8 Inheritance & Polymorphism

Inheritance – Introduction

- Inheritance is
  - **a technique to build a new class based on an old class**
- Example
  - Class employee holds employee personnel record

```
class Employee {
public:
  Employee(const char* name, double salary) ;
  const char* name() const ;
  double salary() const ;
private:
  string _name ;
  double _salary ;
} ;
```

  - Company also employs managers, which in addition to being employees themselves supervise other personnel
    - Manager class needs to contain additional information: list of subordinates
  - Solution: make Manager class that ***inherits*** from Employee

Inheritance – Syntax

- Example of Manager class constructed through inheritance

Declaration of public inheritance

```
class Manager : public Employee {
public:
  Manager(const char* name, double salary,
          vector<Employee*> subordinates) ;
  list<Employee*> subs() const ;
private:
  list<Employee*> _subs ;
} ;
```

Additional data members in Manager class

Inheritance and OOAD

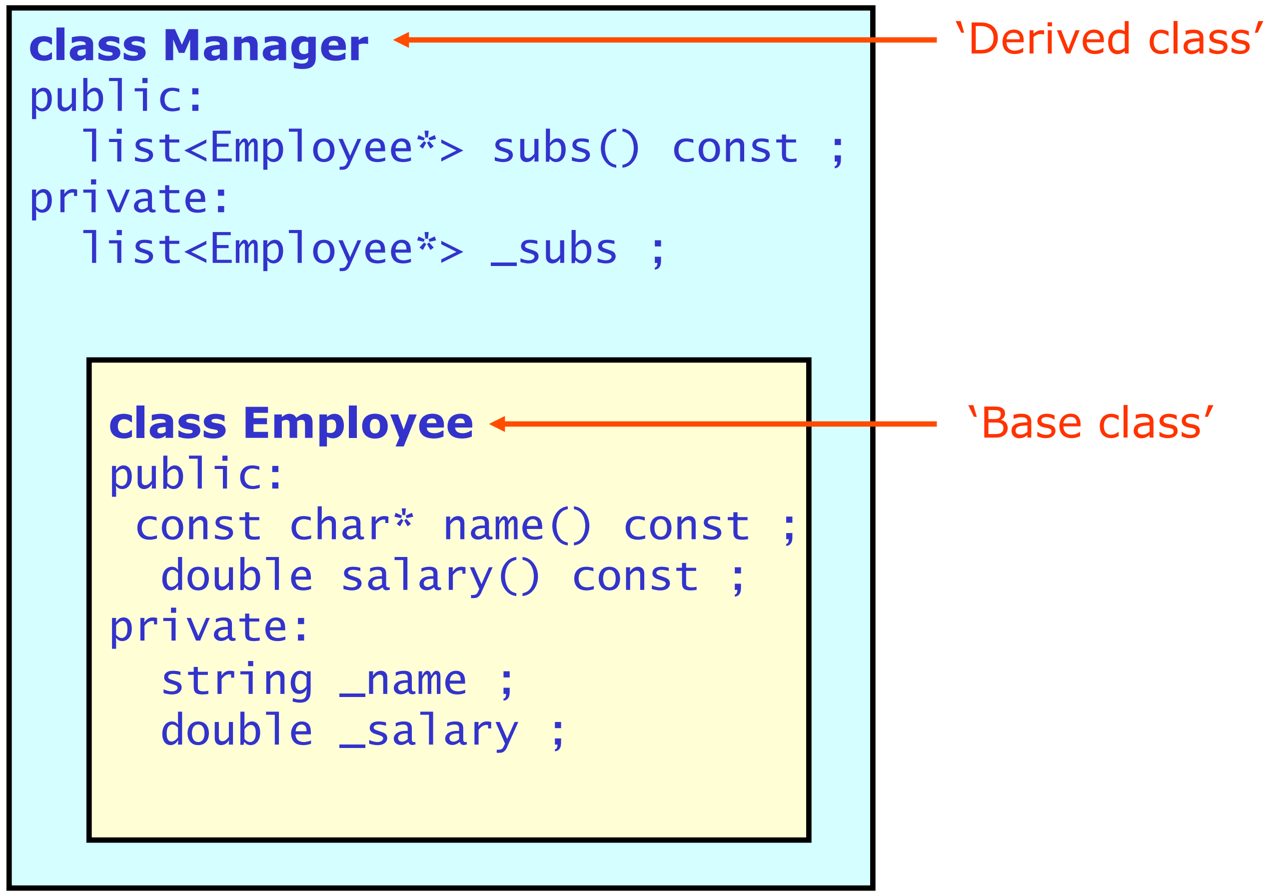
- Inheritance means: Manager **Is-A**n Employee
  - Object of class Manager can be used in exactly the same way as you would use an object of class Employee because:
  - class Manager also has all data members and member functions of class Employee
  - Detail: examples shows '*public inheritance*' – Derived class inherits *public interface* of Base class

- Inheritance offers new possibilities in OO Analysis and Design
  - But added complexity is major source for conceptual problems
  - We'll look at that in a second, let's first have a better look at examples

Inheritance – Example in pictures

- Schematic view of Manager class

Terminology



Inheritance – Example in pictures

- Inheritance can be used recursively

```
class Director
public:
    int numShares() const ;
private:
    int _numShares ;

    class Manager
    public:
      list<Employee*> subs() const ;
    private:
      list<Employee*> _subs ;

        class Employee
        public:
          const char* name() const ;
          double salary() const ;
        private:
          string _name ;
          double _salary ;
```

Terminology

- class Director: 'Derived class'
- class Manager: 'Derived class' & 'Base class'
- class Employee: 'Base class'

Inheritance – Using it

- Demonstration of Manager-IS-Employee concept

```
// Create employee, manager record
Employee* emp = new Employee("Wouter",10000) ;

list<Employee*> subs ;
subs.push_back(emp) ;

Manager* mgr = new Manager("Stan",20000,subs) ;


// Print names and salaries using
// Employee::salary() and Employee::name()
cout << emp->name() << endl ;    // prints Wouter
cout << emp->salary() << endl ;  // prints 10000

cout << mgr->name() << endl ;    // prints Stan
cout << mgr->salary() << endl ;  // prints 20000
```

Inheritance – Using it

- Demonstration of Manager-IS-Employee concept
  - A pointer to a derived class is also a pointer to the base class

```
// Pointer-to-derived IS Pointer-to-base
void processEmployee(Employee& emp) {
   cout << emp.name() << " : " << emp.salary() << endl ;
}

processEmployee(*emp) ;
processEmployee(*mgr) ; // OK Manager IS Employee
```

  - But the reverse is not true!

```
// Manager details are not visible through Employee* ptr
Employee* emp2 = mgr ; // OK Manager IS Employee
emp2->subs() ; // ERROR – Employee is not manager
```

OO Analysis and Design – ‘Is-A’ versus ‘Has-A’

- How is an ‘Is-A’ relationship different from a ‘Has-A’ relationship
  - An Is-A relationship expresses inheritance (A is B)
  - A Has-A relationship expresses composition (A is a component of B)

a `Calorimeter` **HAS-A** `Position`

```
class Calorimeter {
public:
  Position& p() { return _p ; }
private:
  Position _p ;
} ;
```

```
Calorimeter calo ;
// access position part
calo.p() ;
```

An `Manager` **IS-An** `Employee`

```
class Manager :
       public Employee {
public:

private:
} ;
```

```
Manager mgr ;
// Use employee aspect of mgr
mgr.salary() ;
```

Inheritance – constructors, initialization order

- Construction of derived class involves construction of base object ***and*** derived object
  - Derived class constructor must call base class constructor
  - The base class constructor is executed *before* the derived class ctor
  - Applies to all constructors, *including the copy constructor*

```
Manager::Manager(const char* _name, double _salary,
                            list<Employee*>& l) :
   Employee(_name,_salary),
   _subs(l) {
   cout << name() << endl ; // OK - Employee part of object
}                           // is fully constructed at this
                            // point so call to base class
                            // function is well defined

Manager::Manager(const Manager& other) :
   Employee(other), // OK Manager IS Employee
   _subs(other._subs) {
   // body of Manager copy constructor
}
```

Inheritance – Assignment

- If you define your own assignment operator for an inherited class (e.g. because you allocate memory) you need to handle the base class assignment as well
  - Virtual function call mechanism invokes call to derived class assignment operator only.
  - You should call the base class assignment operator in the derived class assignment operator

```
Manager* Manager::operator=(const Manager& other) {

  // Handle self assignment
   if (&other != this) return *this ;

   // Handle base class assignment
   Employee::operator=(other) ;

  // Derived class assignment happens here

  return *this ;
}
```

Inheritance – Destructors, call sequence

- For destructors the reverse sequences is followed
  - First the destructor of the derived class is executed
  - Then the destructor of the base class is executed
- Constructor/Destructor sequence example

```
class A {
  A() { cout << “A constructor” << endl ; }
  ~A() { cout << “A destructor” << endl ; }
} ;

class B : public A {
  B() { cout << “B constructor” << endl ; }
  ~B() { cout << “B destructor” << endl ; }
} ;

int main() {
  B b ;
  cout << endl ;
}
```

Output

```
A constructor
B constructor

B destructor
A destructor
```

Sharing information – protected access

- Inheritance preserves existing encapsulation
  - Private part of base class Employee is **not** accessible by derived class Manager

```
Manager::giveMyselfRaise() {
  _salary += 1000 ; // NOT ALLOWED: private in base class
}
```

- Sometimes useful if derived class can access part of private data of base class
  - Solution: 'protected' -- accessible by derived class, but not by public

```
class Base {
  public:
    int a ;
  protected:
    int b ;
  private:
    int c ;
} ;
```

```
class Derived : public Base {
 void foo() {
   a = 3 ; // OK public
   b = 3 ; // OK protected
  }
} ;

Base base ;
base.a = 3 ; // OK public
base.b = 3 ; // ERROR protected
```

Better example of protected interface

```
class Employee {
public:
  Employee(const char* name, double salary) ;
  annualRaise() { setSalary(_salary*1.03) ; }
  double salary() const { return _salary ; }

protected:
  void setSalary(double newSalary) {
    if (newSalary<_salary) {
      cout << "ERROR: salary must always increase" << endl ;
    } else {
      _salary = newSalary ;
    }
  }

private:
  string _name ;
  double _salary ;
} ;
```

The setSalary() function is protected:

Public cannot change salary except in controlled way through public annualRaise() method

Better example of protected interface

```
class Employee {
public:
  Employee(const char* name, double
  annualRaise() { setSalary(_salary
  double salary() const { return _s

protected:
  void setSalary(double newSalary) {
    if (newSalary<_salary) {
      cout << "ERROR: salary must alw      increase" << endl ;
    } else {
      _salary = newSalary ;
    }
  }

private:
  string _name ;
  double _salary ;
} ;
```

Managers can also get additional raise through `giveBonus()`

Access to protected `setSalary()` method allows `giveBonus()` to modify salary

```
class Manager : public Employee {
public:
   Manager(const char* name, double salary,
           list<Employee*> subs) ;

   giveBonus(double amount) {
      setSalary(salary()+amount) ;
   }
private:
  list<Employee*> _subs ;
} ;
```

Better example of protected interface

```
class Employee {
public:
  Employee(const char* name, double salary) ;
  annualRaise() { setSalary(_salary*1.03) ; }
  double salary() const { return _salary ; }

protected:
  void setSalary(double newSalary) {
    if (newSalary<_salary) {
      cout << “ERROR: salary must always increase” << endl ;
    } else {
      _salary = newSalary ;
    }
  }
```

```
class Manager : public Employee {
public:
   Manager(const char* name, double salary,
           list<Employee*> subs) ;

   giveBonus(double amount) {
      setSalary(salary()+amount) ;
   }
private:
  list<Employee*> _subs ;
} ;
```

Note how accessor/modifier pattern `salary()/setSalary()` is also useful for protected access

Manager is only allowed to change salary through controlled method: negative bonuses are not allowed…

Object Oriented Analysis & Design with Inheritance

- Principal OOAD rule for inheritance: an Is-A relation is an **extension** of an object, **not a restriction**
  - `manager` Is-An `employee` is good example of a valid Is-A relation:

> A manager conceptually is an employee *in all respects*, but with some extra capabilities

  - *Many cases are not that simple however*

- Some other cases to consider
  - A `cat` is a `carnivore` that knows how to meow (maybe)
  - A `square` is a `rectangle` with equal sides (**no!**)
    - *'Is-A except' is a restriction, not an extension*
  - A `rectangle` is a `square` with method to change side lengths (**no!**)
    - *Code in square can make legitimate assumptions that both sides are of equal length*

Object Oriented Analysis & Design with Inheritance

- Remarkably easy to get confused
  - Particularly if somebody else inherits from your class later (and you might not even know about that)

- The Iron-Clad rule: The **L**iskov **S**ubtitution **P**rinciple
  - Original version:

'If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S a subtype of T'

  - In plain English:

'An object of a subclass must behave indistinguishably from an object of the superclass when referenced as an object of the superclass'

  - Keep this in mind when you design class hierarchies using Is-A relationships

Object Oriented Analysis & Design with Inheritance

- Extension through inheritance can be quite difficult
  - 'Family trees' seen in text books very hard to do in real designs

- Inheritance for "extension" is non-intuitive, but for "restriction" is wrong

- Inheritance is hard to get right in advance
  - Few things are straightforward extensions
  - Often behavior needs to be overridden rather than extended
  - Design should consider entire hierarchy

- But do not despair:
  - Polymorphism offers several new features that will make OO design with inheritance easier

Advanced features of inheritance

- Multiple inheritance is also allowed
  - A class with multiple base classes

```
class Manager : public Employee, public ShareHolder {
   …
} ;
```

  - Useful in certain circumstances, but things become complicated very quickly

- Private, protected inheritance
  - Derived class does not inherit public interface of base class
  - Example declaration

```
class Stack : private List {
  …
} ;
```

  - Private inheritance does not describe a 'Is-A' relationship but rather a 'Implemented-by-means-of' relationship
  - Rarely useful
  - Rule of thumb: Code reuse through inheritance is a bad idea

Polymorphism

- Polymorphism is the ability of an object to retain its true identity even when accessed through a base pointer
  - This is perhaps easiest understood by looking at an example *without* polymorphism
- Example without polymorphism
  - Goal: have `name()` append `“(Manager)”` to name tag for manager
  - Solution: implement `Manager::name()` to do exactly that

```
class Manager : public Employee {
public:
  Manager(const char* name, double salary,
          vector<Employee*> subordinates) ;

  const char* name() const {
    cout << _name << “ (Manager)” << endl ;
  }

  list<Employee*> subs() const ;
private:
  list<Employee*> _subs ;
} ;
```

Example without polymorphism

- Using the improved manager class

```
Employee emp("Wouter",10000) ;
Manager mgr("Stan",20000,&emp) ;

cout << emp.name() << endl ; // Prints "Wouter"
cout << mgr.name() << endl ; // Prints "Stan (manager)"
```

- But it doesn't work in all circumstances…

```
void print(Employee& emp) {
   cout << emp.name() << endl ;
}
print(emp) ;  // Prints "Wouter"
print(mgr) ;  // Prints "Stan" – NOT WHAT WE WANTED!
```

  - **Why does this happen?**
  - Function `print()` sees `mgr` as employee, thus the compiler calls `Employee::name()` rather than `Manager::name()` ;
  - Problem profound: `name()` function call selected at compile time. No way for compiler to know that `emp` really is a Manager!

Polymorphism

- Polymorphism is the ability of an object to retain its true identity even when accessed through a base pointer
  - I.e. we want this:

```
Employee emp("Wouter",10000) ;
Manager mgr("Stan",20000,&emp) ;

void print(Employee& emp) {
   cout << emp.name() << endl ;
}
print(emp) ;  // Prints "Wouter"
print(mgr) ;  // Prints "Stan (Manager)"
```

- In other words: Polymorphism is the ability to treat objects of different types the same way
  - To accomplish that we will need to tell C++ compiler to look at ***run-time*** what `emp` really points to.
  - In compiler terminology this is called '***dynamic binding***' and involves the compiler doing some extra work prior to executing the `emp->name()` call

Dynamic binding in C++ – keyword virtual

- The keyword ***virtual*** in a function declaration activates dynamic binding for that function
  - The example class Employee revisited

```
class Employee {
public:
  Employee(const char* name, double salary) ;
  virtual const char* name() const ;
  double salary() const ;
private:
  …
} ;
```

  - No further changes to class Manager needed

… And the broken printing example now works

```
void print(Employee& emp) {
   cout << emp.name() << endl ;
}
print(emp) ;  // Prints "Wouter"
print(mgr) ;  // Prints "Stan (Manager)" EUREKA
```

Keyword virtual – some more details

- Declaration 'virtual' needs only to be done in the base class
  - Repetition in derived classes is OK but not necessary

- Any member function can be virtual
  - Specified on a **member-by-member** basis

```
class Employee {
public:
  Employee(const char* name, double salary) ;
  ~Employee() ;

  virtual const char* name() const ; // VIRTUAL
  double salary() const ;            // NON-VIRTUAL

private:
  …
} ;
```

Virtual functions and overloading

- For overloaded virtual functions either all or none of the functions variants should be redefined

OK – all redefined

```
class A {
 virtual void func(int) ;
 virtual void func(float) ;
} ;

class B : public A {
 void func(int) ;
 void func(float) ;
} ;
```

OK – none redefined

```
class A {
 virtual void func(int) ;
 virtual void func(float) ;
} ;

class B : public A {
} ;
```

NOT OK – partially redefined

```
class A {
 virtual void func(int) ;
 virtual void func(float) ;
} ;

class B : public A {
  void func(float) ;
} ;
```

Virtual functions – Watch the destructor

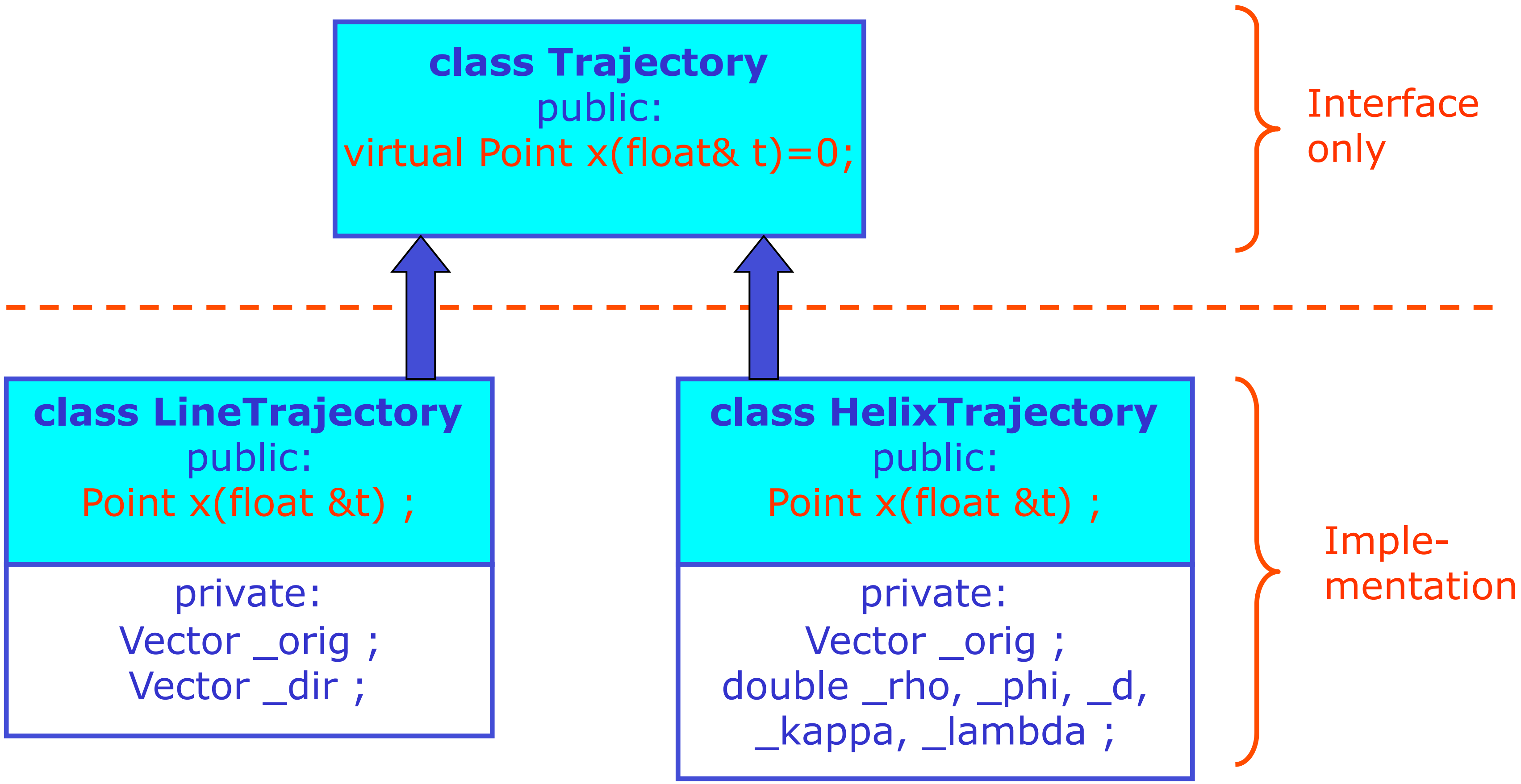
- Watch the destructor declaration if you define virtual functions
  - Example

```
Employee* emp = new Employee("Wouter",10000) ;
Manager* mgr = new Manager("Stan",20000,&emp) ;

void killTheEmployee(Employee* emp) {
   delete emp ;
}

killTheEmployee(emp) ; // OK
killTheEmployee(mgr) ; // LEGAL but WRONG!
                       // calls ~Employee() only, not ~Manager()
```

  - Any resources allocated in Manager constructor will not be released as Manager destructor is not called (just Employee destructor)
  - Solution: make the destructor virtual as well
- Lesson: if you ever delete a derived class through a base pointer your class should have a virtual destructor
  - In practice: Whenever you have any virtual function, make the destructor virtual

Abstract base classes – concept

- Virtual functions offer an important tool to OOAD – the Abstract Base Class
  - An Abstract Base Class is an interface only. It describes how an object can be used but does not offer a (full) implementation



Abstract base classes – pure virtual functions

- A class becomes an abstract base class when it has one or more pure virtual functions
  - A pure virtual function is a declaration without an implementation
  - Example

```
class Trajectory {
public:
   Trajectory() ;
   virtual ~Trajectory() ;
   virtual Point x(float& t) const = 0 ;
} ;
```

  - It is **not possible** to create an **instance** of an **abstract base class**, only of implementations of it

```
Trajectory* t1 = new Trajectory(…) ;    // ERROR abstract class
Trajectory* t2 = new LineTrajectory(…); // OK
Trajectory* t3 = new HelixTrajectory(…);// OK
```

Abstract base classes and design

- Abstract base classes are a way to express common properties and behavior without implementation
  - Especially useful if there are multiple implementations of a common interface possible
  - Example: a straight line '**is a**' trajectory,
    but a helix also '**is a**' trajectory

- Enables you to write code at a higher level abstraction
  - For example, you don't need to know how trajectory is parameterized, just how to get its position at a give flight time.
  - Powered by polymorphism

- Simplifies extended/augmenting existing code
  - Example: can write new class `SegmentedTrajectory`. Existing code dealing with trajectories can use new class without modifications (or even recompilation!)

Abstract Base classes – Example

- Example on how to use abstract base classes

```
void processTrack(Trajectory& track) ;

int main() {
  // Allocate array of trajectory pointers
  Trajectory* tracks[3] ;

  // Fill array of trajectory pointers
  tracks[0] = new LineTrajectory(…) ;
  tracks[1] = new HelixTrajectory(…) ;
  tracks[2] = new HelixTrajectory(…) ;

  for (int i=0 ; i<3 ; i++) {
    processTrack(*tracks[i]) ;
  }
}

void processTrack(Trajectory& track) {
   cout << "position at flight length 0 is "
        << track.pos(0) << endl ;
}
```

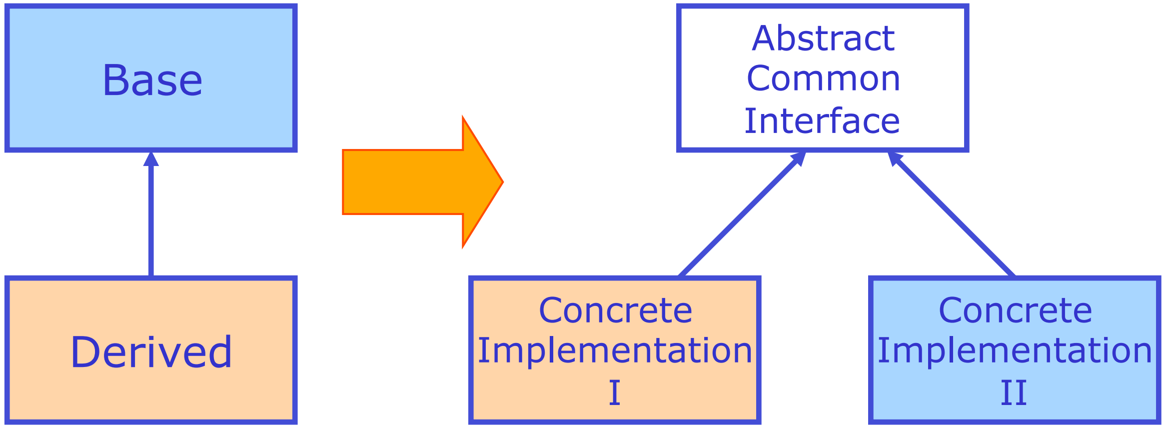
Use Trajectory interface to manipulate track without knowing the exact class you're dealing with (HelixTrajectory or LineTrajectory)

The power of abstract base classes

- You can even reuse existing *compiled* code with new implementations of abstract base classes
- Example of reusing compiled code with a new class
  - First iteration – no magnetic field
    1. Write abstract class `Trajectory`
    2. Write implementation `LineTrajectory`
    3. Write algorithm class `TrackPointPOCA` to find closest point of approach between given cluster position and trajectory using `Trajectory` interface
  - Second iteration – extend functionality to curved tracks in magnetic field
    1. Write implementation `HelixTrajectory`, compile `HelixTrajectory` code
    2. Link `HelixTrajectory` code with existing compiled code into new executable
    3. Your executable can use the newly defined `HelixTrajectory` objects without further modification
- Higher level code `TrackPointPOCA` transparent to future code changes!

Object Oriented Analysis and Design and Polymorphism

- Design of class hierarchies can be much simplified if only abstract base classes are used
  - In plain inheritance derived class forcibly inherits full specifications of base type
  - Two classes that inherit from a common abstract base class can share any subset of their common functionality



Polymorphic objects and storage

- Polymorphic inheritance simplifies many aspects of object use and design – but there are still some areas where you still need to pay attention:

- Storage of polymorphic object collections
  - Reason: when you start allocating memory the true identity of the object matters. You need to know exactly how large it is after all…
  - Storage constructions that assume uniform size of objects also no longer work – Use of arrays, STL container classes not possible

- Cloning of polymorphic object collections
  - Reason: you want to clone the implementation class not the interface class so you must know the true type
  - Ordinarily virtual functions solves such problems, however there is no such thing as a virtual copy constructor…

- Will look into this in a bit more detail in the next slides…

Collections of Polymorphic objects – storage

- Dealing with storage
  - Naïve attempt to make STL list of trajectories

```
LineTrajectory track1(…) ;
HelixTrajectory track2(…) ;

list<Trajectory> trackList ; // ERROR
```

  - Why Error: list<X> calls default constructor for X, but can not instantiate X if X is an abstract classes such as Trajectory
  - Solution: make a ***collection of pointers***

```
Trajectory* track1 = new LineTrajectory(…) ;
Trajectory* track2 = new HelixTrajectory(…) ;

list<Trajectory*> trackList ; // OK
trackList.push_back(&track1) ;
trackList.push_back(&track2) ;
```

Collections of Polymorphic objects – storage

- But remember ownership semantics
  - STL container will delete pointers to objects, but not objects themselves
  - In other words: **deleting `trackList` does NOT delete the tracks!**

- Technical Solution
  - Write a new container class, or inherit it from a STL container class that takes ownership of objects pointed to.
  - NB: This is not so easy – think about what happens if replace element in container: does removed element automatically get deleted on the spot?

- Bookkeeping Solution
  - Document clearly in function that creates `trackList` that contents of `trackList` is owned by caller in addition to list itself
  - More prone to mistakes

Collections of polymorphic objects – copying

- Copying a polymorphic collection also has its issues

```
list<Trajectory*> trackList ;
list<Trajectory*> clonedTrackList ;

list<Trajectory*>::iterator iter ;
for(iter=trackList.begin() ; iter!=trackList.end() ; ++iter) {
  Trajectory* track = *iter ;

  Trajectory* newTrack = new Trajectory(*track);
                                     // NOPE – attempt to
                                     // instantiate abstract class
  cloneTrackList.push_back(newTrack) ;
}
```

- Solution: make your own 'virtual copy constructor'
  - Add a pure virtual `clone()` function to your abstract base class

```
class Trajectory {
public:
   Trajectory() ;
   virtual ~Trajectory() ;
   virtual Trajectory* clone() const = 0 ;
   virtual Point x(float& t) const = 0 ;
} ;
```

The virtual copy constructor

- Implementing the `clone()` function

```
class LineTrajectory : public Trajectory {
   LineTrajectory(…) ;
   LineTrajectory(const LineTrajectory& other) ;
   virtual ~LineTrajectory() ;

   // 'virtual copy constructor'
   virtual Trajectory* clone() const {
       return new LineTrajectory(*this) ; calls copy ctor
   }
} ;
```

- Revisiting the collection copy example

```
list<Trajectory*>::iterator iter ;
for(iter=tl.begin() ; iter!=tl.end() ; ++iter) {
  Trajectory* track = *iter ;
  Trajectory* newTrack = track->clone() ;
  clonedTrackList.push_back(newTrack) ;
}
```

  - `clone()` returns a `Trajectory*` pointer to a `LineTrajectory` for track1
  - `clone()` returns a `Trajectory*` pointer to a `HelixTrajectory` for track2

Run-time type identification

- Sometimes you need to cheat…
  - Example: The preceding example of cloning a list of tracks
  - Proper solution: add `virtual clone()` function
  - But what if (for whatever reason) we cannot touch the base class?
    - For example: it is designed by somebody else that doesn't want you to change it, or it is part of a commercial library for which you don't have the source code
  - Can you still tell what the true type is given a base class pointer?

- Solution: the `dynamic_cast<>` operator
  - Returns valid pointer if you guessed right, null otherwise

```
Trajectory* track ;
LineTrajectory* lineTrack =
dynamic_cast<LineTrajectory*>(track) ;

if (lineTrack != 0) {
  cout << "track was a LineTrajectory" << endl ;
} else {
  cout << "track was something else" << endl ;
}
```

Run time type identification

- Solution to `trackList` clone problem

```
list<Trajectory*>::iterator iter ;
for(iter=tl.begin() ; iter!=tl.end() ; ++iter) {
  Trajectory* track = *iter ;

  LineTrajectory* line = dynamic_cast<LineTrajectory*> track ;
  if (line) {
    newTrack = new LineTrajectory(*line) ;
    continue ;
  }

 HelixTrajectory* helix = dynamic_cast<HelixTrajectory*> track;
  if (helix) {
    newTrack = new HelixTrajectory(*helix) ;
    continue ;
  }

 cout << “ERROR: track is neither helix nor line” << endl ;
}
```

  – *Obviously ugly, maintenance prone, incomplete*
  – **Use `dynamic_cast<>` as last resort only!**

C++ competes with your government

- Flip side of polymorphic inheritance – performance

- Inheritance can be taxed!
  - In C++ you incur a performance overhead if you use virtual functions instead of regular (statically bound) functions
  - Reason: every time you call a virtual function the C++ compiler inserts code that identifies the true identity of the object and decided based on that information what function to call
  - Overhead *only* applies to virtual functions. Regular function in a class with other virtual functions do not incur this overhead

- Use virtual functions judiciously
  - Don't make every function in your class virtual
  - Overhead is not always waste of time. If alternative is figuring out the true identity of the object yourself, the lookup step is intrinsic to your algorithms.