Chapter 15: Inheritance

Instructor: Mark Edmonds

edmonds_mark@smc.edu

Inheritance

- Inheritance allows us to create classes based on other classes
- For instance, think of all animals. We could define a hierarchy of all types of animals. Let's just consider mammals.
    - Each mammal has some common functionality, and as we get more specific in the hierarchy, the functionality becomes more and more specific.
    - For example, all mammals breathe oxygen through air. But whales have fins and rats, monkeys, and humans have limbs.
    - Every specific type of mammal (whale, rats, monkeys, humans) *inherit* the properties and functionality of a mammal.

Base class

- The base class is the starting point for defining a set of classes.
- The most general attributes and methods are defined here
- For example, mammal could be a base class

Derived class

- The derived class extends the base class in some way.
- For example, a whale extends the mammal base class. A monkey could also extend the mammal base class.

Mammal hierarchy

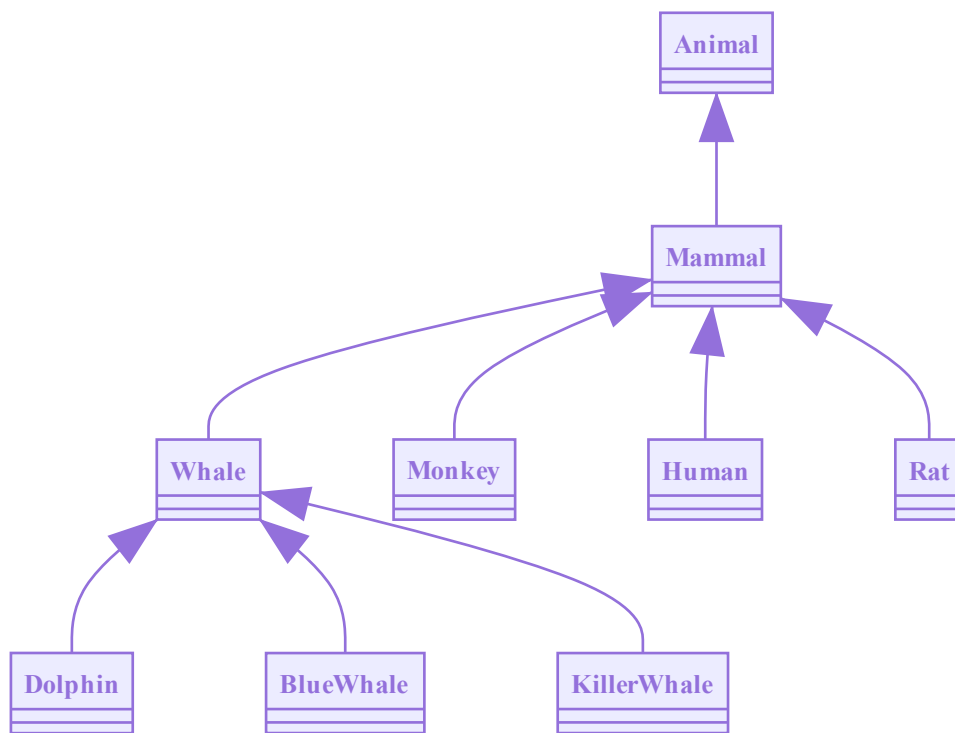For instance, we could have the following hierarchy:

Figure 1: Partial hierarchy of mammals

- Here, Animal is the base class of the derived class Mammal.
  - Mammal is the base class of the derived classes Whale, Rat, Monkey, and Human.
  - So a derived class can serve as the base class for another derived class (similar to how a callee becomes a caller if the callee invokes another function).

Door Example

- Let's design a set of door classes for an adventure game
- What are all the common characteristics of doors?

Door object

- A door (as an object), it should know:
  - status (open or shut)
- A door (as an objecT), can do:
  - Initialize itself as shut

  – Open itself, if possible
  – Close itself
  – Tell whether or not it is open

Door class

- Here's a generic base class:

```
1  class Door {
2  public:
3    Door();
4    bool isOpen() const;
5    void open();
6    void close();
7  protected:
8    bool isShut;
9  }
```

protected qualifier

- The **protected** qualifier is a compromise between **private** and **public**
  - **protected** is **public** to the base class
  - **protected** is **public** to friends of the base class
  - **protected** is **public** to the derived classes
  - **protected** is **public** to friends of the derived classes
  - **protected** is **private** to other classes
- This allows base classes to:
  1. Hide functionality to users of the class (who can only access **public** members)
  2. Expose members to only to derived classes (who can access **public** and **protected** members)
  3. Hide members from both users of the class and derived classes (through **private** members of the base class)
     - Derived classes cannot access **private** members of the base class!

Person, Student, Teacher Example

- In this example, we'll look at an example of a Student and Teacher class that are both derived from a Person base class (after all, students and teachers are both typically people in the real world!)
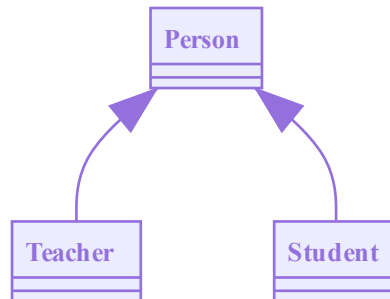
- We'll use the following hierarchy:

Figure 2: Person, Student, Teacher Hierarchy

Example Person.h

- Our `Person` class will serve as our base class

```
1  #ifndef PERSON_H
2  #define PERSON_H
3
4  #include <iostream>
5  #include <string>
6
7  namespace cs52 {
8
9  class Person {
10 public:
11     Person();
12     Person( std::string name, std::string address );
13
14     std::string getName();
15     std::string getAddress();
16     void setName( std::string name );
17     void setAddress( std::string address );
18
19     friend std::ostream& operator<<( std::ostream& outs,
20                                      const Person& p );
21     ~Person();
22 private:
23
```

```
24
25  protected:
26      std::string my_name;
27      std::string my_address;
28      int* integer;
29
30  };
31
32  }
33  #endif
```

Example Person.cpp

```cpp
1  #include "Person.h"
2
3  using namespace cs52;
4
5  namespace cs52 {
6
7  Person::Person() {
8
9  }
10
11  // note the use of an initializer list here
12  Person::Person( std::string name, std::string address ) : my_name( name
       ),
13                                                             my_address(
                                                                 address )
                                                             {
14
15  }
16
17  std::string Person::getName() {
18      return( my_name );
19  }
20
21  std::string Person::getAddress() {
22      return( my_address );
23  }
24
25  void Person::setName( std::string name ) {
26      my_name = name;
```

```
27  }
28
29  void Person::setAddress( std::string address ) {
30      my_address = address;
31  }
32
33  std::ostream& operator<<( std::ostream& outs,
34                            const Person& p ) {
35      outs << "name=";
36      outs << p.my_name;
37      outs << " address=";
38      outs << p.my_address;
39      return( outs );
40  }
41
42  }
```

Example Student.h

- The `Student` class inherits from the `Person`

```
1   #ifndef STUDENT_H
2   #define STUDENT_H
3
4   #include <iostream>
5   #include <string>
6
7   #include "Person.h"
8
9   namespace cs52 {
10
11  class Student : public Person {
12  public:
13      Student();
14      Student( std::string name, std::string address,
15              std::string id,   std::string gpa );
16
17      std::string getGPA();
18      void setGPA( std::string gpa );
19      std::string getID();
20      void setID( std::string id );
21
```

```
22
23      friend std::ostream& operator<<( std::ostream& outs,
24                                        const Student& s );
25 private:
26
27
28 protected:
29      std::string my_ID;
30      std::string my_GPA;
31 };
32
33 }
34 #endif
```

Example Student.cpp

- Take note: we must use an initializer list to construct the Person object: : Person(name, address) in the constructor for Student

```cpp
1  #include "Student.h"
2
3  using namespace cs52;
4
5  namespace cs52 {
6
7  Student::Student() {
8
9  }
10
11 // note the use of an initializer list here
12 Student::Student( std::string name, std::string address,
13                   std::string id,   std::string gpa ) : Person( name,
                       address ),
14                                       my_ID( id ), my_GPA( gpa ) {
15
16 }
17
18 std::string Student::getGPA() {
19     return( my_GPA );
20 }
21
22 void Student::setGPA( std::string gpa ) {
```

```
23        my_GPA = gpa;
24  }
25
26  std::string Student::getID() {
27      return( my_ID );
28  }
29
30  void Student::setID( std::string id ) {
31      my_ID = id;
32  }
33
34  std::ostream& operator<<( std::ostream& outs,
35                            const Student& s ) {
36      Person p = s;
37      outs << p;
38      outs << " id=" << s.my_ID << " gpa=" << s.my_GPA;
39      return( outs );
40  }
41
42  }
```

Example Teacher.h

- The Teacher class also inherits from the Person

```
1   #ifndef TEACHER_H
2   #define TEACHER_H
3
4   #include <iostream>
5   #include <string>
6
7   #include "Person.h"
8
9   namespace cs52 {
10
11  class Teacher : public Person {
12  public:
13      Teacher();
14      Teacher( std::string name, std::string address,
15              std::string dept );
16
17      std::string getDepartment();
```

```
18      void setDepartment( std::string dept );
19
20      friend std::ostream& operator<<( std::ostream& outs,
21                                        const Teacher& t );
22  private:
23
24
25  protected:
26      std::string my_department;
27
28  };
29
30  }
31  #endif
```

Example Teacher.cpp

- Take note: we must use an initializer list to construct the `Person` object: : `Person( name, address )` in the constructor for `Teacher`

```
1   #include "Teacher.h"
2
3   using namespace cs52;
4
5   namespace cs52 {
6
7   Teacher::Teacher() {
8
9   }
10
11  // note the use of an initializer list here
12  Teacher::Teacher( std::string name, std::string address,
13                  std::string dept ) : Person( name, address ),
14                                          my_department( dept ) {
15  }
16
17  std::string Teacher::getDepartment() {
18      return( my_department );
19  }
20
21  void Teacher::setDepartment( std::string dept ) {
22      my_department = dept;
```

```
23  }
24
25  std::ostream& operator<<( std::ostream& outs,
26                            const Teacher& t ) {
27      Person p = t;
28      outs << p;
29      outs << " department=" << t.my_department;
30      return( outs );
31  }
32
33  }
```

Example Main.cpp

```
1   #include <iostream>
2
3   #include "Person.h"
4   #include "Teacher.h"
5   #include "Student.h"
6
7
8   int main() {
9       using namespace std;
10      using namespace cs52;
11
12      Person p( "Howie", "Los Angeles" );
13      Teacher t( "HowieTeacher", "Santa Monica", "Business" );
14      Student s( "Howie", "Los Angeles", "102", "3.5" );
15
16      cout << p << endl;
17      cout << t << endl;
18      cout << s << endl;
19
20      return( 0 );
21  }
```

Inheritance Syntax

- To inherit from another class, we use the following syntax when defining the class:

```
1   class DerivedClass : public BaseClass
```

```
2  {
3    // class definition
4  };
```

- The "syntactic sugar" we added here is the : **public** BaseClass. This is informing the compiler that DerivedClass is inheriting from BaseClass with an "access mode" of public.

Access mode

- The access mode determines how users of the class can interact with the BaseClass when instantiating a DerivedClass
- For this, let's consider the following implementations of BaseClass and DerivedClass

BaseClass

- BaseClass is just a dummy class in this example - we'll get to real world examples in a moment

```
1  class BaseClass
2  {
3  public:
4    int x;
5    BaseClass(int x_, int y_, int z_) : x(x_), y(y_), z(z_) {}
6  protected:
7    int y;
8  private:
9    int z;
10 }
```

DerivedClass

The access modes are:

- **public**: public members of the base class will become public members of the derived class and protected members of the base class will become protected in the derived class
    - In our example above, BaseClass's x member will be treated as **public** in DerivedClass, y will be treated as **protected** in DerivedClass

```
1  class DerivedClass : public BaseClass
2  {
3    // x is public
4    // y is protected
```

```
5    // z is not accessible from DerivedClass
6  public:
7    DerivedClass();
8  }
```

- **protected**: Both the public and protected members of BaseClass become protected in DerivedClass
  - If we changed our declaration of DerivedClass to be: **class** DerivedClass : **protected** BaseClass in our example above, BaseClass's x and y will be treated as **protected** members of DerivedClass

```
1  class DerivedClass : protected BaseClass
2  {
3    // x is protected
4    // y is protected
5    // z is not accessible from DerivedClass
6  public:
7    DerivedClass();
8  }
```

- **private**: Both the public and protected members of the BaseClass become private in DerivedClass
  - If we changed our declaration of DerivedClass to be: **class** DerivedClass : **private** BaseClass in our example above, BaseClass's x and y will be treated as **private** members of DerivedClass

```
1  class DerivedClass : private BaseClass
2  {
3    // x is private
4    // y is private
5    // z is not accessible from DerivedClass
6  public:
7    DerivedClass();
8  }
```

Initializing BaseClasses

- We must always initialize the base class using an initializer list.
  - This is because the base class must be constructed *before* the derived class. By using an initializer list, we construct the BaseClass before the DerivedClass is completely constructed

- For instance, we could write:

```
1  DerivedClass::DerivedClass() : BaseClass(1,2,3) {
2    // rest of constructor goes here
3    // we cannot initialize/assign BaseClass here. It must be initialized
         in the initializer list
4  }
```

Door Example - Lockable Door

- Let's look at another door example. This time, we'll make a new LockableDoor class that derives from the base Door class

```
1  class LockableDoor : public Door {
2  public:
3    LockableDoor();
4    bool isLocked() const;
5    void open();
6    void lock();  void unlock();
7  protected:
8    bool thelock;
9  }
```

- Notice, we did NOT need to define the functions defined in Door. That's the point of inheritance; we get the functionality of Door inside of LockableDoor and only need to add the **new** functionality we want the LockableDoor to have
  - This is very useful when you start writing larger pieces of software
- The LockableDoor object has the following member attributes (data) and member functions:

Member attributes	Member methods
isShut	isLocked()
theLock	open()
	lock()
	unlock()
	isOpen()
	close()

- Notice that `LockableDoor` did not define `isShut`, `open()`, `isOpen()`, or `close()`. These are all part of the base class `Door`, but they are available in `LockableDoor`

Inheritance behavior

- By default, all member methods and member data are inherited down to derived classes
  - This happens without mentioning these methods and attributes in the derived class definition
- Any member method or member attribute can be redefined in the derived class
  - This hides access to the base class versions - the base class still retains its copies of redefined member attributes and member methods
- For instance, we may wish to redefine the `open()` functionality of `DerivedClass` to take into account whether or not the door is locked:

```
1  void LockableDoor::open( )
2  {
3    if (!isLocked()) {
4      Door::open();
5    }
6  }
```

- We can write this method *without* defining it again in `LockableDoor`'s class definition.
- We can use the scope operator `::` to specify which version of a function to call (note: we had to use `Door::open()` above. Had we just written `open()`, we would have started a recursive infinite loop!)

Using the door examples

- We can now use our doors for different circumstances

```
1  Door hallDoor;
2  LockableDoor frontDoor;
3
4  hallDoor.open();
5  frontDoor.lock();
6  frontDoor.open();
7  if (!frontDoor.isOpen())
8      frontDoor.unlock();
```
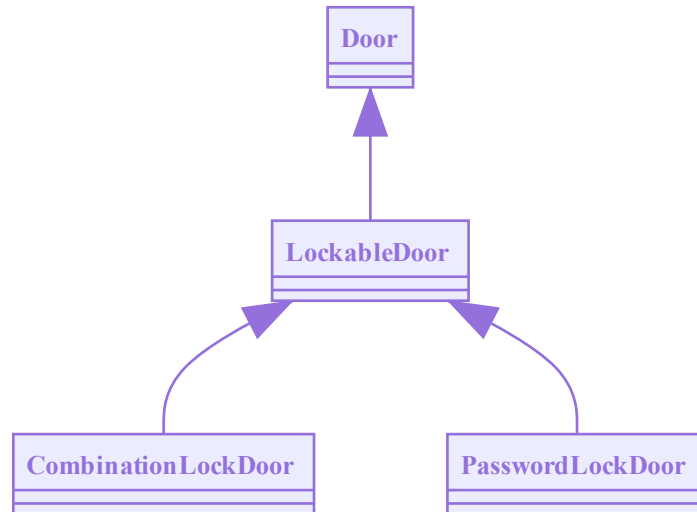
Derived classes can become base classes

Figure 3: Partial hierarchy of mammals

CombinationLockDoor

```
1  class CombinationLockDoor : public LockableDoor {
2  public:
3    CombinationLockDoor( int combo = 0);
4    void unlock( int combo );
5  protected:
6    int thecombination;
7  }
```

- The CombinationLockDoor object has the following member attributes (data) and member functions:

Member attributes	Member methods
isShut	isLocked()
theLock	open()
thecombination	lock()
	unlock(int)
	isOpen()

Member attributes	Member methods
	close()

- You can see how easily we can extend functionality to different cases!

Person, Student, Teacher Example - with Pointers

- This example will show how we can use pointers with inheritance, how **protected** members are accessible to derived classes, and how redefined members hide access to the parent class versions

Example PersonPtr.h

- Our `Person` class will serve as our base class

```cpp
#ifndef PERSON_H
#define PERSON_H

#include <iostream>
#include <string>

namespace cs52 {

class Person {
public:
    Person();
    Person( std::string name, std::string address );

    std::string getName();
    std::string getAddress();
    void setName( std::string name );
    void setAddress( std::string address );

    friend std::ostream& operator<<( std::ostream& outs,
                                     const Person& p );
    friend std::ostream& operator<<( std::ostream& outs,
                                     const Person* p );

private:

```

```
26
27  protected:
28      std::string my_name;
29      std::string my_address;
30
31  };
32
33  }
34  #endif
```

Example PersonPtr.cpp

```
 1  #include "Person.h"
 2
 3  using namespace cs52;
 4
 5  namespace cs52 {
 6
 7  Person::Person() {
 8
 9  }
10
11  // note the use of an initializer list here
12  Person::Person( std::string name, std::string address ) : my_name( name
        ),
13                                                          my_address(
                                                              address )
                                                          {
14
15  }
16
17  std::string Person::getName() {
18      return( my_name );
19  }
20
21  std::string Person::getAddress() {
22      return( my_address );
23  }
24
25  void Person::setName( std::string name ) {
26      my_name = name;
27  }
```

```
28
29   void Person::setAddress( std::string address ) {
30       my_address = address;
31   }
32
33   std::ostream& operator<<( std::ostream& outs,
34                             const Person& p ) {
35       outs << "name=";
36       outs << p.my_name;
37       outs << " address=";
38       outs << p.my_address;
39       return( outs );
40   }
41
42   std::ostream& operator<<( std::ostream& outs,
43                             const Person* p ) {
44       if (p == NULL) {
45           outs << "NULL pointer";
46       }
47       else {
48           outs << "name=";
49           outs << p->my_name;
50           outs << " address=";
51           outs << p->my_address;
52       }
53       return( outs );
54   }
55
56   }
```

Example StudentPtr.h

- The Student class inherits from the Person

```
1   #ifndef STUDENT_H
2   #define STUDENT_H
3
4   #include <iostream>
5   #include <string>
6
7   #include "Person.h"
8
```

```
 9  namespace cs52 {
10
11  class Student : public Person {
12  public:
13      Student();
14      Student( std::string name, std::string address,
15              std::string id,   std::string gpa );
16
17      std::string getGPA();
18      void setGPA( std::string gpa );
19      std::string getID();
20      void setID( std::string id );
21
22
23      friend std::ostream& operator<<( std::ostream& outs,
24                                       const Student& s );
25      friend std::ostream& operator<<( std::ostream& outs,
26                                       const Student* s );
27
28  private:
29
30
31  protected:
32      std::string my_ID;
33      std::string my_GPA;
34  };
35
36  }
37  #endif
```

Example StudentPtr.cpp

- Take note: we must use an initializer list to construct the Person object: : Person(name, address) in the constructor for Student

```
1  #include "Student.h"
2
3  using namespace cs52;
4
5  namespace cs52 {
6
7  Student::Student() {
```

```
 8
 9   }
10
11   // note the use of an initializer list here
12   Student::Student( std::string name, std::string address,
13                     std::string id,   std::string gpa ) : Person( name,
                            address ),
14                                          my_ID( id ), my_GPA( gpa ) {
15   }
16
17   std::string Student::getGPA() {
18       return( my_GPA );
19   }
20
21   void Student::setGPA( std::string gpa ) {
22       my_GPA = gpa;
23   }
24
25   std::string Student::getID() {
26       return( my_ID );
27   }
28
29   void Student::setID( std::string id ) {
30       my_ID = id;
31   }
32
33   std::ostream& operator<<( std::ostream& outs,
34                             const Student& s ) {
35       Person p = s;
36       outs << p;
37       outs << " id=" << s.my_ID << " gpa=" << s.my_GPA;
38       return( outs );
39   }
40
41   std::ostream& operator<<( std::ostream& outs,
42                             const Student* s ) {
43       if (s == NULL) {
44           outs << "NULL pointer";
45       }
46       else {
47           const Person* p = s;
48           outs << p;
49           outs << " id=" << s->my_ID << " gpa=" << s->my_GPA;
```

```
50        }
51        return( outs );
52  }
53
54  }
```

Example TeacherPtr.h

- The Teacher class also inherits from the Person

```
1   #ifndef TEACHER_H
2   #define TEACHER_H
3
4   #include <iostream>
5   #include <string>
6
7   #include "Person.h"
8
9   namespace cs52 {
10
11  class Teacher : public Person {
12  public:
13      Teacher();
14      Teacher( std::string name, std::string address,
15              std::string dept );
16
17      std::string getDepartment();
18      void setDepartment( std::string dept );
19
20      friend std::ostream& operator<<( std::ostream& outs,
21                                       const Teacher& t );
22      friend std::ostream& operator<<( std::ostream& outs,
23                                       const Teacher* t );
24
25  private:
26
27
28  protected:
29      std::string my_department;
30
31  };
32
```

```
33  }
34  #endif
```

Example TeacherPtr.cpp

- Take note: we must use an initializer list to construct the Person object: : Person(name, address) in the constructor for Teacher

```
 1  #include "Teacher.h"
 2
 3  using namespace cs52;
 4
 5  namespace cs52 {
 6
 7  Teacher::Teacher() {
 8
 9  }
10
11  // note the use of an initializer list here
12  Teacher::Teacher( std::string name, std::string address,
13                    std::string dept ) : Person( name, address ),
14                                            my_department( dept ) {
15  }
16
17  std::string Teacher::getDepartment() {
18      return( my_department );
19  }
20
21  void Teacher::setDepartment( std::string dept ) {
22      my_department = dept;
23  }
24
25  std::ostream& operator<<( std::ostream& outs,
26                            const Teacher& t ) {
27      Person p = t;
28      outs << p;
29      outs << " department=" << t.my_department;
30      return( outs );
31  }
32
33  std::ostream& operator<<( std::ostream& outs,
34                            const Teacher* t ) {
```

```
35      if (t == NULL) {
36          outs << "NULL pointer";
37      }
38      else {
39          const Person* p = t;
40          outs << p;
41          outs << " department=" << t->my_department;
42      }
43      return( outs );
44  }
45
46  }
```

Example MainPtr.cpp

```
1   #include <iostream>
2
3   #include "Person.h"
4   #include "Teacher.h"
5   #include "Student.h"
6
7
8   int main() {
9       using namespace std;
10      using namespace cs52;
11
12      Person* p = new Person( "Howie", "Los Angeles" );
13      Teacher* t = new Teacher( "HowieTeacher", "Santa Monica", "Business
            " );
14      Student* s = new Student( "Howie", "Los Angeles", "102", "3.5" );
15
16      cout << p << endl;
17      cout << t << endl;
18      cout << s << endl;
19
20      delete( p );
21      delete( t );
22      delete( s );
23
24      return( 0 );
25  }
```

Relationships between Objects

- **IS-A**
    - One class "is a kind of" another class
    - Base class is a general class
    - Derived class is a specialization of the general concept
- **PART-OF**
    - One class "is a part of" another class
    - often used to represent compound objects

{.mermaid format=svg caption="Difference between IS-A and PART-OF relationship "}} graph BT Mac -->|IS A| Computer Monitor -->|PART OF| Computer

- Here is an example of the difference between IS-A and PART-OF.
    - Our Mac is-a computer, and the monitor is part-of of a computer

```
 1  class Monitor {
 2
 3  };
 4
 5  class Computer {
 6  private:
 7    Monitor theMonitor; // a monitor is part-of a computer
 8  };
 9
10  // a mac is-a computer
11  class Mac : public Computer {
12
13  };
```

PasswordLockDoor

- Password lock doors are doors that require a password to open or close
- Let's represent the password as a string
- The password is "part-of" a PasswordLockDoor

```
 1  class PasswordLockDoor : public LockableDoor {
 2  public:
 3    PasswordLockDoor(const char c[]="");
 4    void unlock( const char c[]="");
 5  protected:
 6    string thepassword;
```

```
7  }
```

- The `PasswordLockDoor` object has the following member attributes (data) and member functions:

Member attributes	Member methods
isShut	isLocked()
theLock	open()
thepassword	lock()
	unlock(char[])
	isOpen()
	close()

Pointers to base classes

- Pointers can be made to point to derived classes
- Consider the following:

```
1  typedef Door* DoorPtr;
2  DoorPtr p = new Door(); // dynamically allocating a door
3  p->open();  // calls Door::open
4  ...
5  p = new LockableDoor();
6  p->open();  // which open??
```

virtual functions

- Late binding allows the selection of which implementation of a member function to execute to be determined at runtime
- C++ performs late binding via `virtual` functions
- Consider this generic base class

```
1  class Door {
2  public:
3    Door();
4    bool isOpen() const;
5    virtual void open();
```

```
6    void close();
7  protected:
8    bool isShut;
9  }
```

Auto example

- This example shows how `virtual` functions work. The key thing to note here is the definition of `Auto` functions as virtual in `Auto.h` and the use of `ptrAuto` in `Main.cpp`.

Example Auto.h

- Our `Auto` class will serve as our base class

```
1  #ifndef AUTO_H
2  #define AUTO_H
3  #include "Settings.h"
4
5  namespace cs52 {
6
7  class Auto {
8  public:
9      Auto( );
10
11 #ifdef USEVIRTUALFUNCTIONS
12     virtual void insertKey();
13     virtual void turn();
14     virtual void drive();
15 #else
16     void insertKey();
17     void turn();
18     void drive();
19 #endif
20 };
21
22 }
23
24 #endif
```

Example Auto.cpp

```
1   #include "Auto.h"
2   #include <iostream>
3
4   namespace cs52 {
5
6   Auto::Auto() {
7     // empty
8   }
9
10  void Auto::insertKey() {
11    using namespace std;
12    cout << "AUTO--inserting the key" << endl;
13  }
14
15  void Auto::turn() {
16    using namespace std;
17    cout << "AUTO--turning the key" << endl;
18  }
19
20  void Auto::drive() {
21    using namespace std;
22    cout << "AUTO--driving the car" << endl;
23  }
24
25  }
```

Example Honda.h

- The Honda class inherits from the Auto

```
1   #ifndef HONDA_H
2   #define HONDA_H
3   #include "Settings.h"
4   #include "Auto.h"
5
6   namespace cs52 {
7
8   class Honda : public Auto {
9   public:
10      Honda( );
11
```

```
12  #ifdef USEVIRTUALFUNCTIONS
13      virtual void insertKey();
14      virtual void turn();
15      virtual void drive();
16  #else
17      void insertKey();
18      void turn();
19      void drive();
20  #endif
21  };
22
23  }
24
25  #endif
```

Example Honda.cpp

```
1   #include "Honda.h"
2   #include <iostream>
3
4   namespace cs52 {
5
6   Honda::Honda() : Auto() {
7     // empty
8   }
9
10  void Honda::insertKey() {
11    using namespace std;
12    cout << "HONDA--waking up the mouse..." << endl;
13  }
14
15  void Honda::turn() {
16    using namespace std;
17    cout << "HONDA--feeding the mouse..." << endl;
18  }
19
20  void Honda::drive() {
21    using namespace std;
22    cout << "HONDA--mouse is turning the wheels..." << endl;
23  }
24
25  }
```

Example Settings.h

- Use this file to control whether or not `Auto` has virtual functions or not.
- To prevent `Auto` from using `virtual` functions, just comment out the `#define` in this file

```
1  /// uncomment this to create virtual methods
2  #define USEVIRTUALFUNCTIONS
```

Example Main.cpp

- Take note of the final `PTRAUTO POINTING AT HONDA` portion
- We have an `Auto` pointer (`ptrAuto`) pointing to a `Honda` derived class, and due to the use of `virtual` functions, the `Auto` pointer calls the `Honda` versions of the functions
  - One would think, given `ptrAuto` is a pointer to the base class `Auto`, that these calls would call the `Auto` versions of the functions. But because they are virtual functions in `Auto`, the derived class's versions are called.
- This is late binding at runtime!

```cpp
1  #include "Auto.h"
2  #include "Honda.h"
3  #include <iostream>
4
5
6  int main() {
7    using namespace std;
8    using namespace cs52;
9
10   cout << "------AUTO------" << endl;
11   Auto a;
12   a.insertKey();
13   a.turn();
14   a.drive();
15
16   cout << "------HONDA------" << endl;
17   Honda h;
18   h.insertKey();
19   h.turn();
20   h.drive();
21
```

```
22     cout << "doing the same thing with pointer variables..." << endl;
23     Auto * ptrAuto = NULL;
24     ptrAuto = &a;
25     cout << "------PTRAUTO POINTING AT AN AUTO------" << endl;
26     ptrAuto->insertKey();
27     ptrAuto->turn();
28     ptrAuto->drive();
29
30     ptrAuto = &h;
31     cout << "------PTRAUTO POINTING AT AN HONDA------" << endl;
32     // This is where the magic happens.
33     // remember ptrAuto is a pointer to an Auto, not to a Honda
34     // But since Auto has these marked as virtual functions,
35     // the derived class's (Honda) functions are called!
36     ptrAuto->insertKey();
37     ptrAuto->turn();
38     ptrAuto->drive();
39
40        return 0;
41  }
```