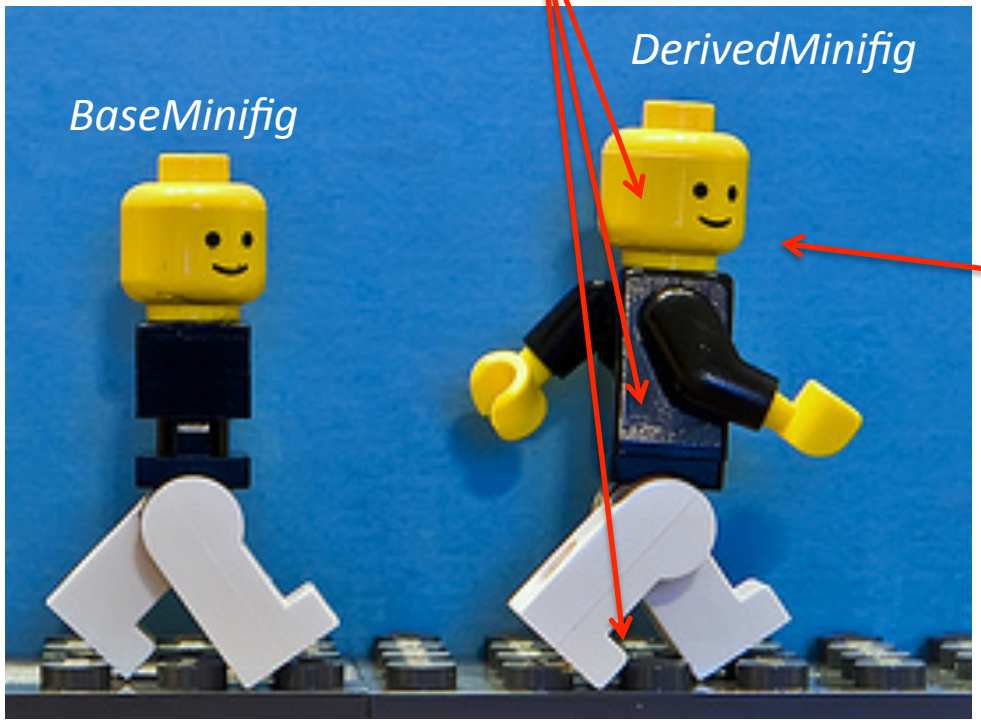
Chapter 11 & 12 – Inheritance & Polymorphism

FALL 2015

Inheritance

You can create a class that **inherits** attributes of an existing class.



A "derived class" contains behaviors inherited from its base class and can contain additional behaviors.

"Base Class" **"Derived Class"**

"Inheritance is a form of software reuse in which you create a class that absorbs an existing class's data and behaviors and enhances them with new capabilities."

Time and MilliTime Objects with NO INHERITENCE

```
class Time
{
public:
Time(int = 0, int = 0, int = 0);

  //set functions
  void setTime(int,int,int);
  void setHour(int);
  void setMinute(int);
  void setSecond(int);

  //get functions
  int getHour();
  int getMinute();
  int getSecond();

  //general member functions
  void printTime();

private:
  int hour; //0-23 hours
  int minute; //0 to 59
  int second; //0 to 59

}; //end of time class
```

```
class MilliTime
{
public:
MilliTime(int = 0, int = 0, int = 0, int=0);
  //set functions
  void setTime(int,int,int);
  void setHour(int);
  void setMinute(int);
  void setSecond(int);
  void setMilliSec(int);

  //get functions
  int getHour();
  int getMinute();
  int getSecond();
  int getMilliSec();
  //general member functions
  void printTime();

private:
  int hour; //0-23 hours
  int minute; //0 to 59
  int second; //0 to 59
  int milliSec; //0 to 999
}; //end of time class
```

Time and MilliTime Objects with "INHERITENCE"

BASE CLASS

```
class Time
{
public:
Time(int = 0, int = 0, int = 0);

  //set functions
  void setTime(int,int,int);
  void setHour(int);
  void setMinute(int);
  void setSecond(int);

  //get functions
  int getHour();
  int getMinute();
  int getSecond();

  //general member functions
  void printTime();

private:
  int hour; //0-23 hours
  int minute; //0 to 59
  int second; //0 to 59

}; //end of time class
```

DERIVED CLASS

```
class MilliTime : public Time
{
public:
MilliTime(int = 0, int = 0, int = 0, int=0);

//set functions
void setMilliSec(int);

 //get functions
 int getMilliSec();

 //general member functions
 void printTime();

private:
   int milliSec; //0 to 999
}; //end of time class
```

Notice new printTime function!

MilliTime "is-a" Time Object

Software Engineering Observation

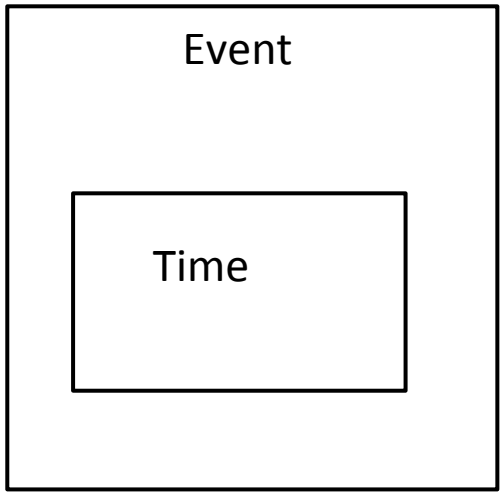
Software Engineering Observation 12.1

Copying and pasting code from one class to another can spread many physical copies of the same code and can spread errors throughout a system, creating a code-maintenance nightmare. To avoid duplicating code (and possibly errors), use inheritance, rather than the "copy-and-paste" approach, in situations where you want one class to "absorb" the data members and member functions of another class.

Inheritance

composition

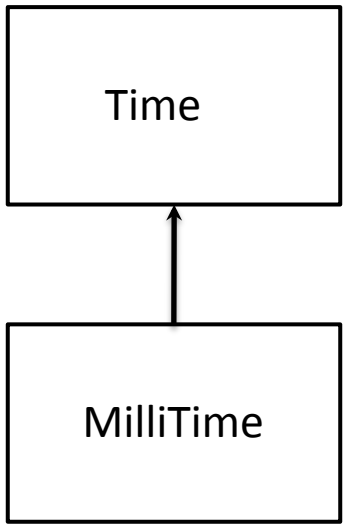
"has-a" relationship



Event "has-a" Time Object

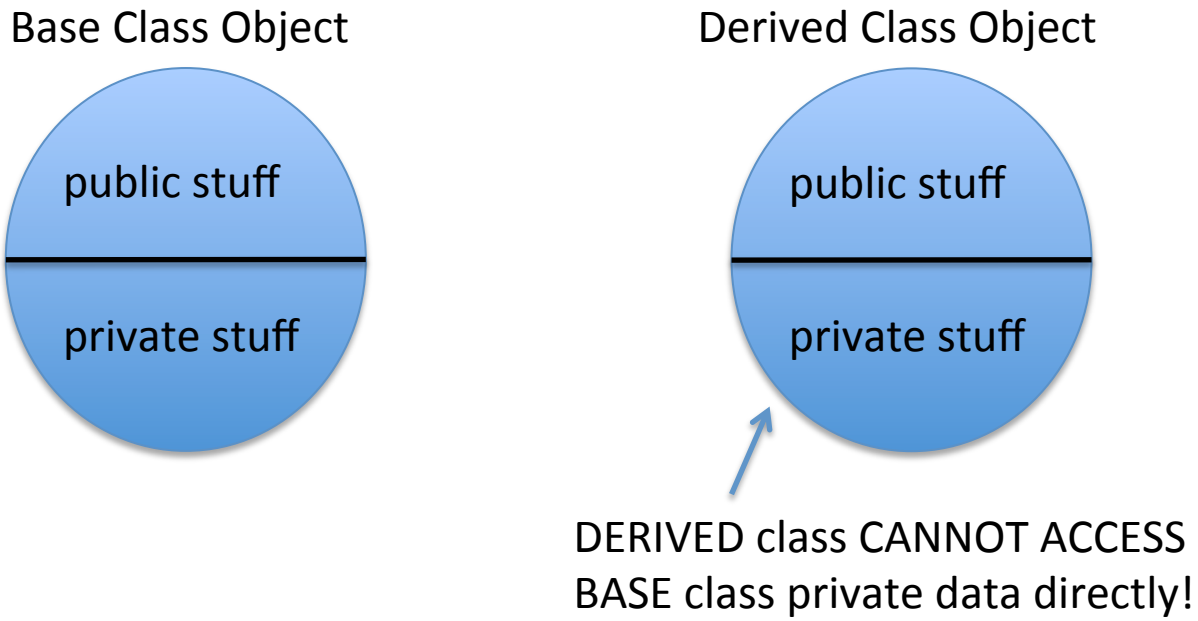
inheritance

"is-a" relationship



MilliTime "is-a" Time Object

Encapsulation Ideas



Time and MilliTime Objects with “INHERITENCE”

Not Valid

```
void MilliTime::printTime()
{

     cout << setfill('0') << setw(2)
      << hour << ":"
      <<setw(2) << minute << ":"
      <<setw(2) << second << “:”
       <<setw(3) << milliSec;

}//end printMilliTime
```

Valid

```
void MilliTime::printTime()
{

     cout << setfill('0') << setw(2)
      << getHour() << ":"
      <<setw(2) << getMinute() << ":"
      <<setw(2) << getSecond() << “:”
       <<setw(3) << milliSec;//no endl

}//end printMilliTime
```

Member Functions for the BASE CLASS is still public to the DERIVE CLASS

Data for the BASE CLASS is still private to the DERIVE CLASS

"Protected" Access Specifier and Public Inheritance

```
class Time
{
public:
Time(int = 0, int = 0, int = 0);

  //set functions
  void setTime(int,int,int);
  void setHour(int);
  void setMinute(int);
  void setSecond(int);

  //get functions
  int getHour();
  int getMinute();
  int getSecond();

  //general member functions
  void printTime();

protected:
  int hour; //0-23 hours
  int minute; //0 to 59
  int second; //0 to 59

}; //end of time class
```

Calling of protected data from BASE class is now valid!

```
void MilliTime::printTime()
{

 cout << setfill('0') << setw(2)
     << hour << ":"
     <<setw(2) << minute << ":"
     <<setw(2) << second  << ":"
      <<setw(3) << milliSec;

}//end printTime
```

changing access specifier to protected!

Software Engineering Observation

Data members private…

Software Engineering Observation 12.5

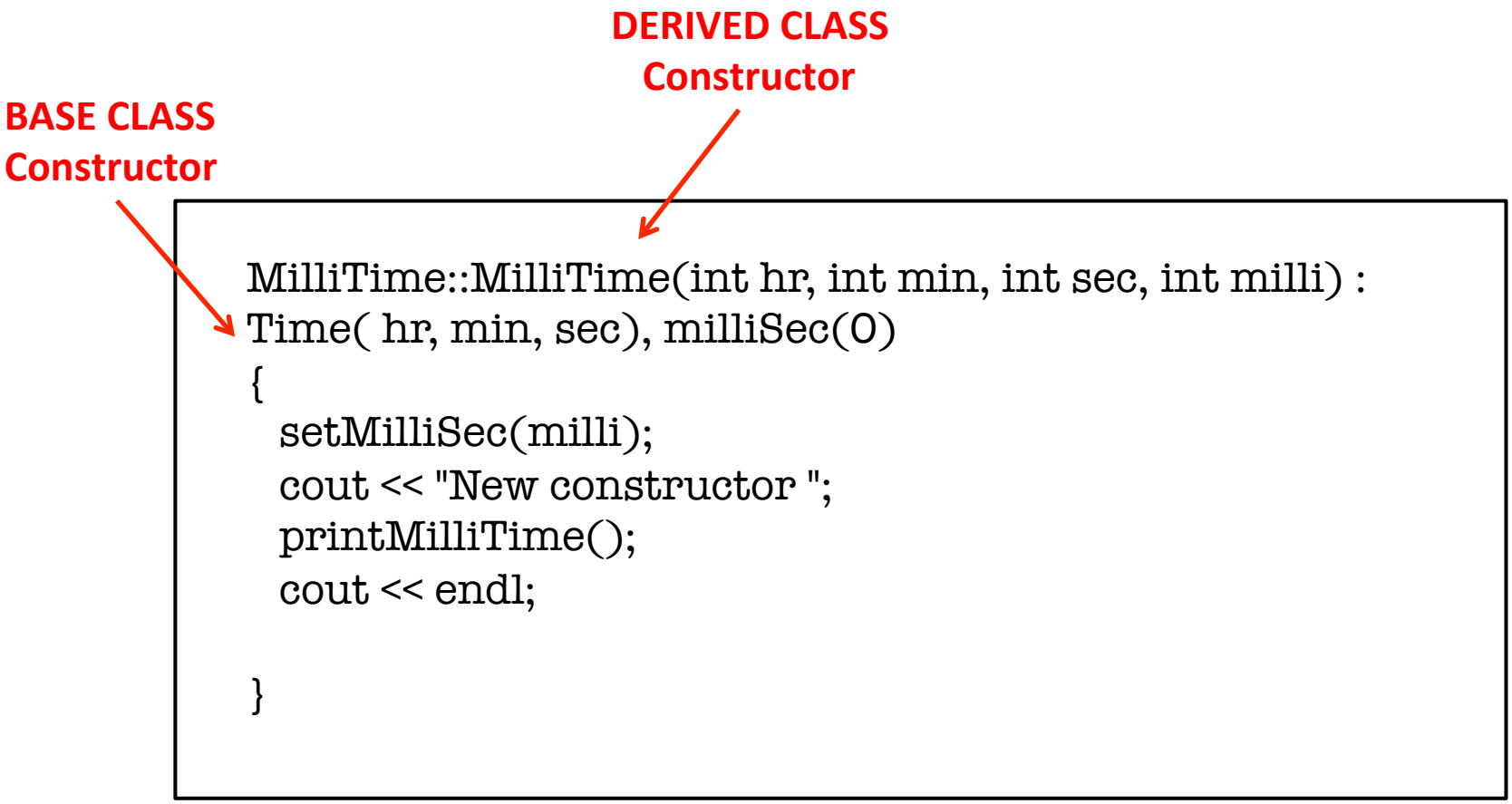
Declaring base-class data members `private` (as opposed to declaring them `protected`) enables you to change the base-class implementation without having to change derived-class implementations.

Specialize Member Functions Protected…

Software Engineering Observation 12.4

It's appropriate to use the `protected` access specifier when a base class should provide a service (i.e., a member function) only to its derived classes and `friend`s.

Constructor Calls from Derived Class

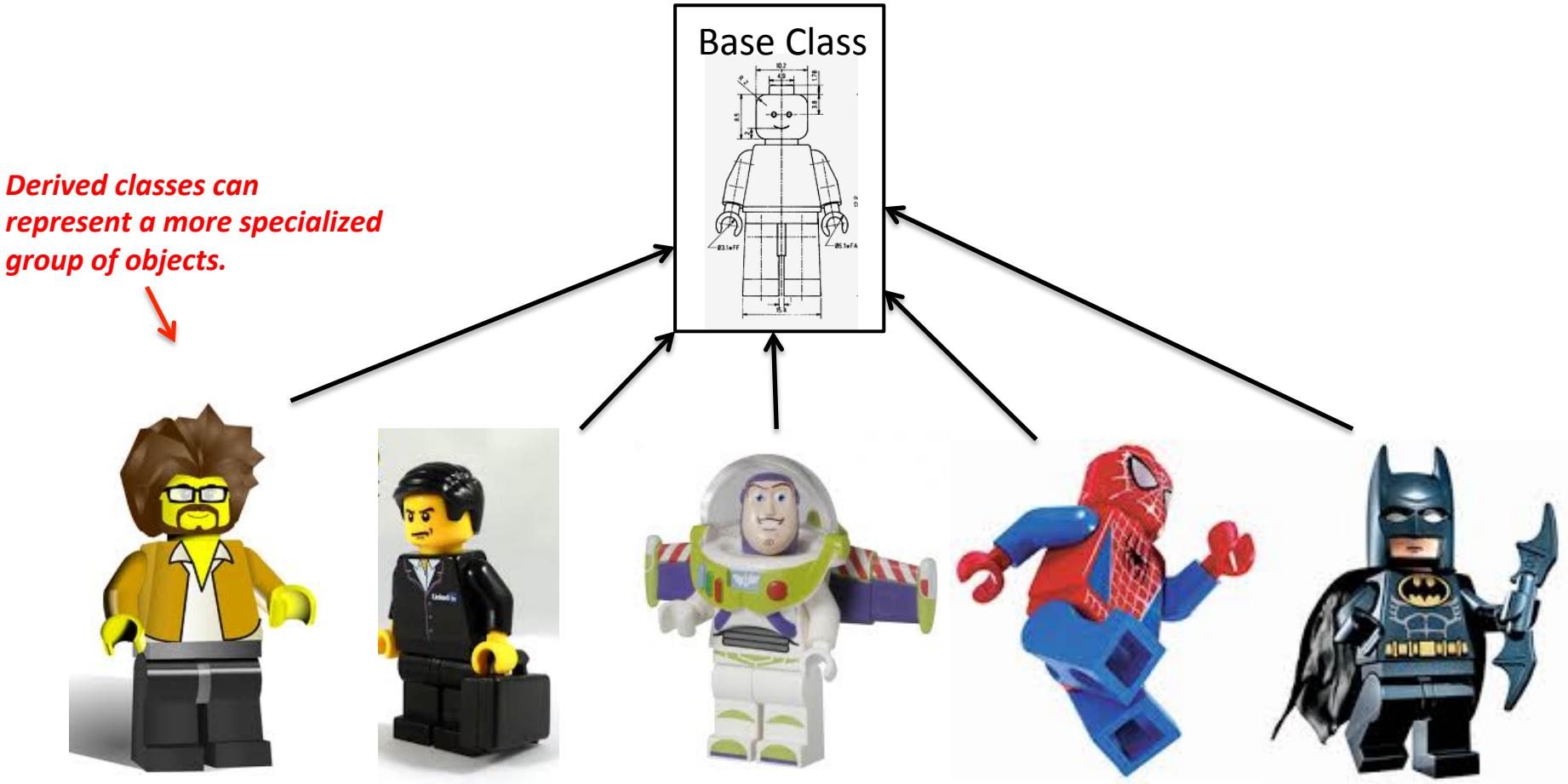


Output

The time object 03:30:30 has been created
New constructor 03:30:30:300

Inheritance Ideas

Factor out common attributes and put them in the base class!



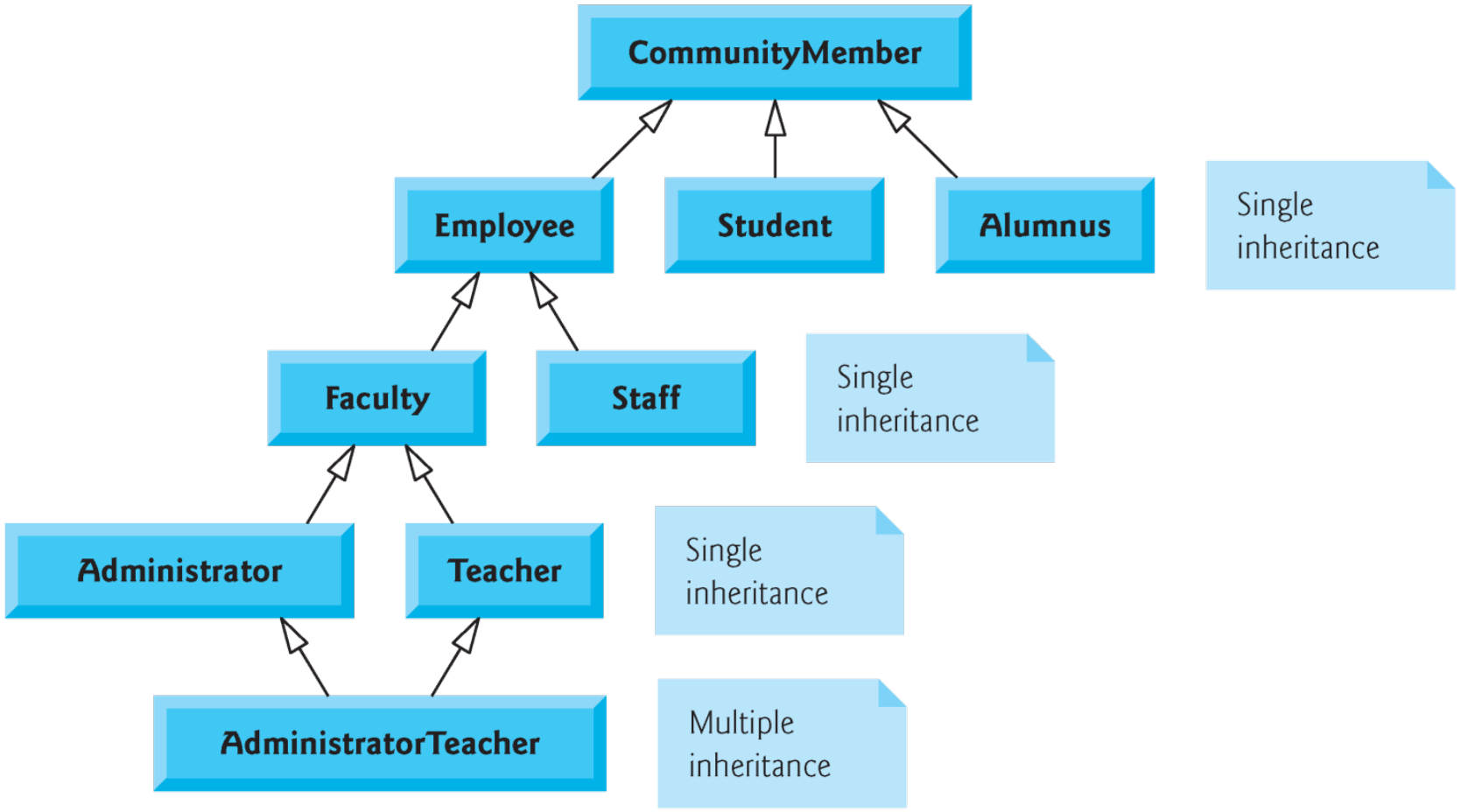
Use inheritance to form the DERIVED CLASSES.

Software Engineering Observation

Software Engineering Observation 12.2

With inheritance, the common data members and member functions of all the classes in the hierarchy are declared in a base class. When changes are required for these common features, you need to make the changes only in the base class—derived classes then inherit the changes. Without inheritance, changes would need to be made to all the source code files that contain a copy of the code in question.

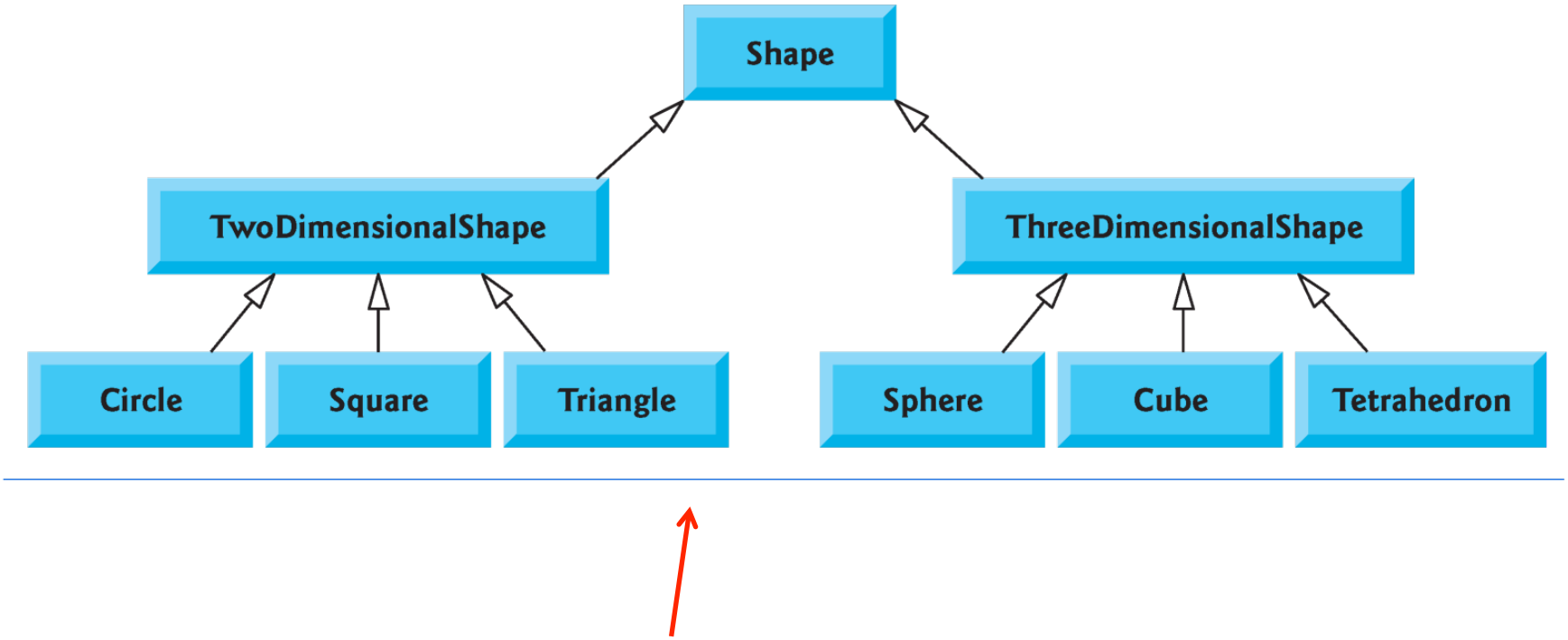
Class Hierarchy Examples! (UML Diagrams)



Example: CommunityMember is a "DIRECT BASE CLASS" of Employee.

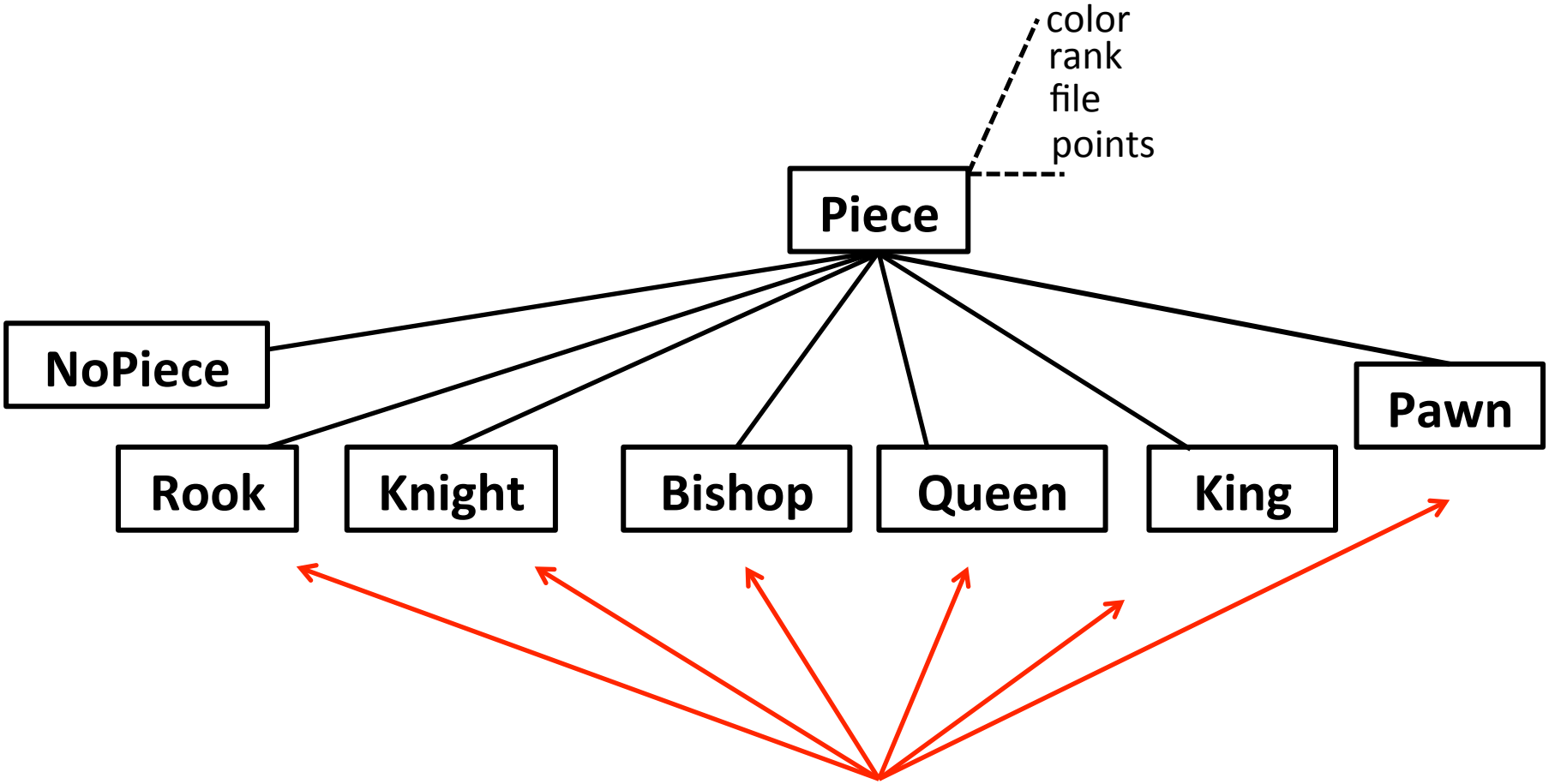
Example: CommunityMember is an "INDIRECT BASE CLASS" of Administrator.

Class Hierarchy Examples! (UML Diagrams)



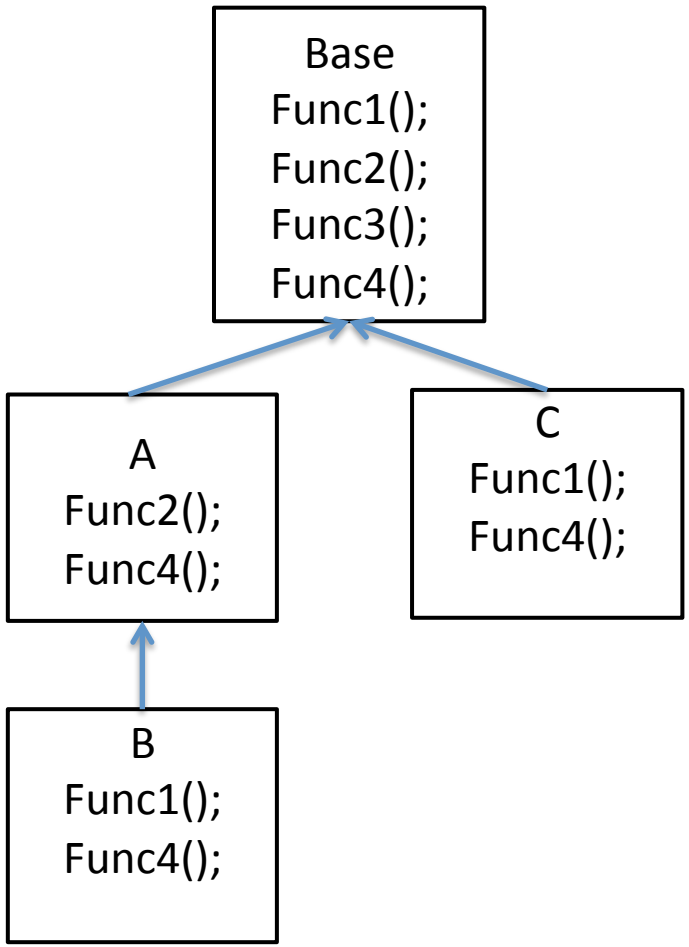
This is a 3 level hierarchy!!!

Example: Alternative Chess Class



Each of these derived classes could have a validMoves member function!

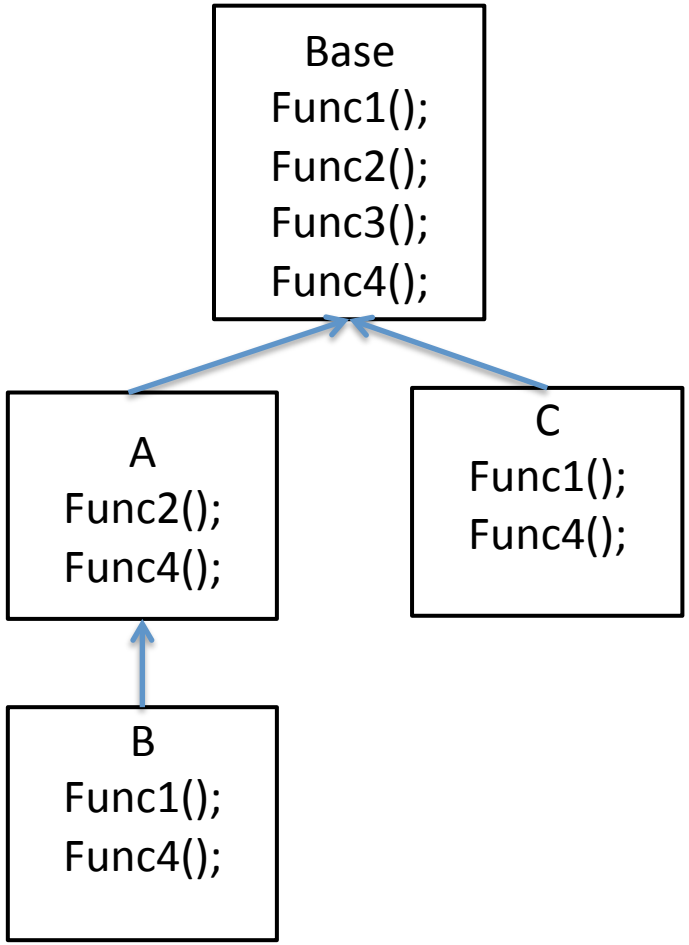
Example: Inheritance



InheritanceExample1.cc

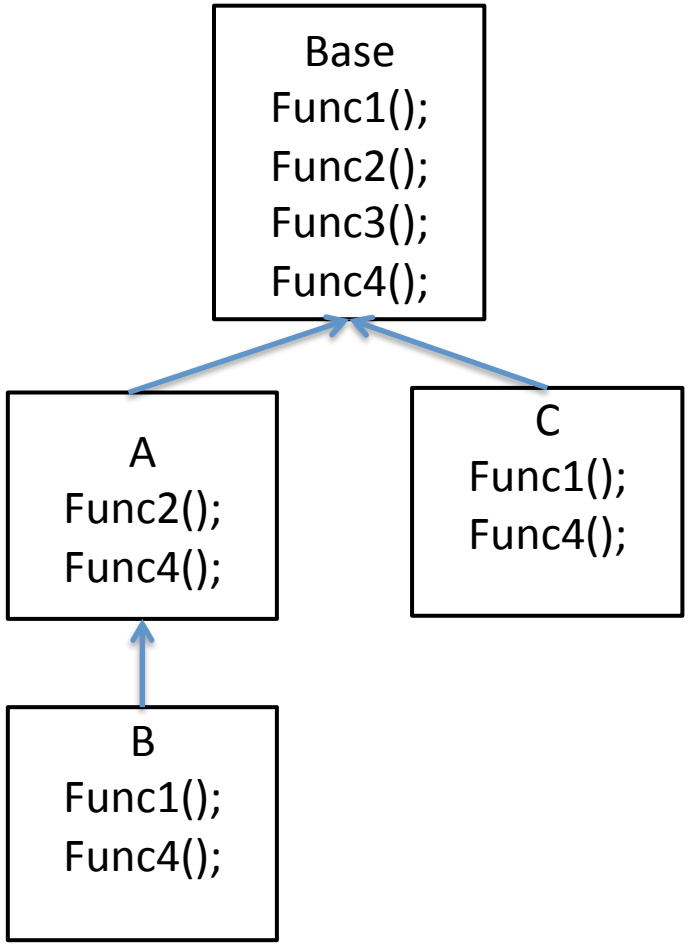
```
class Base
{ // Define a base class
 public:
  void Func1();
  void Func2();
  void Func3();
  void Func4();
};
//------------------------
class A : public Base
{ // Class A derives from Base
 public:
  void Func2();
  void Func4();
};
//------------------------
class B : public A
{// Class B derives from A
 public:
  void Func1();
  void Func4();
 };
//------------------------
class C : public Base
 { // Class C derives from Base
 public:
  void Func1();
   void Func4();
 };
//------------------------
```

Example: Inheritance



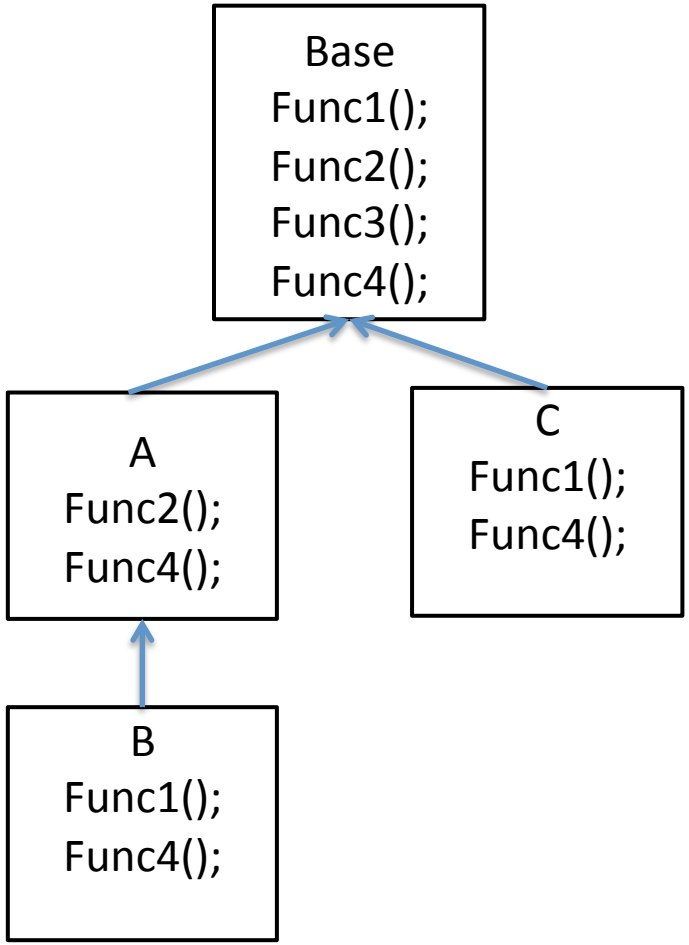

```
// Base Class Methods
void Base::Func1() { cout << "Hello from Base::Func1()" << endl; }
void Base::Func2() { cout << "Hello from Base::Func2()" << endl; }
void Base::Func3() { cout << "Hello from Base::Func3()" << endl; }
void Base::Func4() { cout << "Hello from Base::Func4()" << endl; }

// Class A Methods
void A::Func2() { cout << "Hello from A:Func2()" << endl; }
void A::Func4() { cout << "Hello from A:Func4()" << endl; }

// Class B Methods
void B::Func1() { cout << "Hello from B:Func1()" << endl; }
void B::Func4() { cout << "Hello from B:Func4()" << endl; }

// Class C Methods
void C::Func1() { cout << "Hello from C:Func1()" << endl; }
void C::Func4() { cout << "Hello from C:Func4()" << endl; }
```

Example: Inheritance




```
int main()
{ Base BaseObject;
  A  AObject;
  B  BObject;
  C  CObject;

  //Let's call Func1() from every object
  BaseObject.Func1();

  AObject.Func1();

  BObject.Func1();

  CObject.Func1();


…
}
```

Example: Inheritance



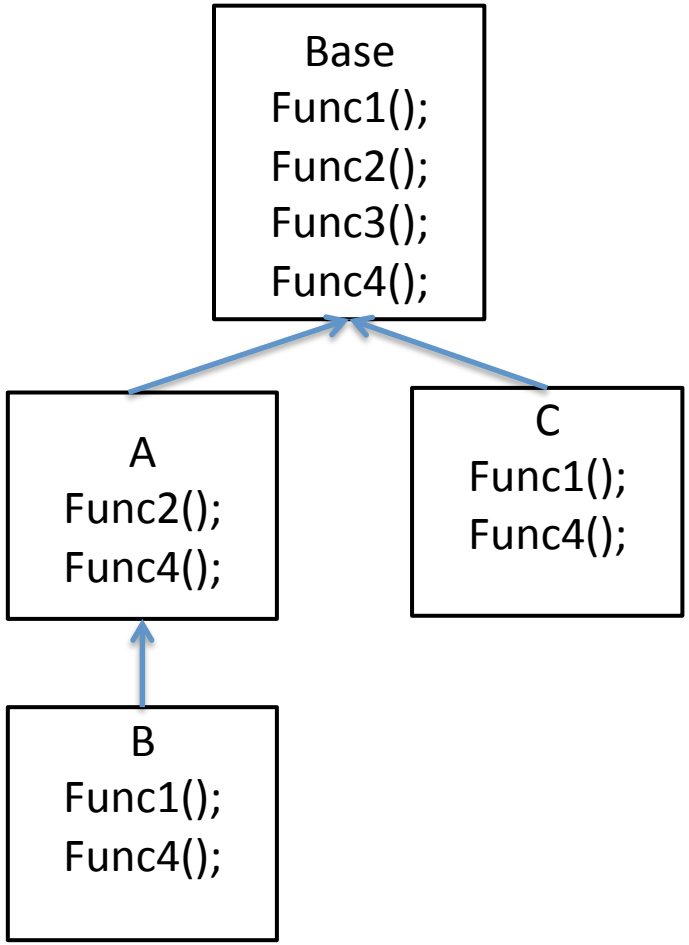
Hello from Base::Func1()

Hello from Base::Func1()

Hello from B:Func1()

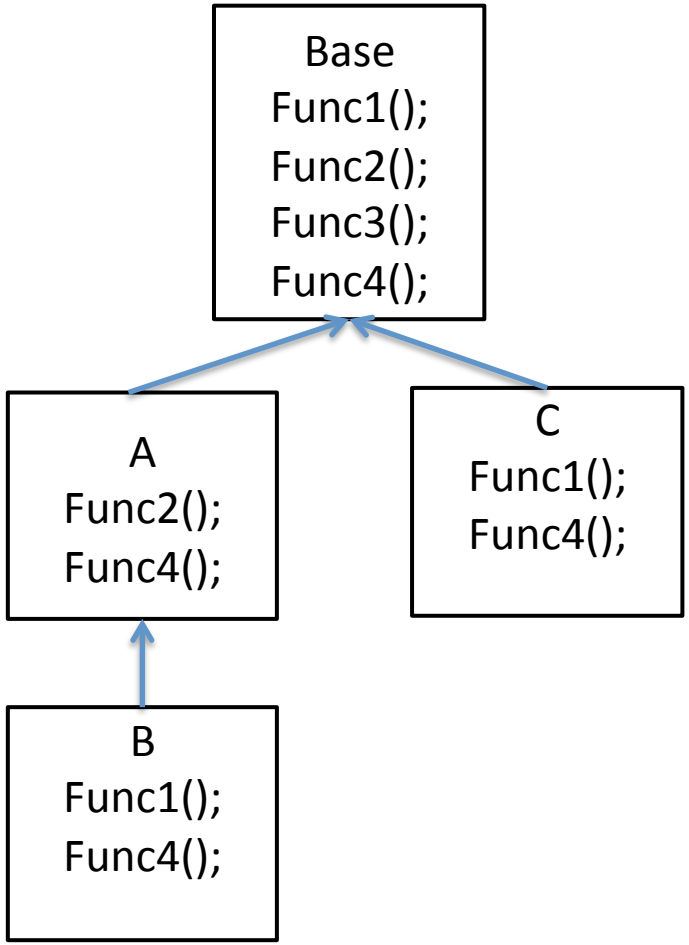
Hello from C:Func1()

Example: Inheritance




```
//Let's call Func2() from every object

  BaseObject.Func2();

  AObject.Func2();

  BObject.Func2();

  CObject.Func2();
```

Example: Inheritance



//Let's call Func2() from every object

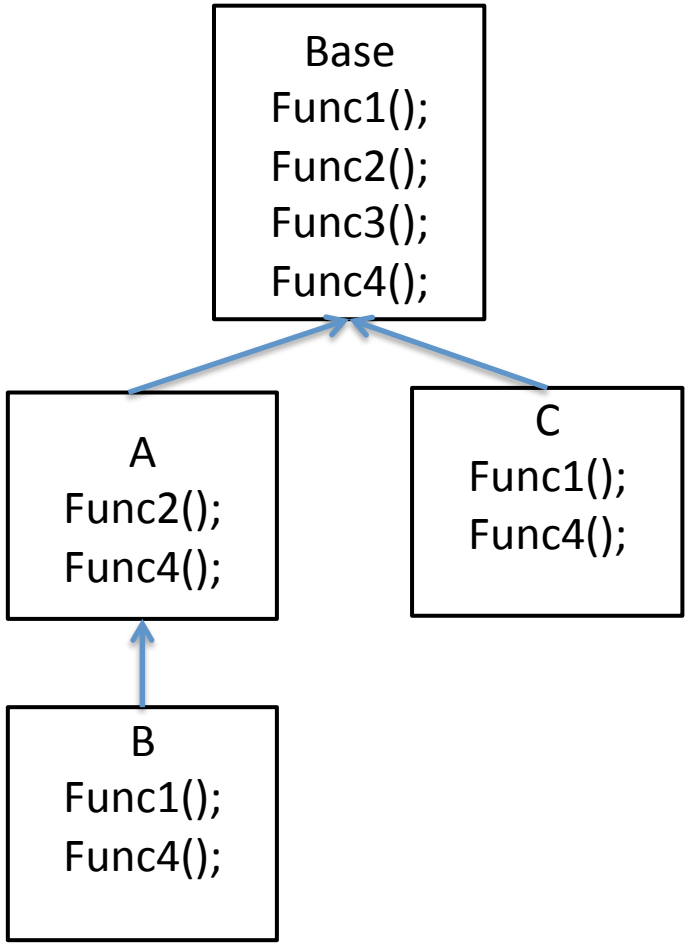
Hello from Base::Func2()

Hello from A:Func2()

Hello from A:Func2()

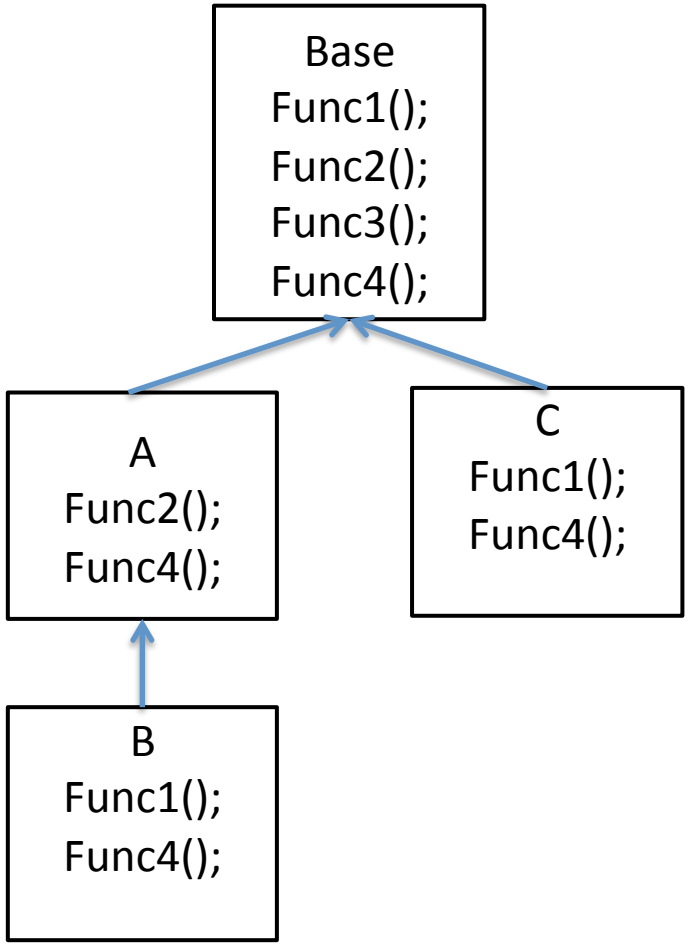
Hello from Base::Func2()

Example: Inheritance



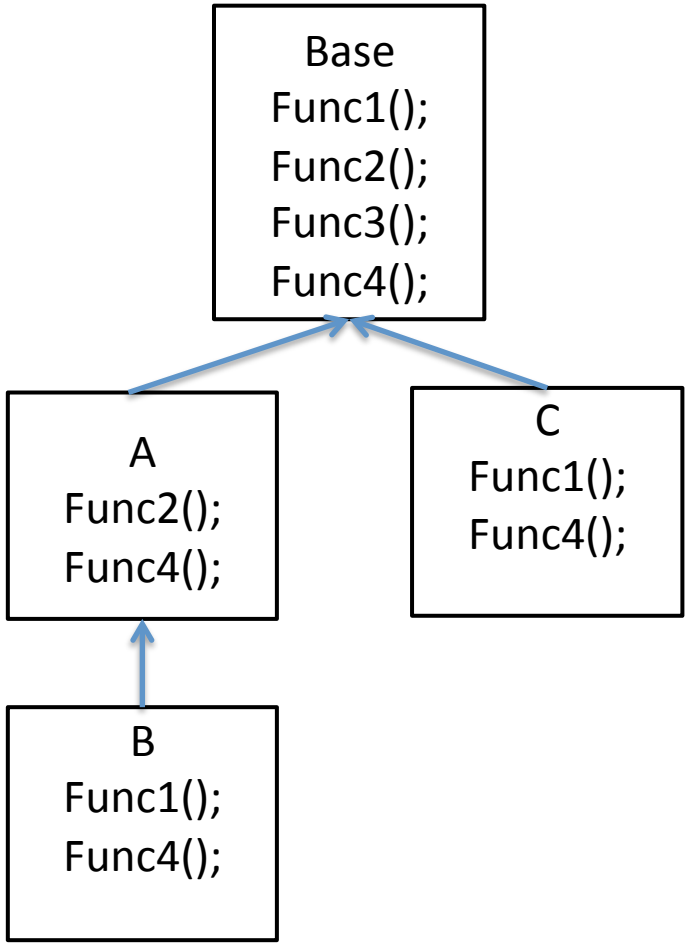

```
//Let's call Func3() from every object

BaseObject.Func3();

AObject.Func3();

BObject.Func3();

CObject.Func3();
```

Example: Inheritance



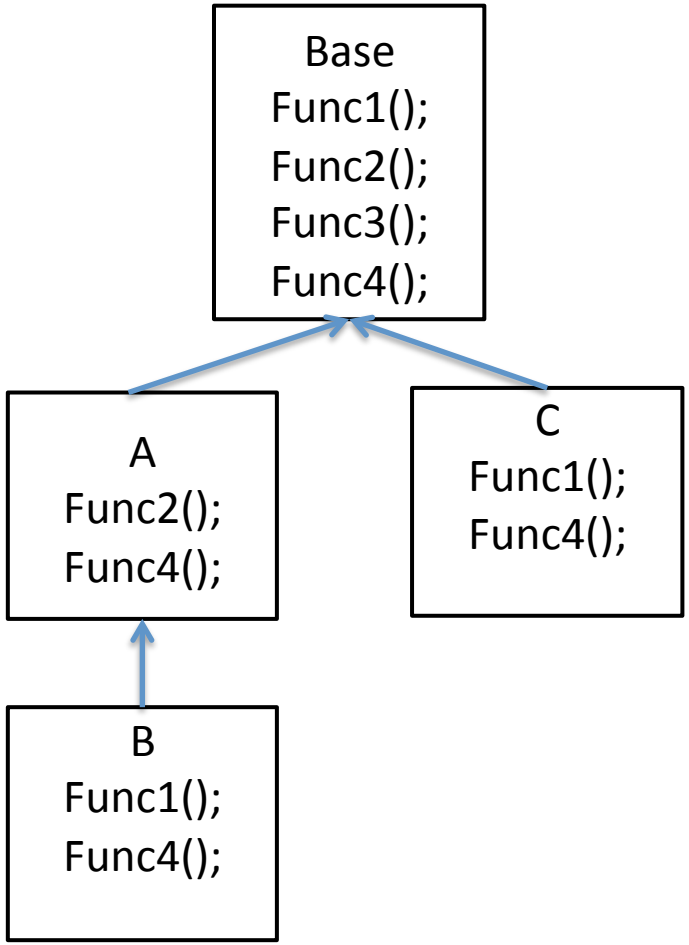

```
//Let's call Func3() from every object

Hello from Base::Func3()

Hello from Base::Func3()

Hello from Base::Func3()

Hello from Base::Func3()
```

Example: Inheritance




```
//Let's call Func4() from every object

  BaseObject.Func4();

  AObject.Func4();

  BObject.Func4();

  CObject.Func4();
```

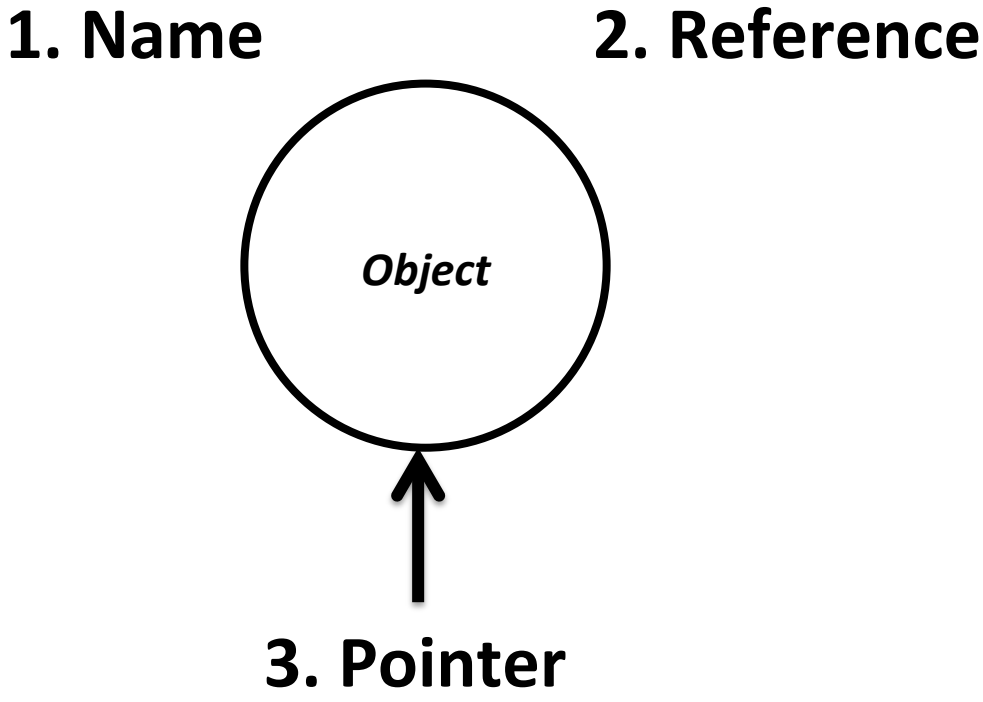
Example: Inheritance




```
//Let's call Func4() from every object

  Hello from Base::Func4()

  Hello from A:Func4()

  Hello from B:Func4()

  Hello from C:Func4()
```

Hierarchy & Handles

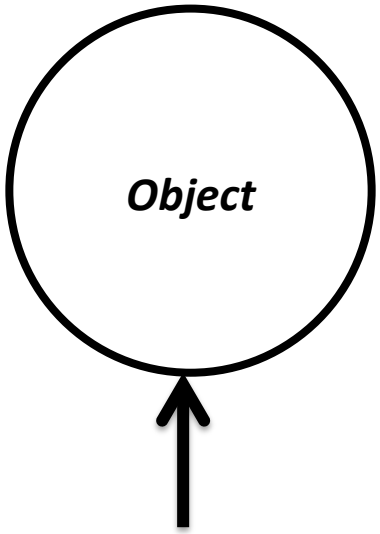
Let's look at using different "HANDLES" to access an objects in a hierarchy



Example

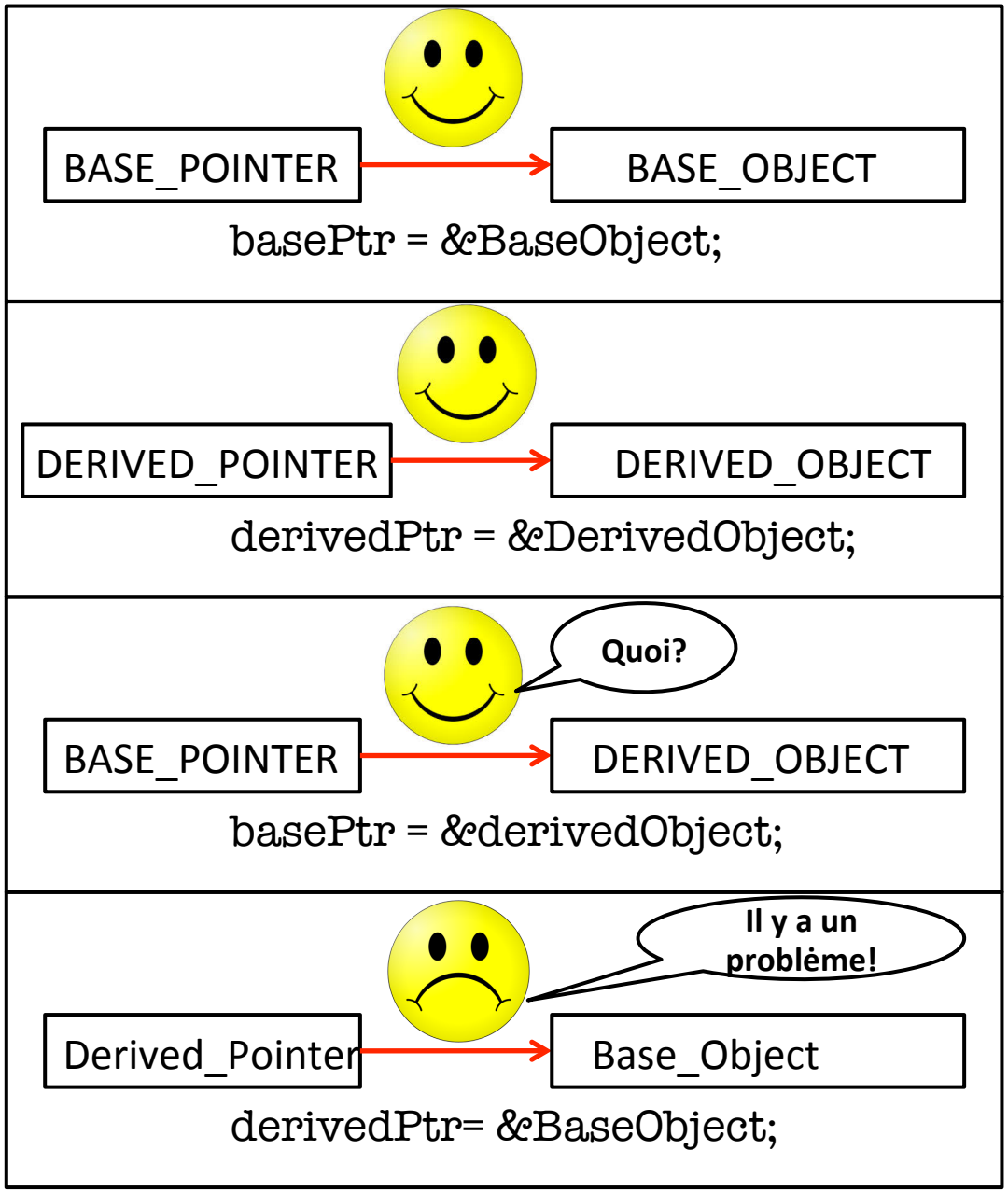
```
A  AObject;
A & RefToA = AObject;
A * PtrToA = &AObject;
```

1. AObject **2. RefToA**



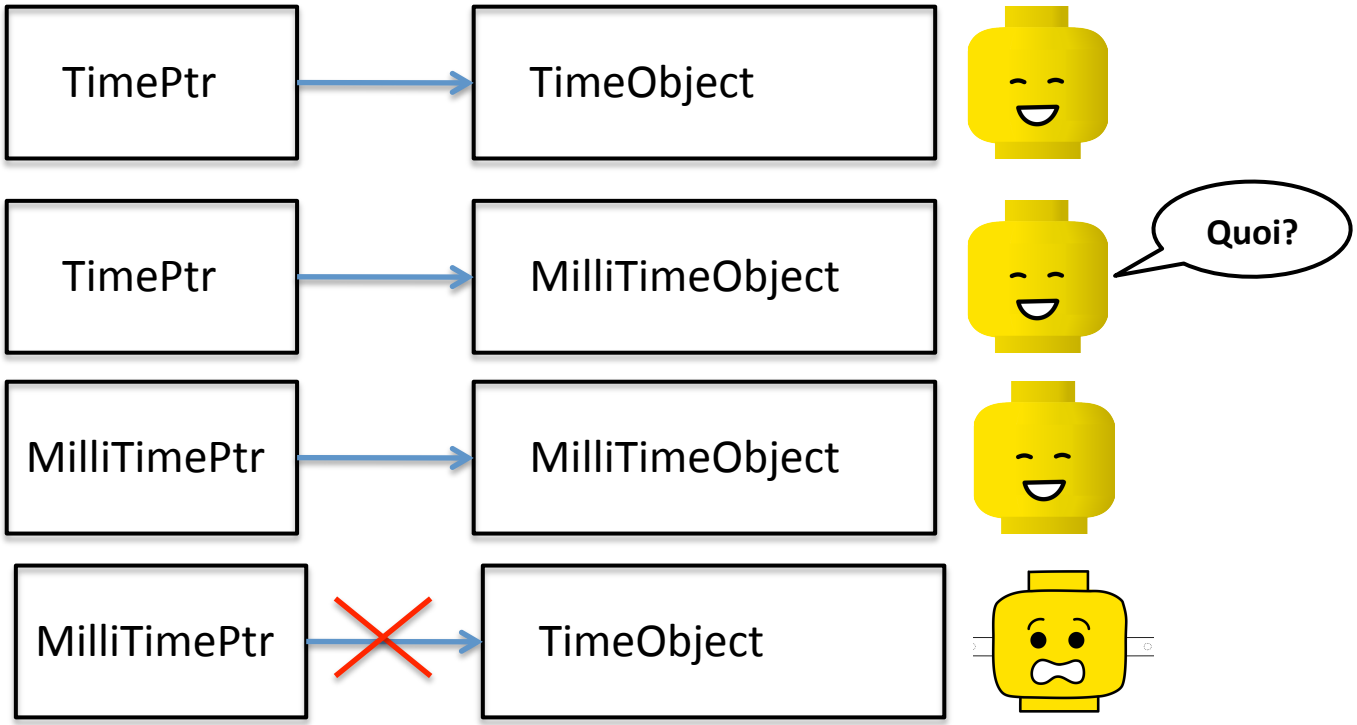
3. PtrToA

Base-Class and Derived-Class **Pointers**



Time Example

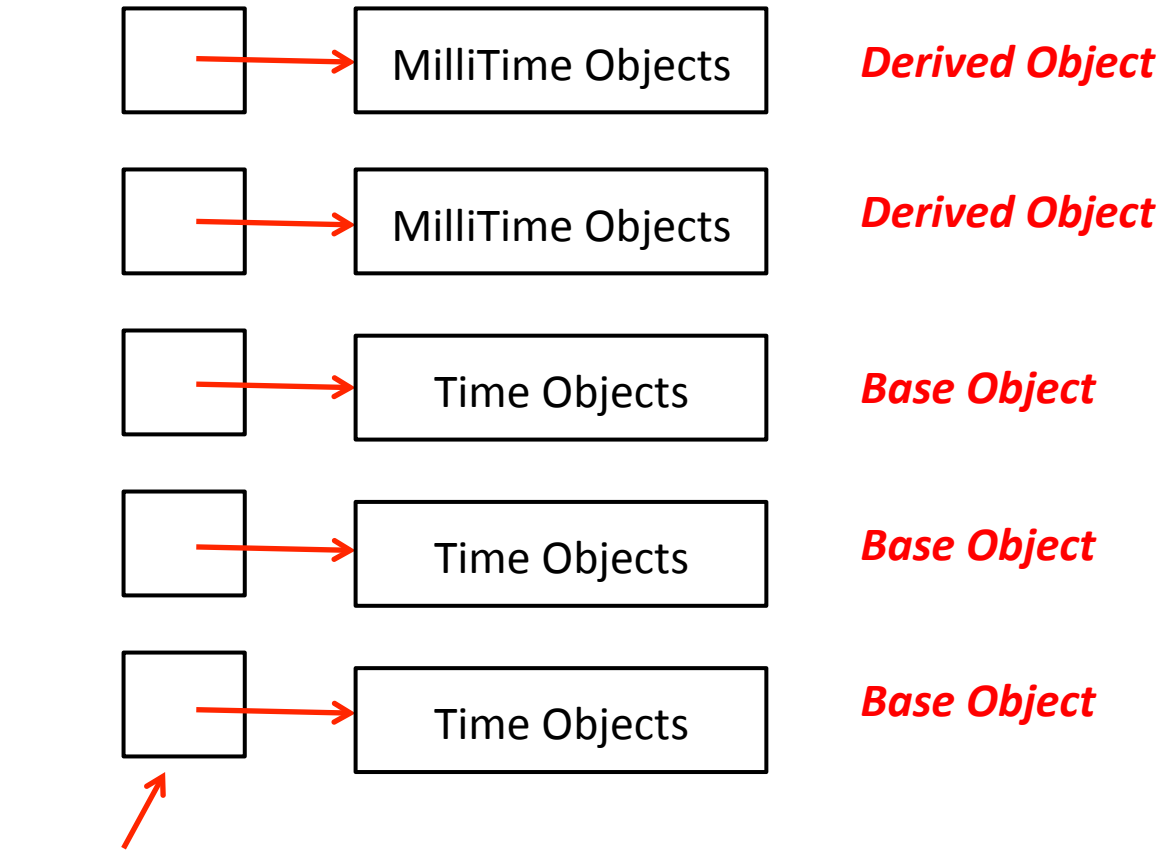
This is the new thing – We can aim a base-class pointer at a derived-class object!!



Show this with the Time class and MilliTime class example!!

Example: Array of Base Pointers

Array of Time Pointers!

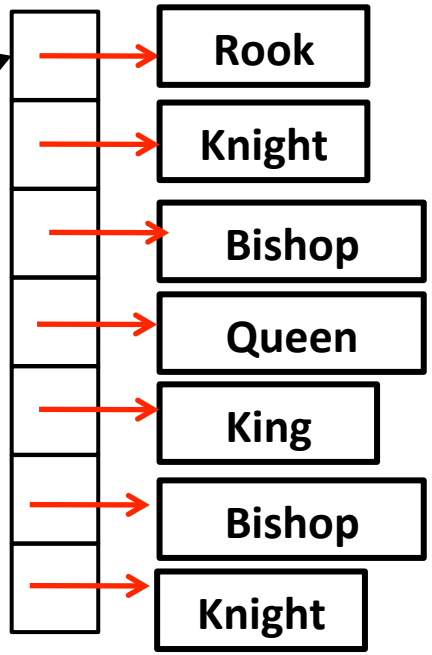


Chess Example

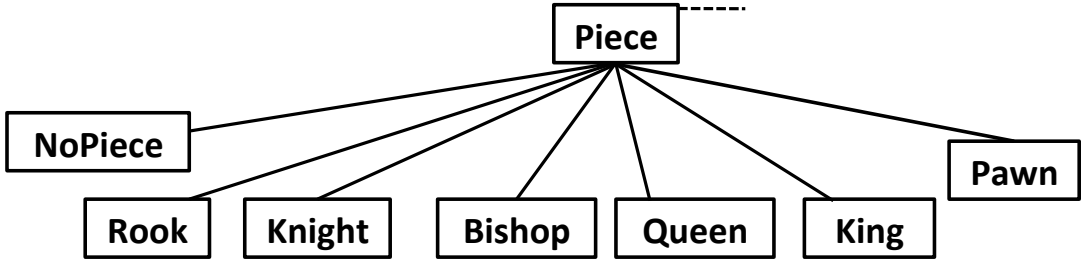
1-D array of all Pieces left in the game

These are Base Class Pointers!

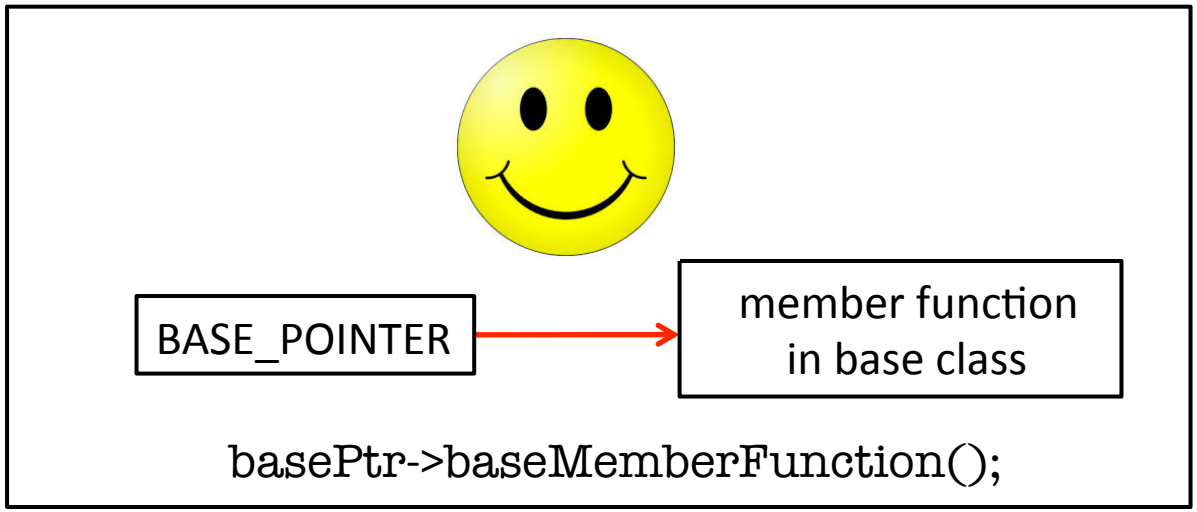
(i.e. pointers to Piece Class)

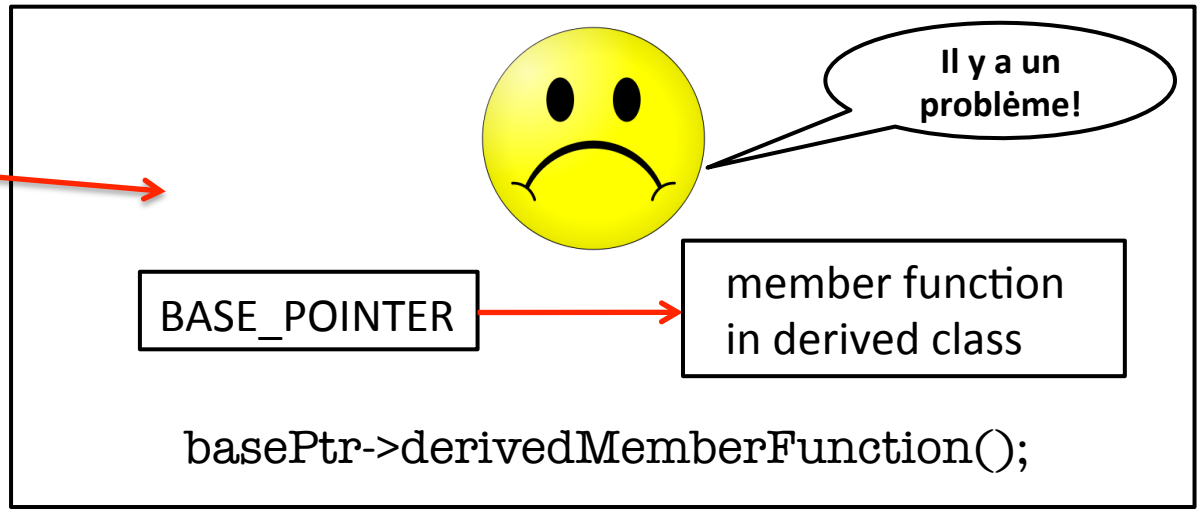


We can aim a base-class pointer at a derived-class object!!



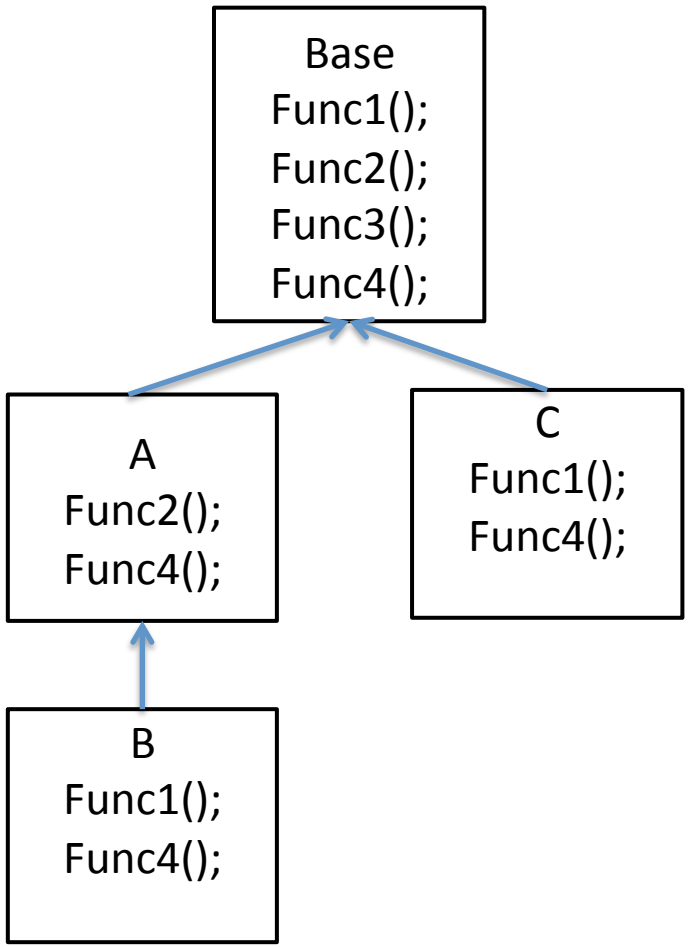
Accessing Member Functions Through Pointer (or Reference) Handle





This seems kind of awkward – but you have to be aware of this!!!!!

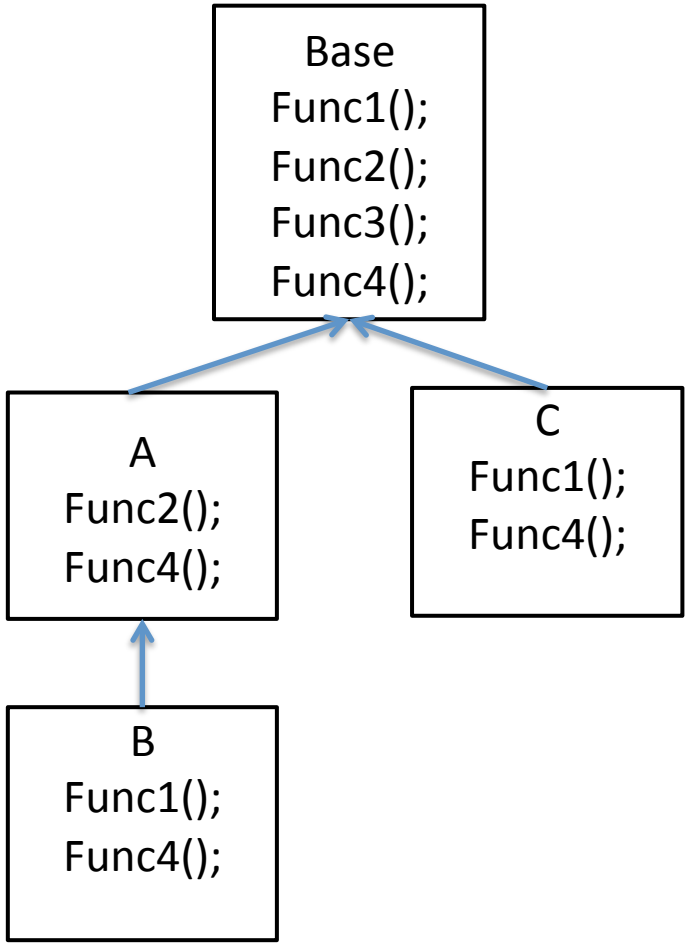
Example: Access Through Pointers



InheritanceExample2.cc

```
class Base
{ // Define a base class
 public:
  void Func1();
  void Func2();
  void Func3();
  void Func4();
};
//------------------------
class A : public Base
{ // Class A derives from Base
 public:
  void Func2();
  void Func4();
};
//------------------------
class B : public A
{// Class B derives from A
 public:
  void Func1();
  void Func4();
 };
//------------------------
class C : public Base
 { // Class C derives from Base
 public:
  void Func1();
   void Func4();
 };
//------------------------
```

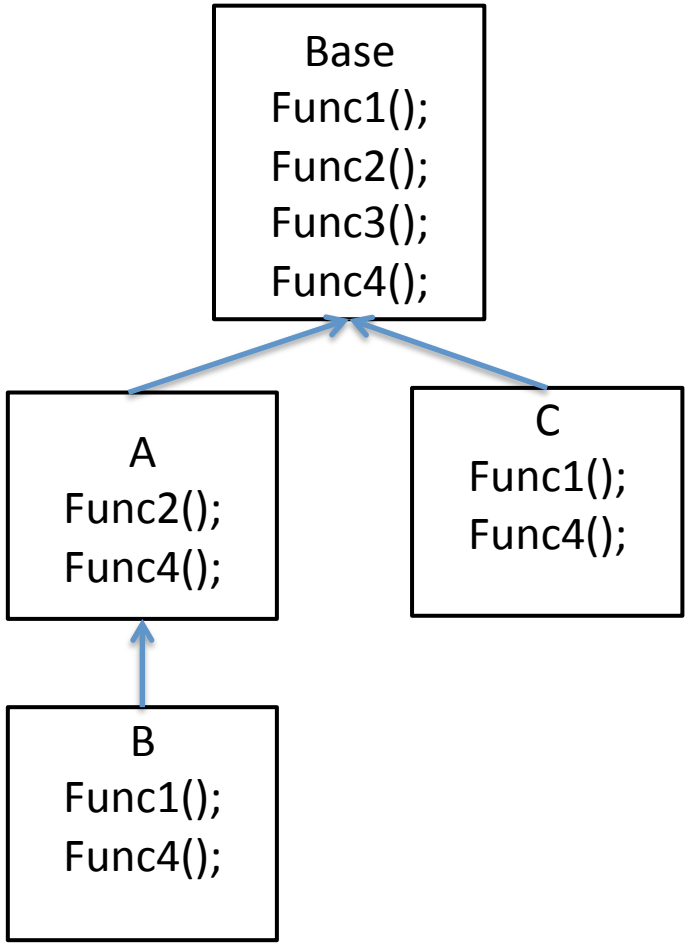
Example: Access Through Pointers




```
//Access through its name handler
Base BaseObject;
A  AObject;
B  BObject;
C  CObject;

//Access through its own pointer
Base * BasePtr = &BaseObject;
A * APtr = &AObject;
B * BPtr = &BObject;
C * CPtr = &CObject;

//Access through the base pointer
Base * BaseAPtr = &AObject;
Base * BaseBPtr = &BObject;
Base * BaseCPtr = &CObject;
```

InheritanceExample2.cc

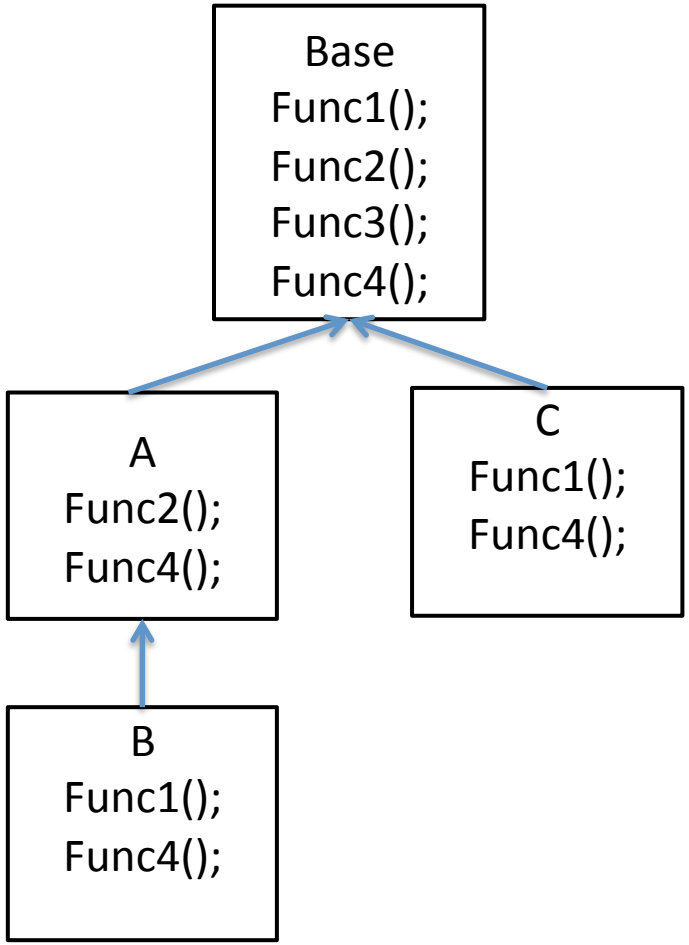
Example: Access Through Pointers




```
//Let's call Func1() from every object through
//its own pointer

  BasePtr->Func1();
  APtr->Func1();
  BPtr->Func1();
  CPtr->Func1();

//Let's call Func2() from every object through
//its own pointer

  BasePtr->Func2();
  APtr->Func2();
  BPtr->Func2();
  CPtr->Func2();
```

InheritanceExample2.cc

Example: Access Through Pointers



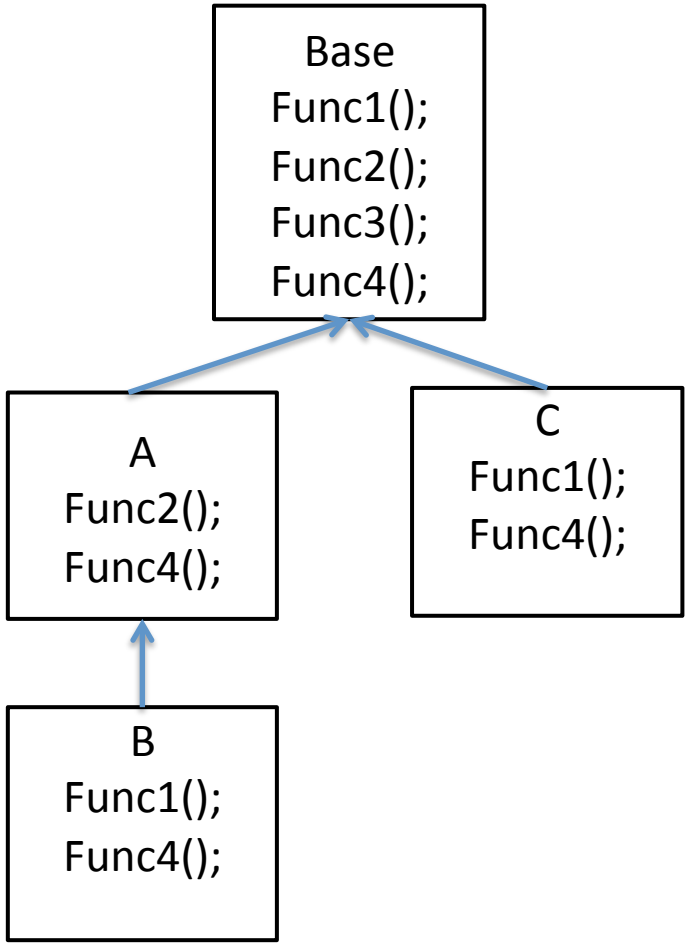
Call the Func1() through its own pointer
Hello from Base::Func1()
Hello from Base::Func1()
Hello from B:Func1()
Hello from C:Func1()

Call the Func2() through its own pointer
Hello from Base::Func2()
Hello from A:Func2()
Hello from A:Func2()
Hello from Base::Func2()

Note: This is the same as accessing through the name handle

InheritanceExample2.cc

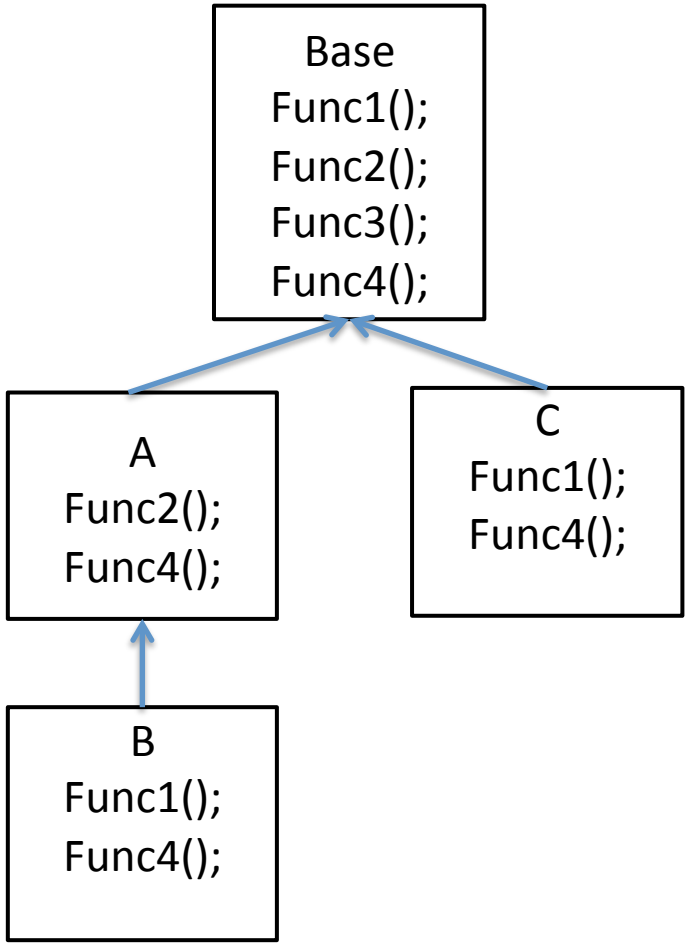
Example: Access Through Pointers




```
//Let's call Func1() from every object through
// the BASE Pointer
   BasePtr->Func1();
   BaseAPtr->Func1();
   BaseBPtr->Func1();
   BaseCPtr->Func1();

 //Let's call Func2() from every object through
//the BASE Pointer
   BasePtr->Func2();
   BaseAPtr->Func2();
   BaseBPtr->Func2();
   BaseCPtr->Func2();
```

InheritanceExample2.cc

Example: Access Through Pointers



Call the Func1() through the base pointer
Hello from Base::Func1()
Hello from Base::Func1()
Hello from Base::Func1()
Hello from Base::Func1()

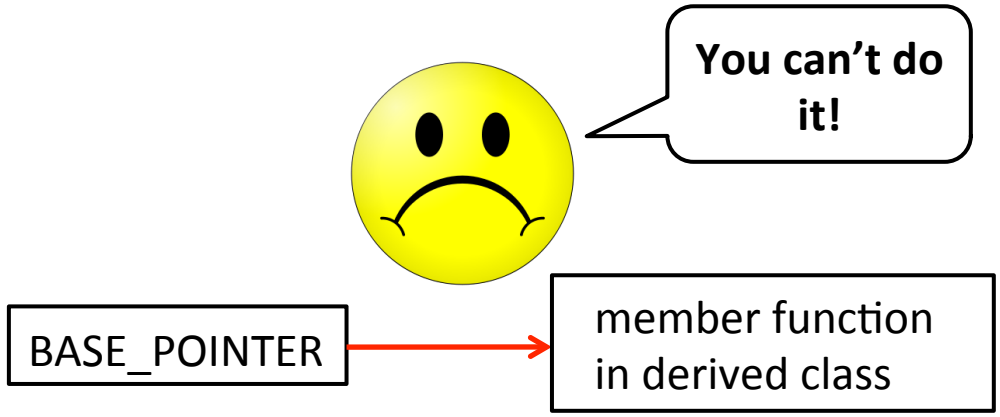
Call the Func2() through the base pointer
Hello from Base::Func2()
Hello from Base::Func2()
Hello from Base::Func2()
Hello from Base::Func2()

References have the same rules!!

InheritanceExample2.cc

Is it useful to get around this?

Is there a way around this?

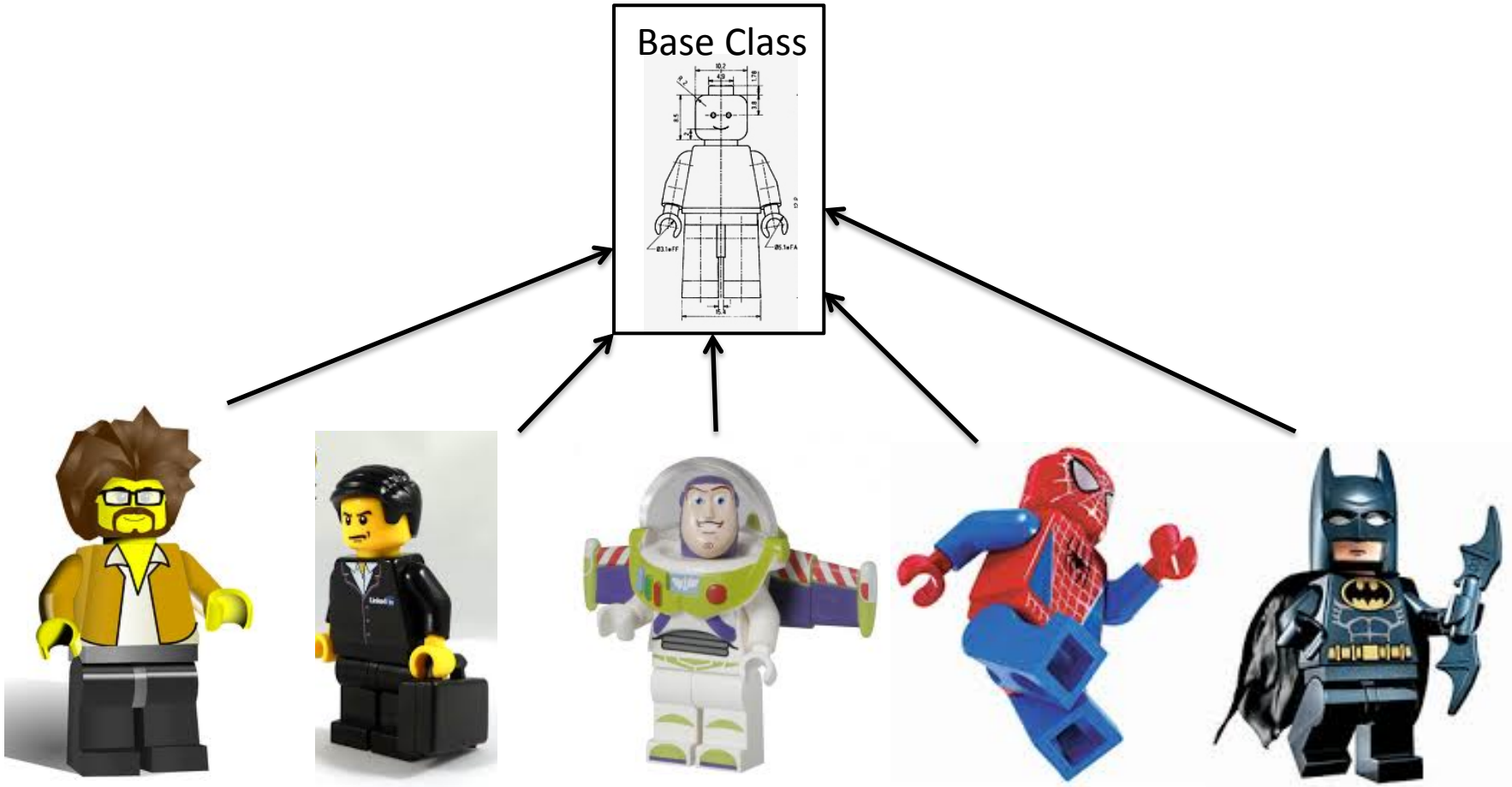


What time is class over?

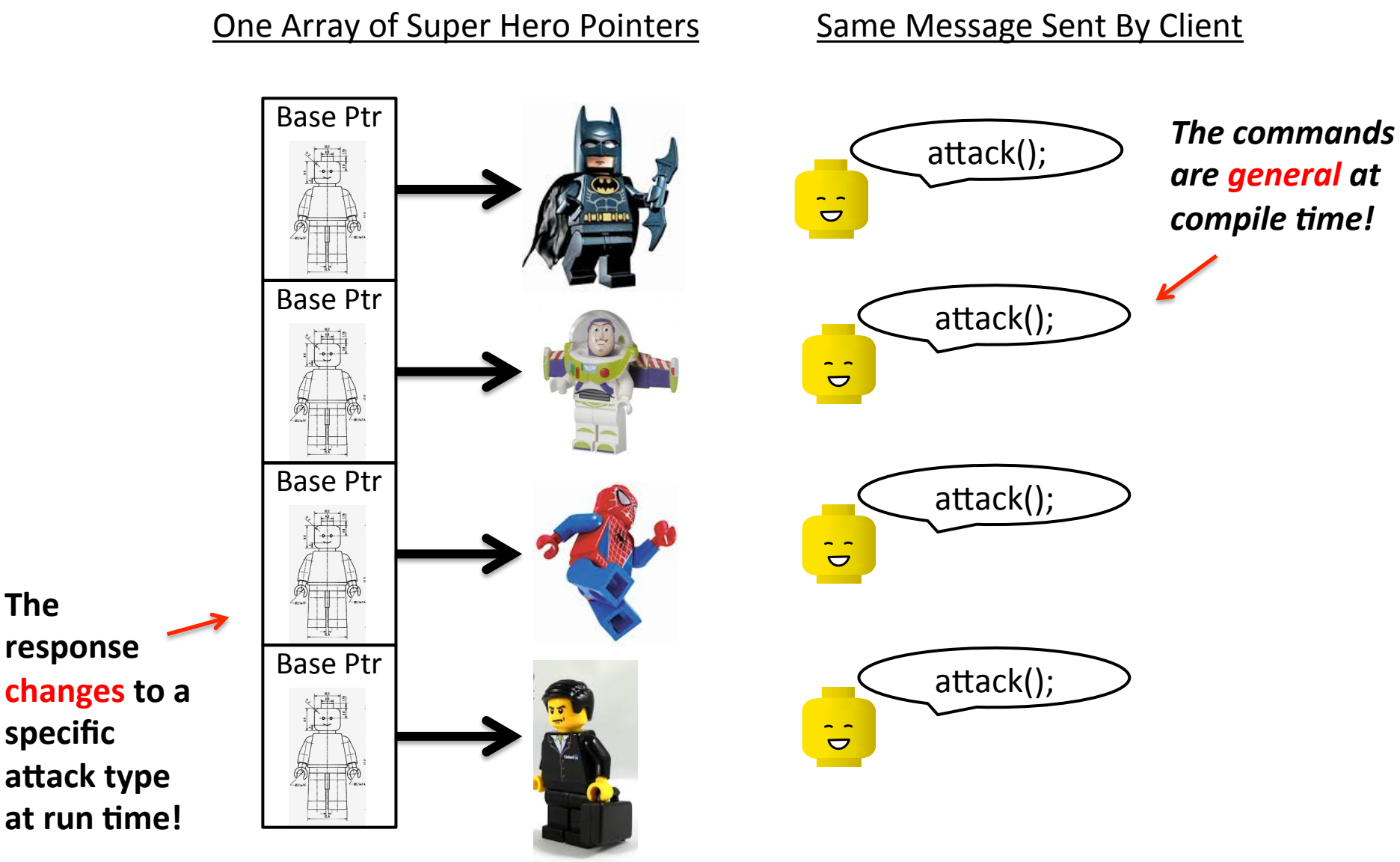
Polymorphism and Virtual Functions

Polymorphism Ideas

Assume that each of the derived classes has a member function called -- attack();

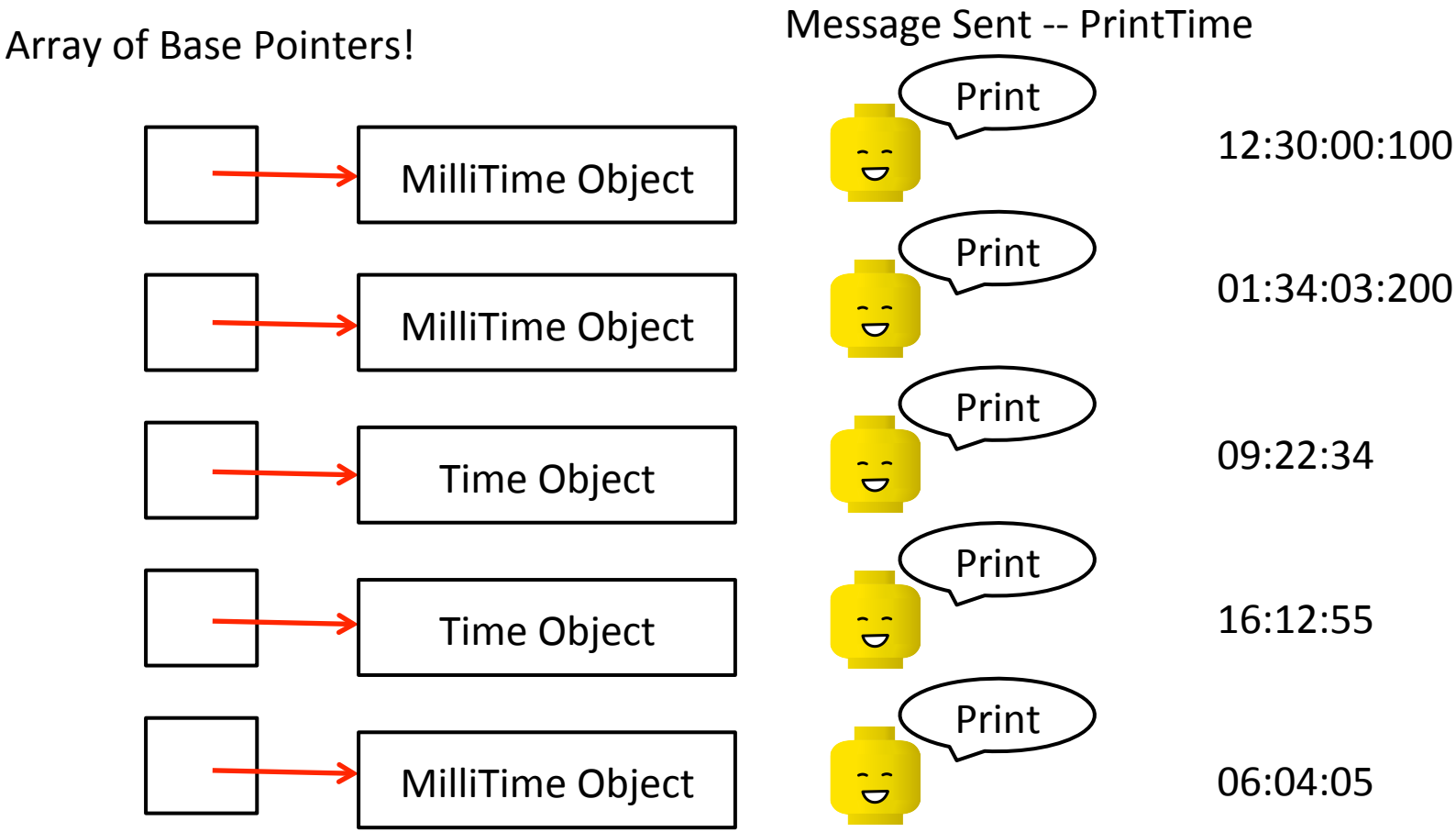


Polymorphism Ideas



Polymorphism Idea

You can send the same message to to a variety of objects and get a variety of results.

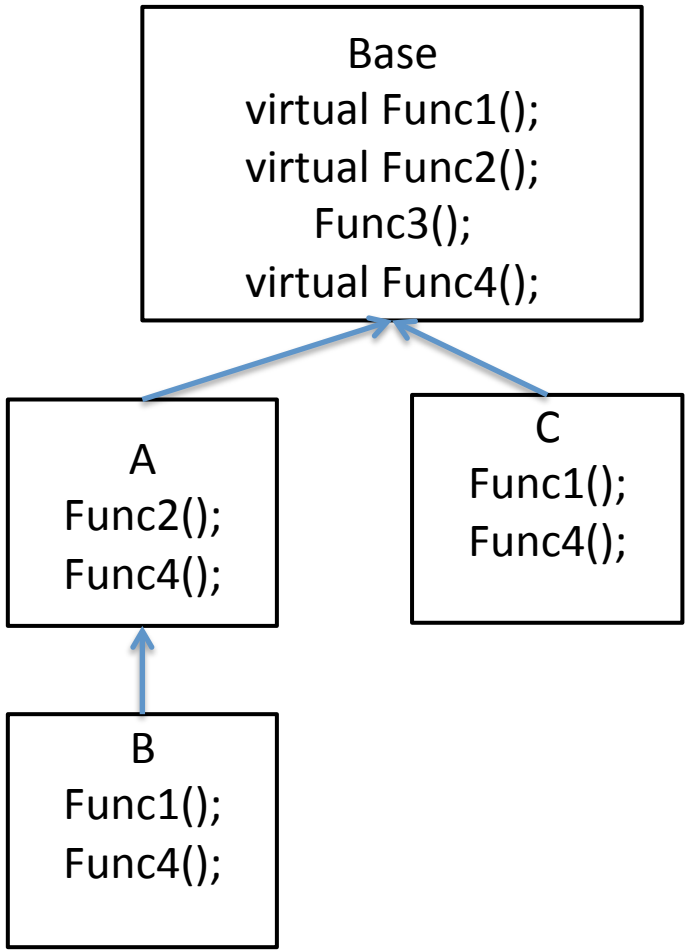


The 'point' is that you need to call the correct PrintTime function that is called through the base pointer!!

Virtual Functions enable access to DERIVE CLASS member functions THROUGH THE BASE POINTER

We declare a virtual function by preceding the function's prototype with the key-word `virtual` in the base class.

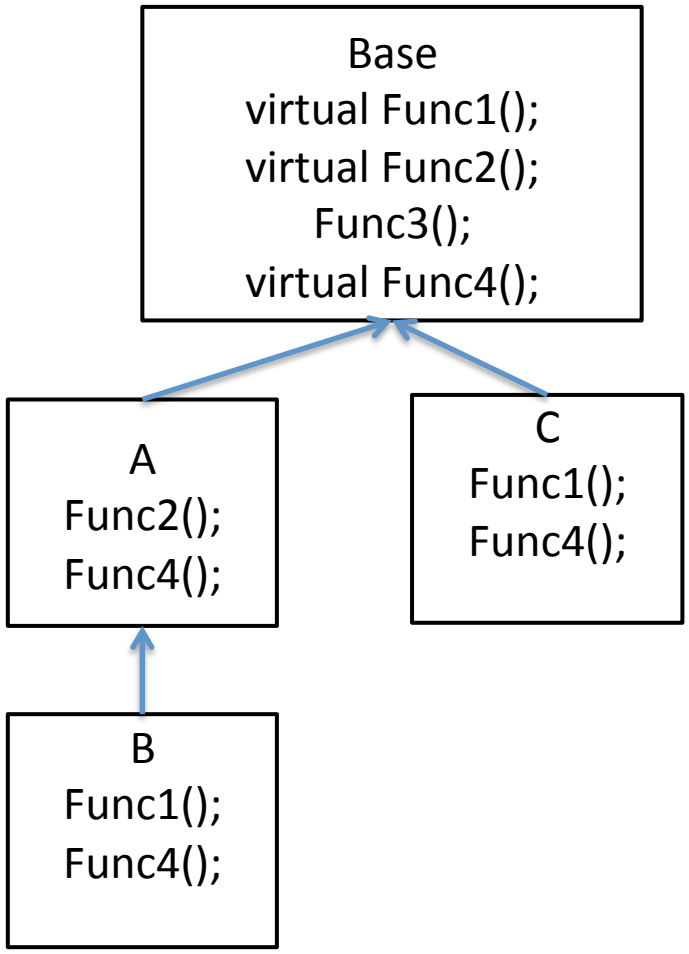
Example: Virtual Functions




```
class Base
{ // Define a base class
 public:
  virtual void Func1();
  virtual void Func2();
  void Func3();
  virtual void Func4();
};
//------------------------
class A : public Base
{ // Class A derives from Base
 public:
  void Func2();
  void Func4();
};
//------------------------
class B : public A
{// Class B derives from A
 public:
  void Func1();
  void Func4();
 };
//------------------------
class C : public Base
 { // Class C derives from Base
 public:
  void Func1();
  void Func4();
 };
```

virtualExample1.cc

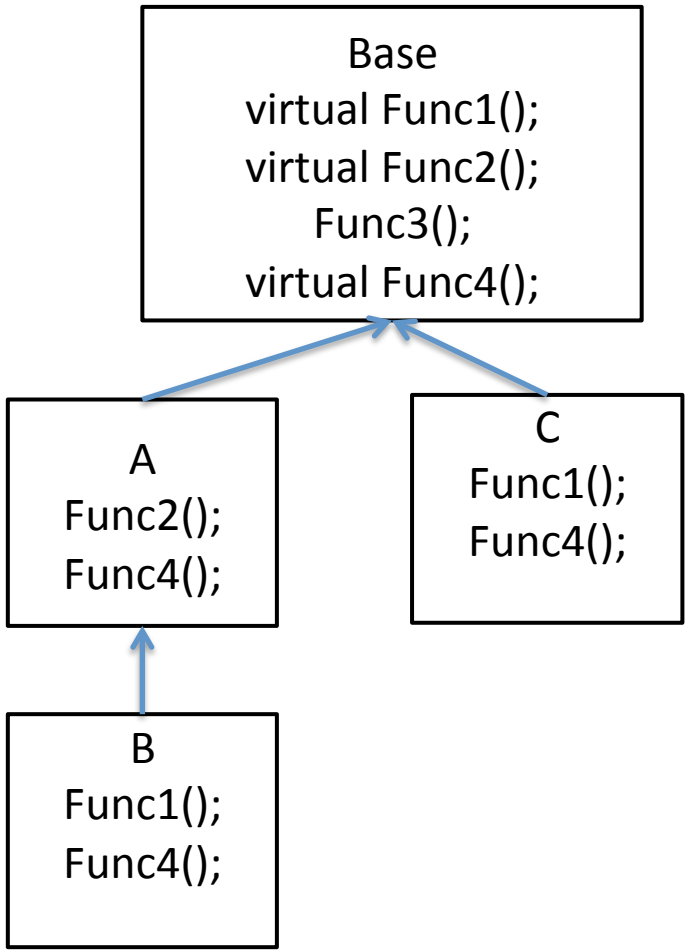
Example: Virtual Functions




```
Base BaseObject;
 A  AObject;
 B  BObject;
 C  CObject;

 Base * BasePtr = &BaseObject;
 A * APtr = &AObject;
 B * BPtr = &BObject;
 C * CPtr = &CObject;

 Base * BaseAPtr = &AObject;
 Base * BaseBPtr = &BObject;
 Base * BaseCPtr = &CObject;
```

virtualExample1.cc

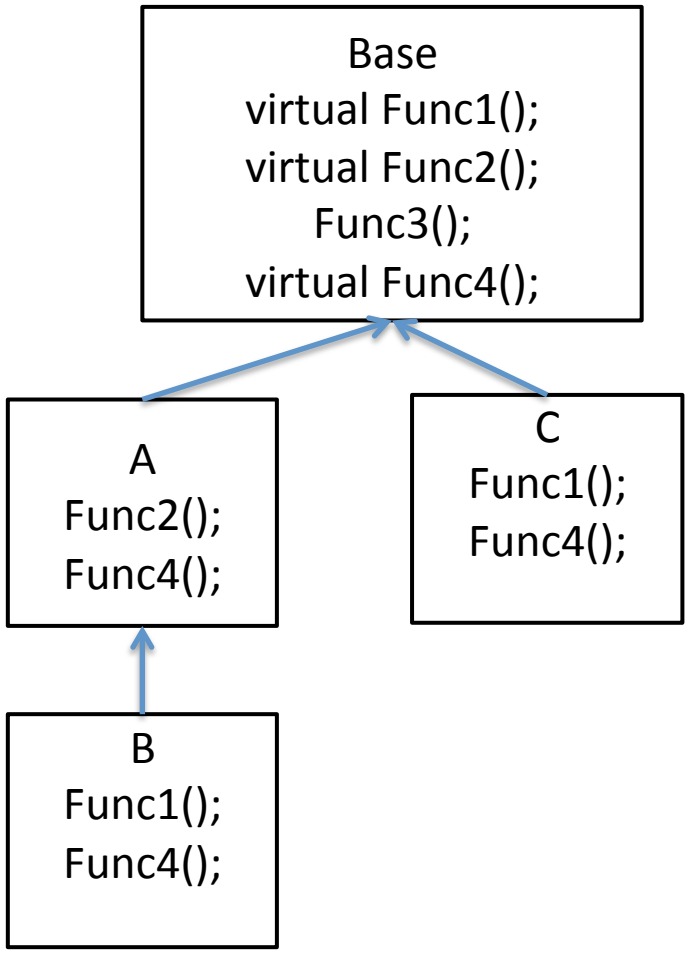
Example: Virtual Functions




```
 //Let's call Func1() from every object through its own pointer
   BasePtr->Func1();
   APtr->Func1();
   BPtr->Func1();
   CPtr->Func1();

   //Let's call Func2() from every object through its own pointer
   BasePtr->Func2();
   APtr->Func2();
   BPtr->Func2();
   CPtr->Func2();
```

virtualExample1.cc

Example: Virtual Functions

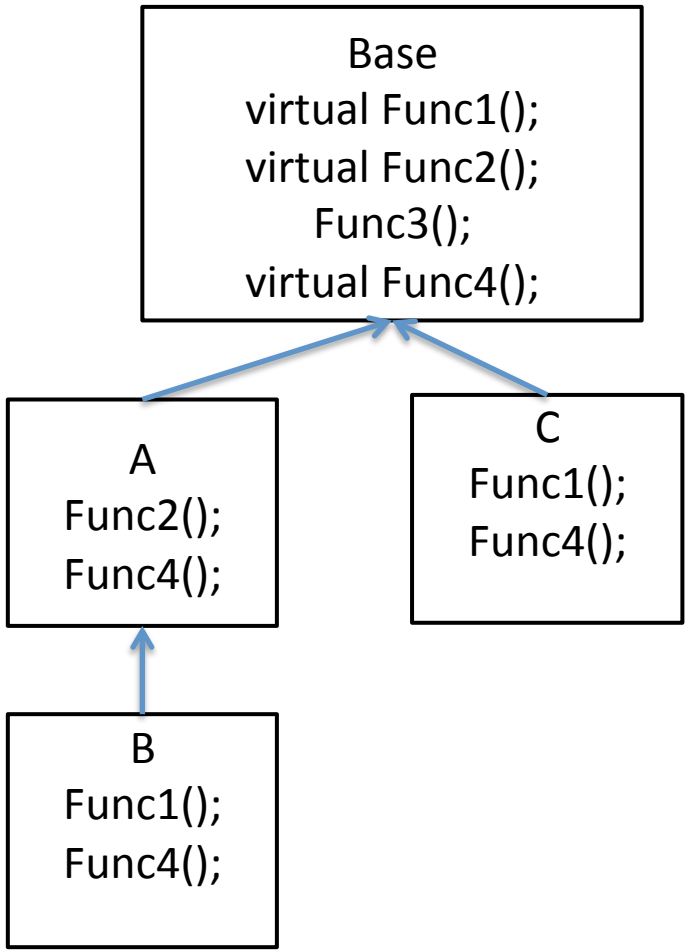


Call the Func1() through its own pointer
Hello from Base::Func1()
Hello from Base::Func1()
Hello from B:Func1()
Hello from C:Func1()

Call the Func2() through its own pointer
Hello from Base::Func2()
Hello from A:Func2()
Hello from A:Func2()
Hello from Base::Func2()

virtualExample1.cc

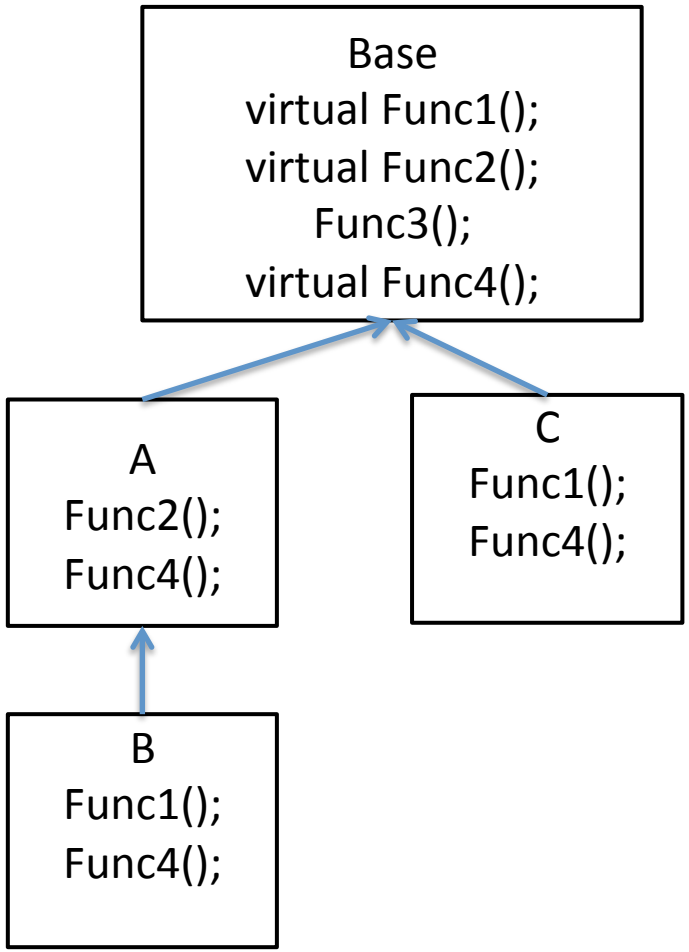
Example: Virtual Functions




```
//Let's call Func1() from every object through the BASE Pointer
  BasePtr->Func1();
  BaseAPtr->Func1();
  BaseBPtr->Func1();
  BaseCPtr->Func1();

  //Let's call Func2() from every object through the BASE Pointer
  BasePtr->Func2();
  BaseAPtr->Func2();
  BaseBPtr->Func2();
  BaseCPtr->Func2();
```

virtualExample1.cc

Example: Virtual Functions

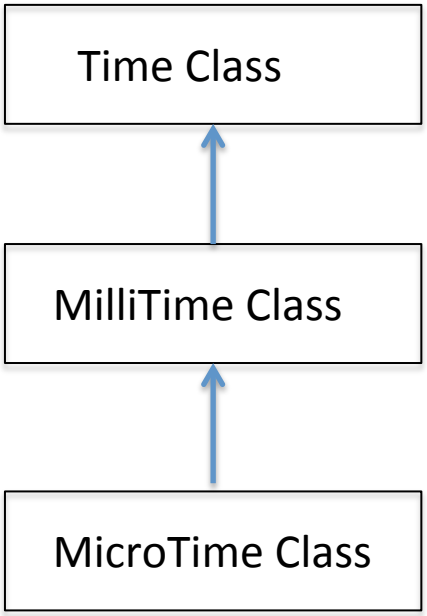


Call the Func1() through the base pointer
Hello from Base::Func1()
Hello from Base::Func1()
Hello from B:Func1()
Hello from C:Func1()

Call the Func2() through the base pointer
Hello from Base::Func2()
Hello from A:Func2()
Hello from A:Func2()
Hello from Base::Func2()

virtualExample1.cc

Example 3-Levels



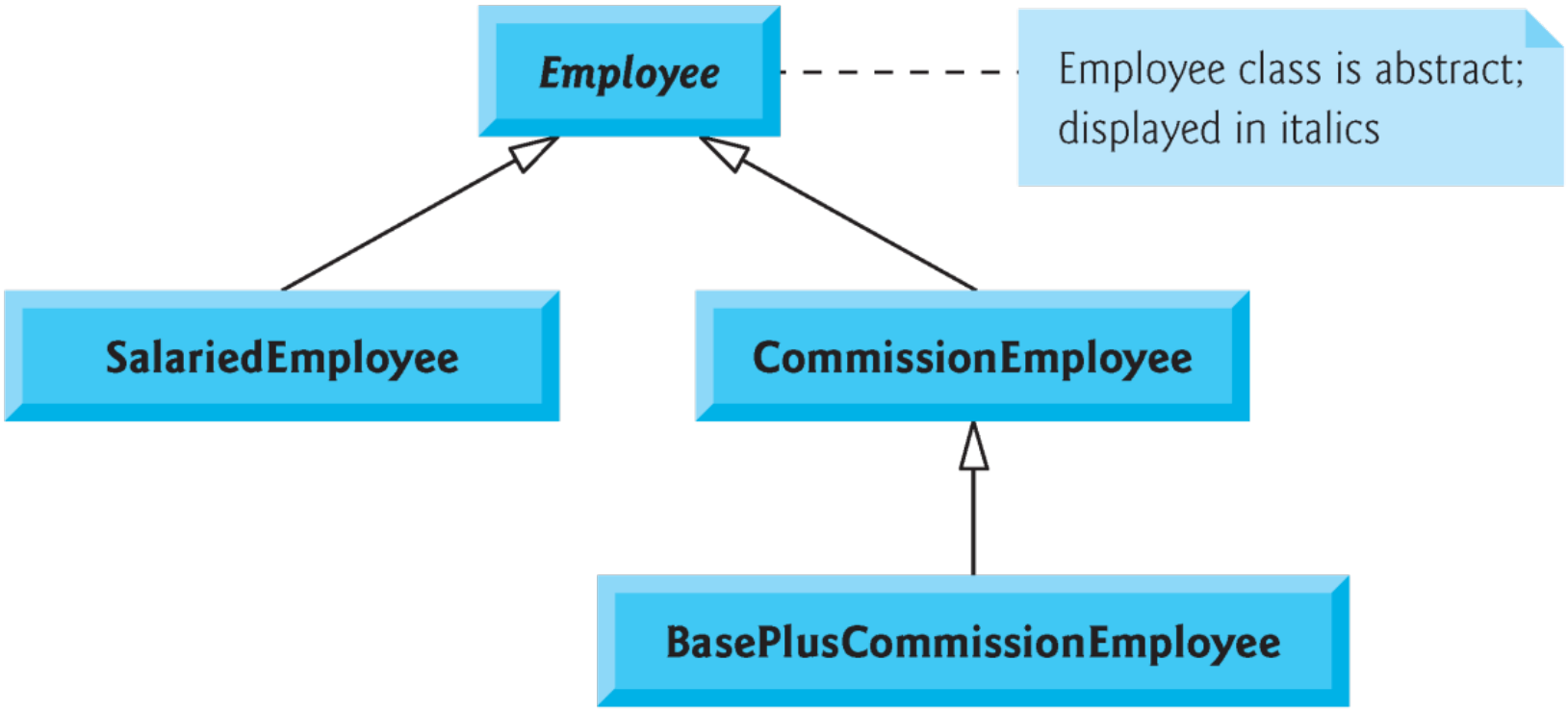
ThreeLevelHierarchy.cc

Illustrate calling through a pointer and a reference without virtual functions!

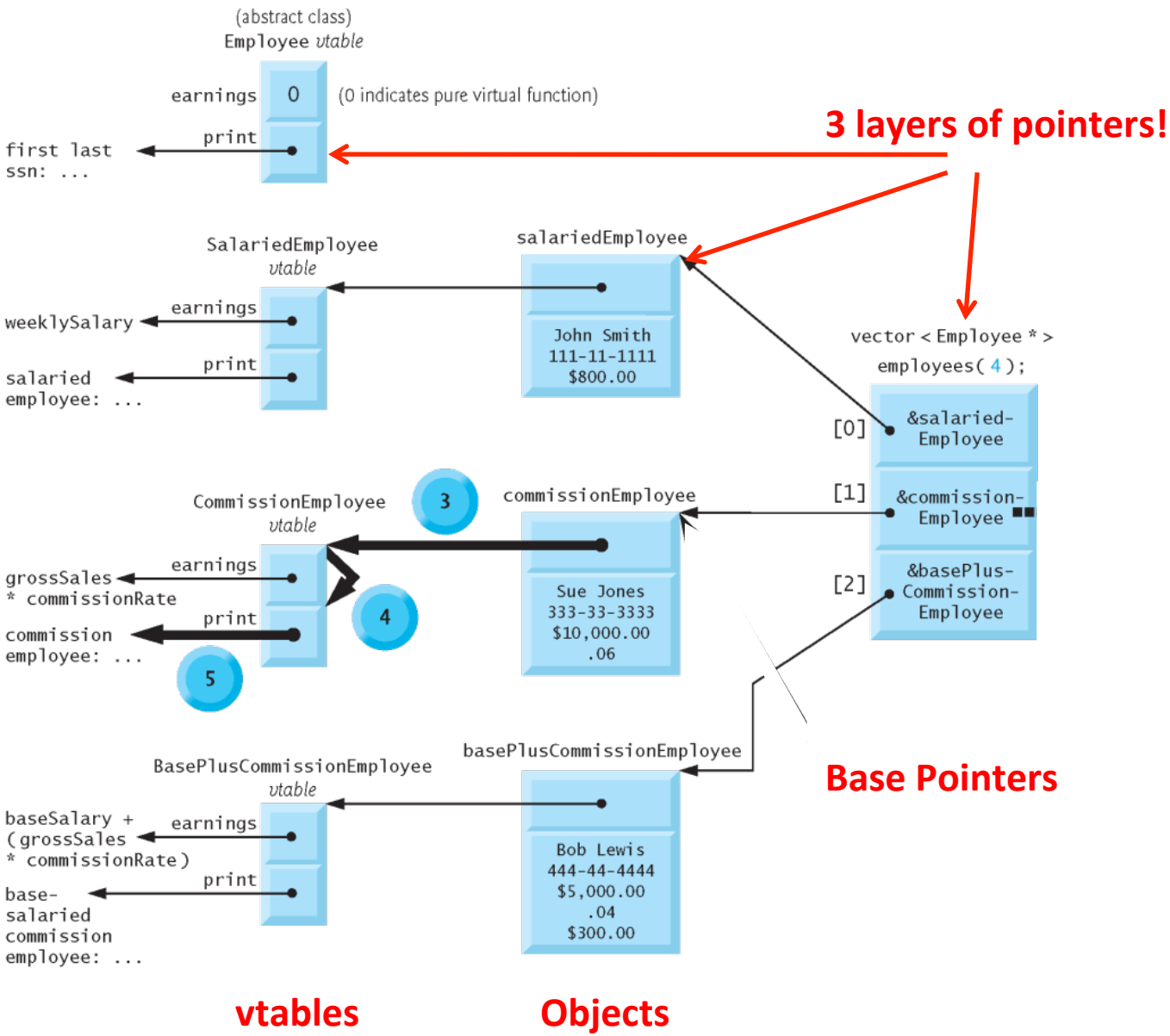
Result: Only Base Functions can be called through Base Pointer!

You can try this one on your own!!!

vTables and Virtual Functions



vTables and Virtual Functions



Static vs. Dynamic Binding

(resolved at compile time)

Static Binding

```
baseObject.memberFunction();
```

(resolved at run time)

Dynamic Binding

```
basePtr->virtualFunction();
```

Abstract Classes vs. Concrete Classes

Example: Clash of Clans

Screen Object Types: Villagers

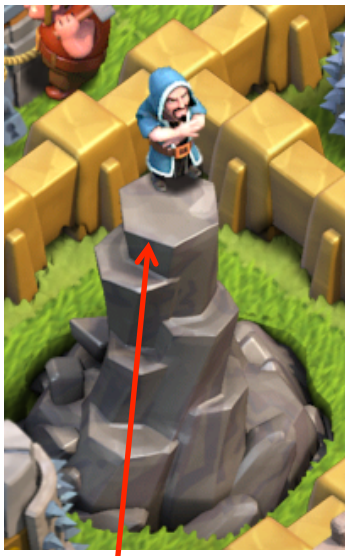
Women

Men

Screen Object Types: Warriors

Archers

Giants



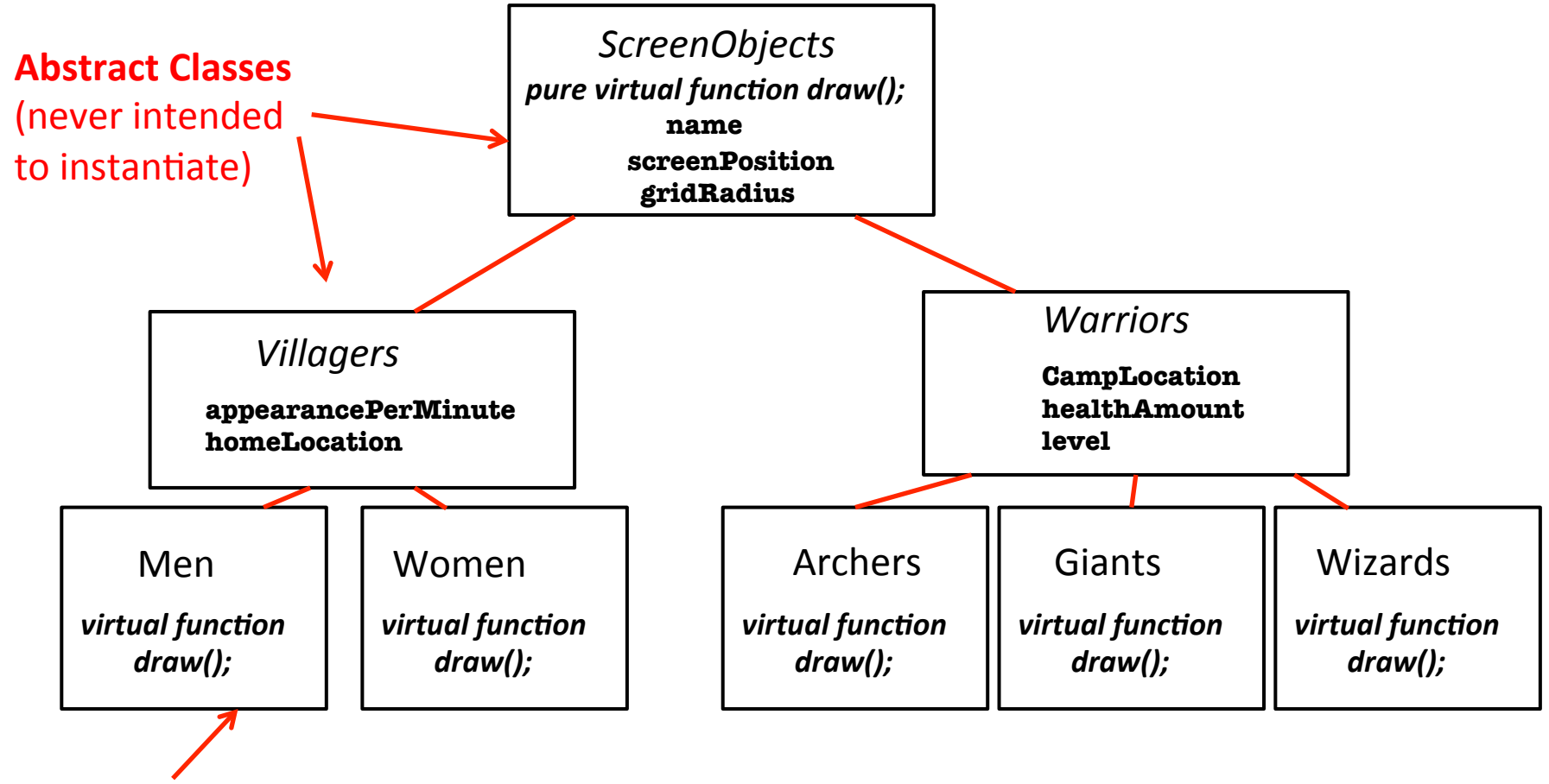
Wizards

Character Types: Warriors

Army Camps

Polymorphism Examples

Example: Video game creation and enhancement – "Clash of the Clans"



All concrete classes must have OR INHERIT an implementation of the pure virtual function!!

Syntax for pure virtual functions

You do NOT need an implementation of *.cc file!

virtual return_type pureVirtualFunctionName() =0;

```
class ScreenObject
{
public:
...
virtual void draw(Frame) = 0;

private:
string name;
Position screenPosition;
int gridRadius;

};//end of ScreenObject
```

This syntax tells the compiler that this is a PURE virtual function.

Polymorphism Motivation

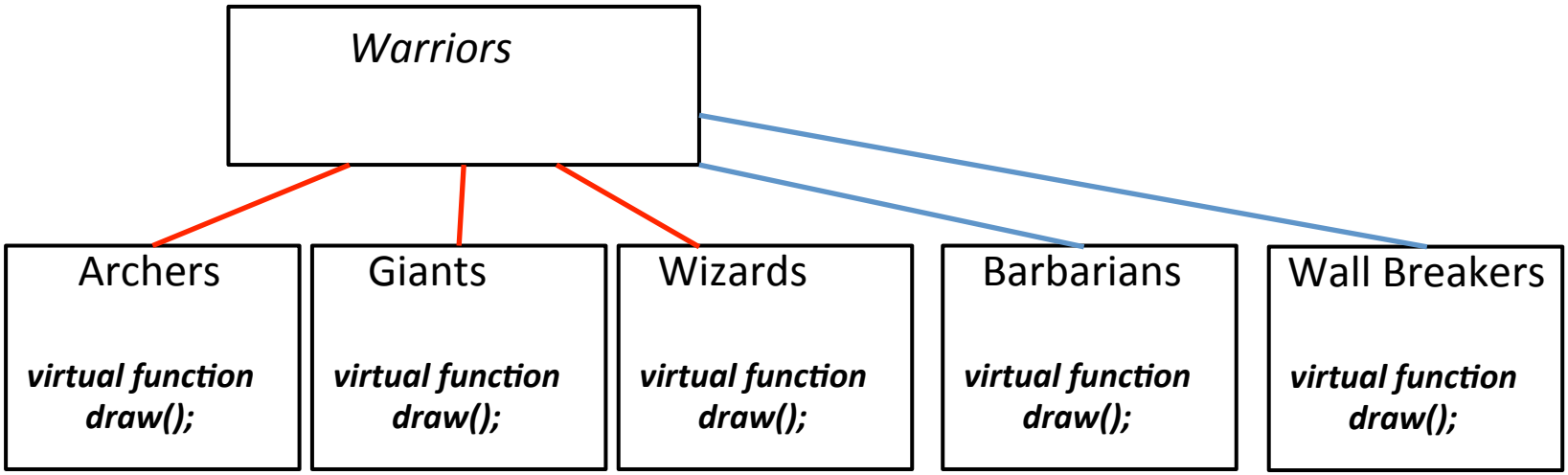
The idea is that the code you create to make the game work needs to be created GENERALLY to work with these polymorphic objects.

WHY? → Ease of code enhancement!

For example, if you add characters to your game that you will not have to change significant portions of the main part of your code… just the added object!!

Enhancement with Polymorphism

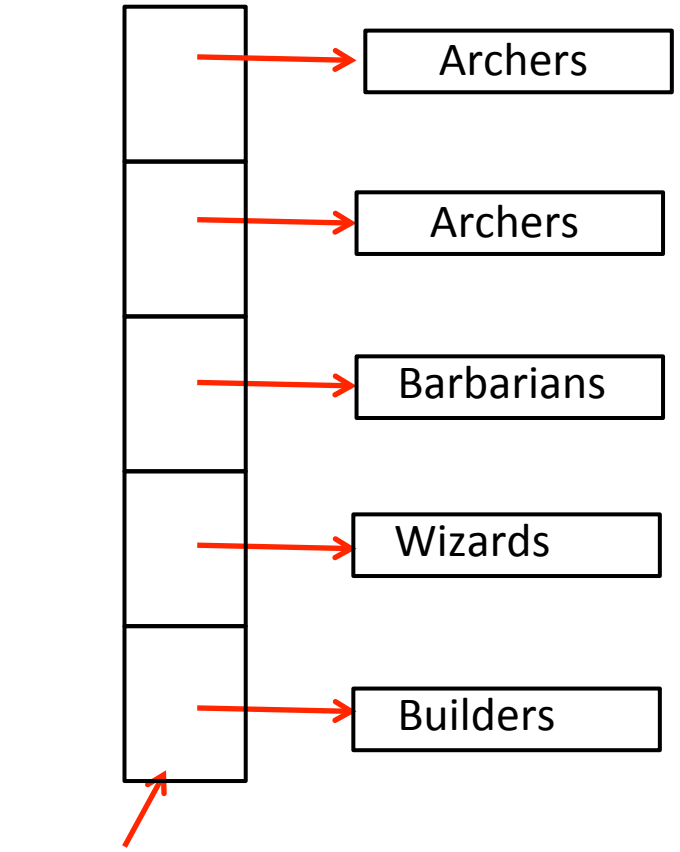
Example: Video game creation and enhancement – "Clash of the Clans"



We would like to add these two new objects with MININAL changes to the system's code. Polymorphism can allow us to "plug in" our new classes without modification to the screen manager code.

General Code Structure Comment

Vector of ScreenObject Pointers!



(i.e. base pointers)

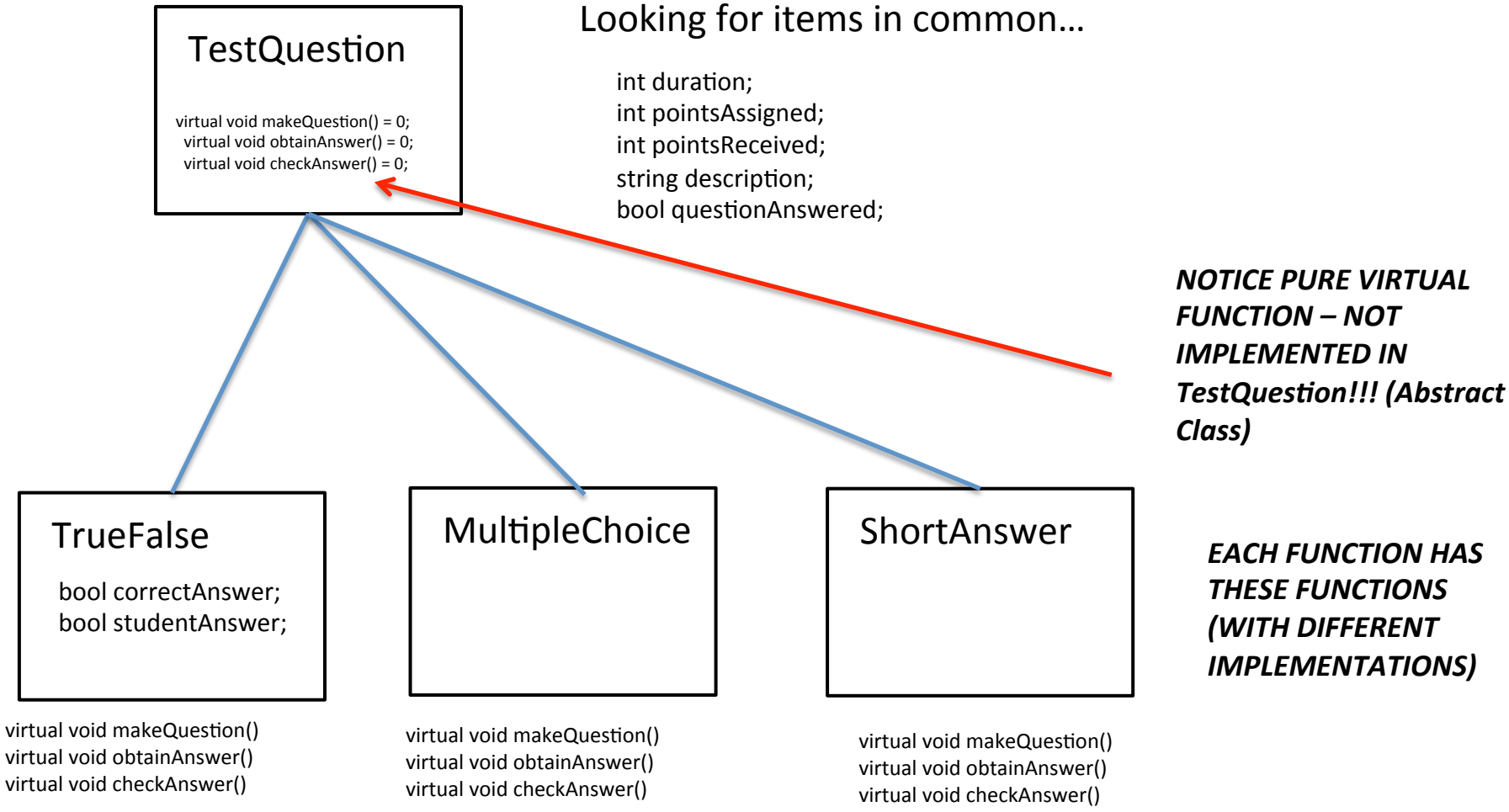
notice code becomes very "generic" and won't change as I add characters too much!

```
DrawBackground();

//draw all objects on the screen
   for(i=0; i < ScreenObjectVector.size; ++i)
        ScreenObjectVector[i] -> draw()
```

```
//character destroyed (at d_index)
 delete ScreenObjectVector[d_index];
 ScreenObjectVector.erase(d_index);
```

```
//wizard character "instantiated"
NewScreenObjectPtr = new  wizard();
ScreenObjectVector.push_back(NewScreenObjectPtr);
```

Example: TestQuestion Hierarchy



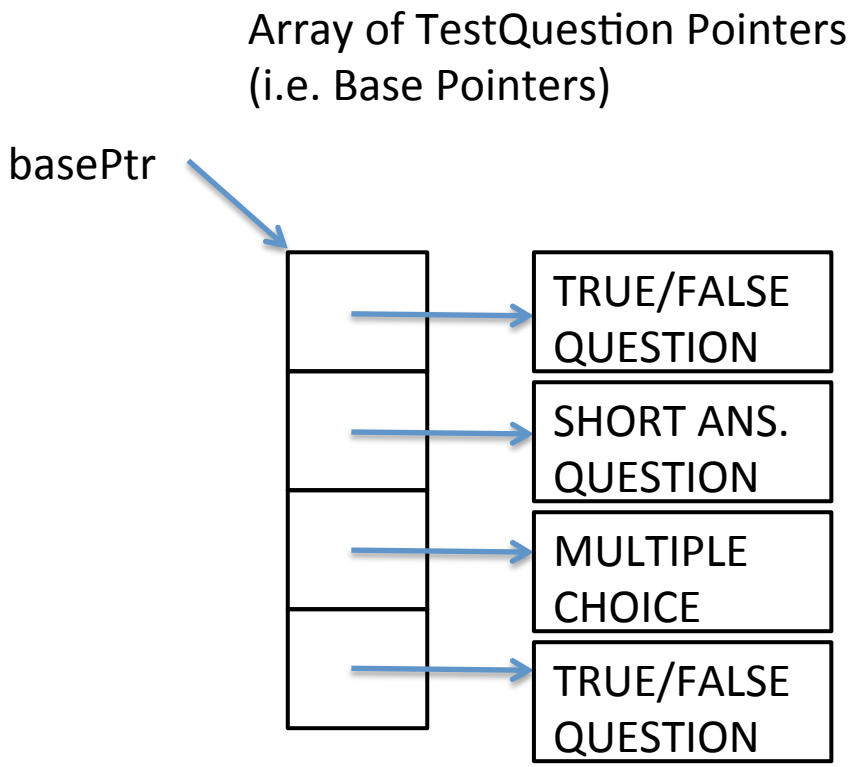
Example: TestQuestion Hierarchy

```
class Test
{
//Note this has memory leaks that need to be corrected!!

//--- constructors
  public:
  Test(string testname, int numQ)
  {
     name = testname;
     numberQuestions = numQ;
     basePtr = new TestQuestion*[numQ];
  }


//--- member functions
  void makeTest();
  void takeTest();
  void gradeTest();
  void printOptions();
  void allocateQuestion(int,int);

  private:
  string name;
  int numberQuestions;
  TestQuestion ** basePtr; //dynamically allocated array pointers

}; // This is the end of the test
```

```
int main()
{

Test test2("ECE2036", 3);

test2.makeTest();
test2.takeTest();
test2.gradeTest();

return 0;
}
~
```

Example: TestQuestion Hierarchy

```
void Test::takeTest()
{
//now go through the array polymorphically
for (int i=0; i < numberQuestions; i++)
{
   cout << "Question " << i+1 << ": " ;
   (*(basePtr+i))->obtainAnswer();
}
} //takeTest
```



Textbook Software Engineering Observations

"Polymorphism promotes extensibility: Software that invokes polymorphic behavior is independent of the object types to which messages are sent. New object types that can respond to existing method calls can be incorporated into a system without requiring modification of the base system. Only client code that *instantiates new objects* must be modified to accommodate new types."