&)=0;
};
```

```
class HumanPlayer: public Player {
public:
  HumanPlayer(char ps)
         {playerSymbol = ps;}
  void getMove(Board, int&, int&);
};
```

```
class SmartPlayer: public player {
public:
  SmartPlayer(char ps)
         {playerSymbol = ps;}
 void getMove(Board, int&, int&);
};
```

Dynamic binding and up-casting

- Note that results of dynamic binding depend on the ways of casting.
  - Casting by pointer or reference: works better




```
Game(Board* b, Player* p[]) {
      board = b;
      player[0] = p[0];//deep copy
      player[1] = p[1];
}
//in main function
Board* b = new Board(10);
Player* p[2];
P[0] = new HumanPlayer(‘C’); // upcasting
P[1] = new RandomPlayer(‘B’); //upcasting
Game game(b, p);
```

```
class Player {
protected:
  char playerSymbol;
public:
  virtual void getMove(Board, int&, int&);
};
```

Dynamic binding and upcasting

- However, when casting the whole object or pass parameter by value, the object would be sliced to the object of its base class

Check the differences from the following two functions:

```
void describe(Pet p) { // Slice the object
  cout << p.description() << endl;
}

void describe(Pet *ptr) { // no object slicing
  cout << ptr->description() << endl;
}
```

See example ObjectSlicing.cpp

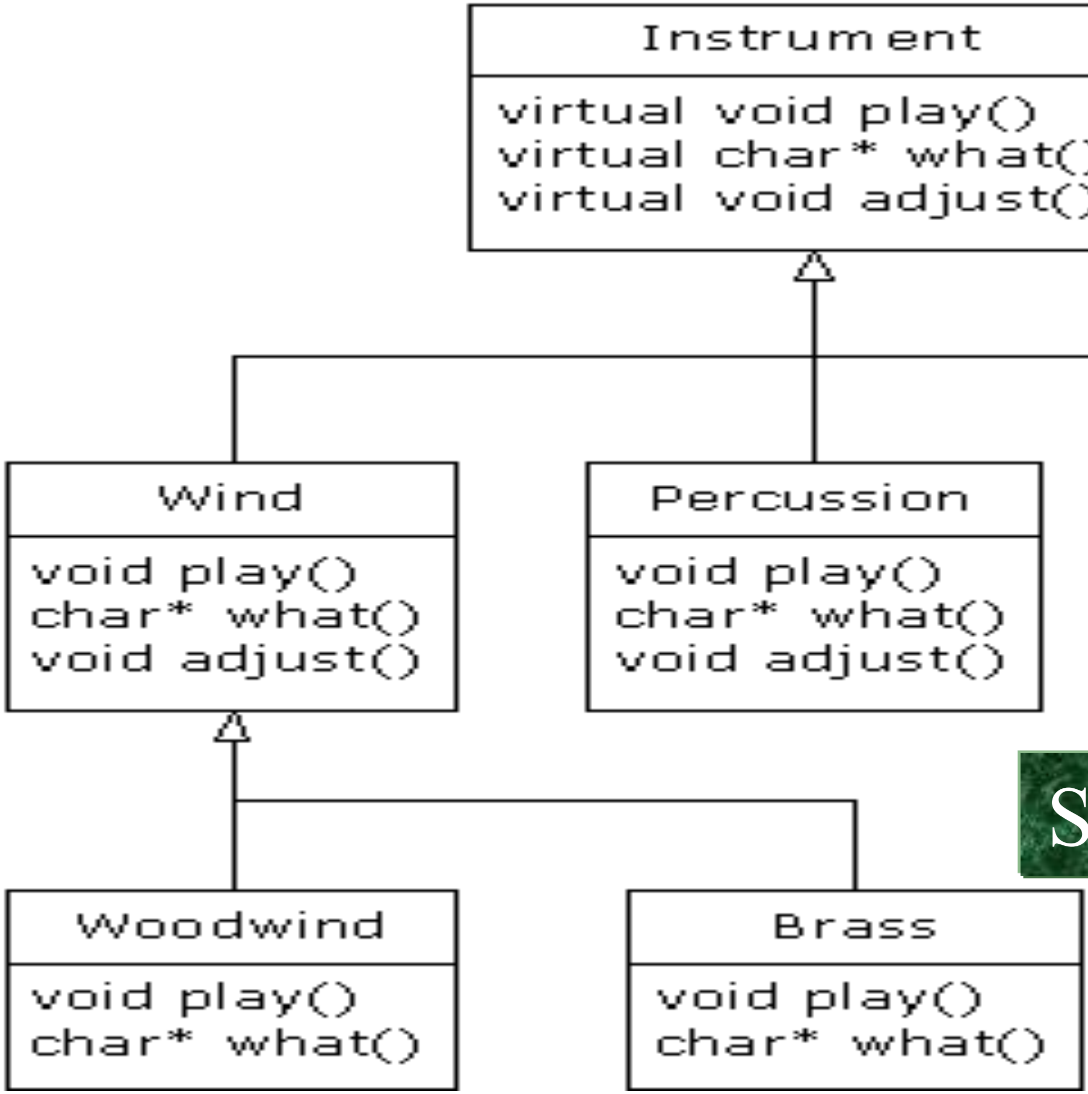
Pure Virtual Function and Abstract Classes

- Pure Virtual Functions: A virtual function is made pure by the initializer of =0.
- No definition is needed for pure virtual functions.
- Abstract Class: A class with at least one virtual function is called an abstract class. No object of an abstract class can be created but a pointer of an abstract class can be defined.

```
class Player {
protected:
  Char playerSymbol;
public:
  virtual void getMove(Board*, int&, int&) = 0;
};

Player p; //illegal
Player* p; //legal
```

Hierarchy of Instruments



See InstrumentH.cpp

Mixture of overloading & overriding

- Overriding a virtual function can't change the return type. See ChangeReturn.cpp
- If overriding one of the overloaded member functions in the base class, the other overloaded versions become hidden in the derived class.

See NameHidding.cpp

Virtual destructor

See destructors.cpp

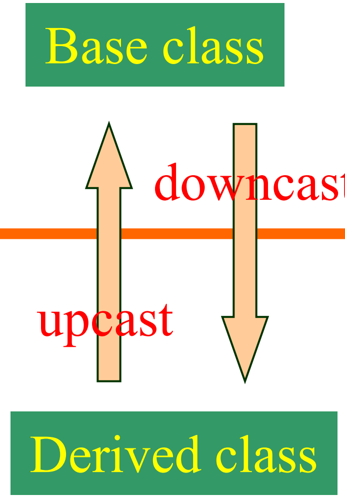
- You may have experienced warning message "no virtual destructor" even though you don't think you need a destructor.

```
class Base {
    virtual void function();
  // but no destructor here
};
class Derived : public Base {
     void function() { int* p = new int[1000]; }
     ~Derived() {
          delete [] p;
     }
};
//in another part of the program
Base *b = new Derived();
delete b; // Here's the problem!
```

Borrowed memories, which are used by the overridden methods

The base class's destructor can't do the clean-up for the derived class because b is a pointer of the base class. You might incur some memory leaking.

Downcasting



- Two ways to downcast an object:
  - static_cast: unsafe
  - dynamic_cast: safe but no guaranty of success.

When you use dynamic_cast *to try to cast down to a particular type, the return value will be a pointer to the desired type only if the cast is proper and successful; otherwise, it will return zero to indicate that this was not the correct type.*

See DynamicCast.cpp

```
Pet* d = new Dog;
Dog* d1 = dynamic_cast<Dog*>(d);
Cat* c = dynamic_cast<Cat*>(d);
```

Homework

- Read Textbook chapter 15.
- Complete online tutorial 2.
- Attend this week's PASS session, which provides necessary training for assignment 2.