PY410 / 505
Computational Physics 1

Salvatore Rappoccio

Code

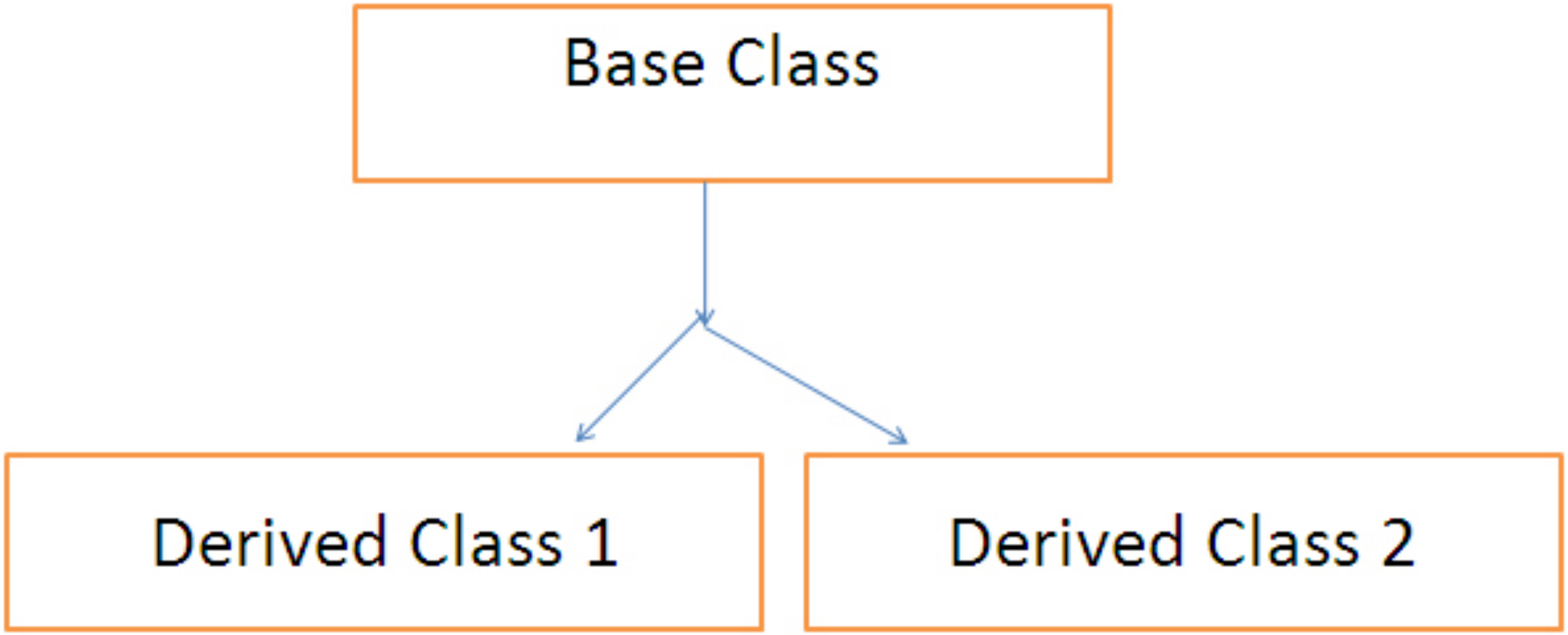
- Code is in
  - CompPhys/ReviewCpp/InheritanceExamples
  - CompPhys/ReviewCpp/Maps
  - CompPhys/ReviewCpp/TemplateExamples

Inheritance

C++: Inheritance

- You've already seen inheritance:
  - std::istream is a PARENT of std::ifstream

- Base (parent) classes are more general
- Derived (child) classes are more specific



C++: Inheritance

- Distinguish between “has a” and “is a” relationships
- Example:
  - A Honda HAS A motor
  - A Honda IS A car
- Now: make this into a data model

C++: Inheritance

- Example: StudentRecord

- From your homework, you looked at StudentRecord.

- StudentRecord HAS A first name
- StudentRecord HAS A last name
- StudentRecord HAS A score

```
class StudentRecord {
 . . .
  std::string firstname_;
  std::string lastname_;
  double score_;

};
```

C++: Inheritance

- Now try to make this into an interface with SPECIALIZED attributes

- Suppose we have a special StudentRecord, i.e. a record with a student's score in a specific class (say, Physics).

- How do we generalize?
  - Make a BASE CLASS out of StudentRecord
  - Make DERIVED CLASSES to implement the specifics

C++: Inheritance

- Syntax:

```
class StudentRecord {
  . . .
};
```

Base Class

```
class StudentRecordForAlgebra :
      public StudentRecord {
  . . .
};
```

Derived Class

Inherits PUBLICLY from StudentRecord
(can have private but don't worry about it)

C++: Inheritance

- What's the use?
  - Generalization!

- Suppose you have a base class, "Car", and two derived classes "Honda" and "Chevy". You want to write code to have a robot fill up the gas tank.

- The logic is really about Cars, not Hondas or Chevies.
  - So something like:

```
class Car {
   bool fill(Gas const & gas);
};
class Chevy : public Car {???};
class Honda : public Car {???};
```

C++: Inheritance

- Syntax is to make the function VIRTUAL in the base class:

```
class Car {
   virtual bool fill(Gas const & gas);
};
```

- Then OVERRIDE them in the derived classes and fill in relevant details:

```
class Chevy : public Car {
   virtual bool fill(Gas const & gas){
      fillLeftSide(gas);
   }
};
class Honda : public Car {
   virtual bool fill(Gas const & gas){
      fillRightSide(gas);
   }
};
```

C++: Inheritance

- In C++:
  - Use “virtual” keyword

```
class Chevy : public Car {
   virtual bool fill(Gas const & gas){
      fillLeftSide(gas);
   }
};
```

- In C++11 and later: can use “override” keyword:

```
class Chevy : public Car {
   bool fill(Gas const & gas) override {
      fillLeftSide(gas);
   }
};
```

This is a bit stricter, requires that the function be exactly the same as a virtual function in the base class.

C++: Inheritance

- Now we understand “public”, “protected”, and “private”:

- Public: available to all code
- Protected: available to “this” class and any derived classes
- Private: available only to “this” class

C++: Inheritance

- With inheritance, you can use polymorphism:
  - Call code from the DERIVED class by accessing through the BASE class

```
std::vector< Car *> cars;
Chevy malibu;
Honda accord;
cars.push_back( &malibu );
cars.push_back( &accord );
```

Can make BASE CLASS POINTERS to the derived class objects

```
for ( Car * pcar : cars ) {
   pcar->Fill(gas);
}
```

Then treat them the same way in code without having to know specific details!

C++: Inheritance

- This works through the VIRTUAL TABLE (vtable)
  - Pointer to code that is defined at RUN TIME
    - "Dynamic binding" or "late binding"
    - To call derived classes, need to DEREFERENCE the pointer in the vtable
      - This can sometimes lead to poorer performance… more later
  - Depending on run-time value of objects, looks at a different class
- Example:

```
std::vector< Car *> cars;
Car * pcar_1 = get_car_from_user();
Car * pcar_2 = get_car_from_user();
cars.push_back( pcar_1 );
cars.push_back( pcar_2 );


for ( Car * pcar : cars ) {
   pcar->Fill(gas);
}
```

C++: Inheritance

- If your base class doesn't actually refer to a real type, it is a PURE INTERFACE
- Also known as an "abstract base class"

- In our "Car" example, there is no such model of "Car" that is just "Car". You need to have a Honda or a BMW or a Toyota or a Chevy or a Ford.
  - Therefore Car should be abstract!

- Syntax: provide virtual DECLARATION of a function, with no DEFINITION, set it equal to zero:

```
class Car {
   virtual bool fill(Gas const & gas) = 0;
};
```

C++: Inheritance

- Example of generalizing StudentRecord:

- Suppose we have a specialized cases of StudentRecord:
  - StudentRecordPhysics : inputs TWO scores
  - StudentRecordLiterature : inputs THREE scores

- To make generic: make “base” class have a vector<double> to represent scores

Friends

C++: Friend Classes

- Data access:
  - Protected : derived classes

- What about OTHER classes or functions you want to be able to access?
  - Friends!

  - Yes, really.

- One common use : operator<< and operator>>

C++: Friend classes

- Syntax is pretty simple: in your class definition, just like the Mines of Moria:

- speak friend, and enter

```
class A{
public:

   friend class B;
};
```

B now has access to all of A's private and protected data

C++: Friend Classes

- Often use for operator<< and operator>>:

- In header (.h) file:

```
class StudentRecord {
 public:
    . . .

 friend std::ostream & operator<<( std::ostream & out, StudentRecord const & );
  friend std::istream & operator>>( std::istream & in , StudentRecord const &);
}
```

- In source (.cc) file:

```
std::ostream & operator<<( std::ostream & out, StudentRecord const &right )
{ right.print(out); return out; }

std::istream & operator>>( std::istream & in, StudentRecord  &right )
{ right.input(in); return in; }
```

Hands on

- Folder “InheritanceExample”

- Type “make” and you get the executable

Class Templates

C++: Templates

- We’ve been using templates since last week:
- Operates on ANY class
- std::vector<int>, std::vector<double>, std::vector<StudentRecordPhysics>, etc

- Now let’s look a bit deeper

C++: Class Templates

Cookie template

Cookie

Class template

```
template< class T>
class Storage
```

```
Storage<int> s( 2 );
```

Class

C++: Class Templates

- Similar to function templates, declare with “template <class T>”

- Then the code you write has a PLACEHOLDER value called “T”. “T” is not a class. It is a dummy. It does not exist.

- This defines an INTERFACE to operate on the class object

C++: Class Templates

- So std::vector<T> is a template

- It doesn’t matter if “T” is a float or an int (or a StudentRecord), the operations of “std::vector” are unchanged!

- Class template std::vector<T>, then make classes like std::vector<int>, std::vector<StudentRecordPhysics>, etc.

C++: Class Templates

- If you do not correctly set up your templates, you will get sixty-seven pages of compiler warnings
- It issues the warnings for EVERY instance of the class template
- Usually the first one is the one you care about, and it is actually usually descriptive, so READ THEM!

C++: Class Templates

- Can also have MULTIPLE template parameters
- Example: Associative container “std::map”
  - Like a vector, but has an abstract KEY and can be sorted in any way you like “Compare”

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

- Example: “Maps”