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:

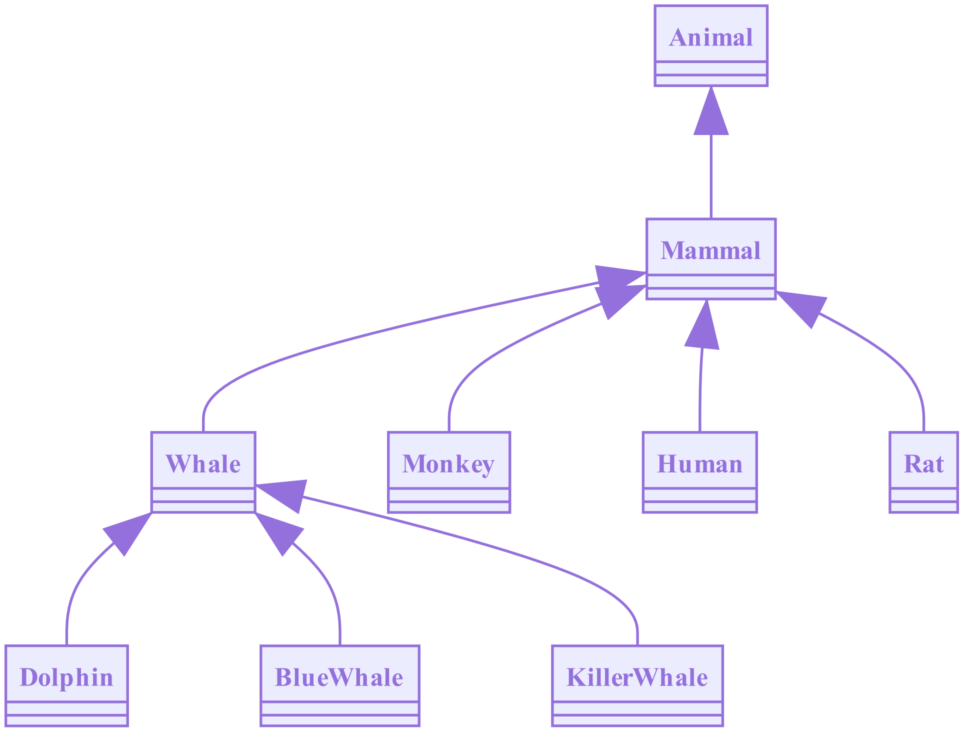

Figure 1: Partial hierarchy of mammals

- 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:

```
class Door {
public:
  Door();
  bool isOpen() const;
  void open();
  void close();
protected:
  bool isShut;
}
```

`protected` qualifier

- The `protected` qualifier is a compromise between `private` and `public`
  - `protected` is `public` to the base class
  - `protected` is `public` to `friend`s of the base class
  - `protected` is `public` to the derived classes
  - `protected` is `public` to `friend`s 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:

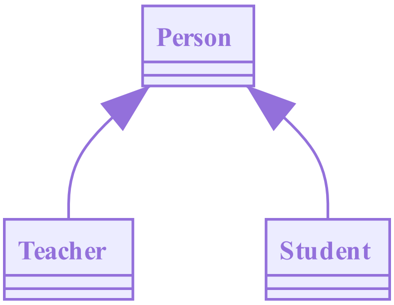

Figure 2: Person, Student, Teacher Hierarchy

Example Person.h

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

```
#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 );
    ~Person();
private:

```

```

protected:
    std::string my_name;
    std::string my_address;
    int* integer;

};

}
#endif
```

Example Person.cpp

```
#include "Person.h"

using namespace cs52;

namespace cs52 {

Person::Person() {

}

// note the use of an initializer list here
Person::Person( std::string name, std::string address ) : my_name( name ),
                                                          my_address( address ) {

}

std::string Person::getName() {
    return( my_name );
}

std::string Person::getAddress() {
    return( my_address );
}

void Person::setName( std::string name ) {
    my_name = name;
```

```
}

void Person::setAddress( std::string address ) {
    my_address = address;
}

std::ostream& operator<<( std::ostream& outs,
                          const Person& p ) {
    outs << "name=";
    outs << p.my_name;
    outs << " address=";
    outs << p.my_address;
    return( outs );
}

}
```

Example Student.h

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

```
#ifndef STUDENT_H
#define STUDENT_H

#include <iostream>
#include <string>

#include "Person.h"

namespace cs52 {

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

    std::string getGPA();
    void setGPA( std::string gpa );
    std::string getID();
    void setID( std::string id );

```

```

    friend std::ostream& operator<<( std::ostream& outs,
                                     const Student& s );
private:


protected:
    std::string my_ID;
    std::string my_GPA;
};

}
#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`

```
#include "Student.h"

using namespace cs52;

namespace cs52 {

Student::Student() {

}

// note the use of an initializer list here
Student::Student( std::string name, std::string address,
                  std::string id,   std::string gpa ) : Person( name, address ),
                                       my_ID( id ), my_GPA( gpa ) {

}

std::string Student::getGPA() {
    return( my_GPA );
}

void Student::setGPA( std::string gpa ) {
```

```
    my_GPA = gpa;
}

std::string Student::getID() {
    return( my_ID );
}

void Student::setID( std::string id ) {
    my_ID = id;
}

std::ostream& operator<<( std::ostream& outs,
                          const Student& s ) {
    Person p = s;
    outs << p;
    outs << " id=" << s.my_ID << " gpa=" << s.my_GPA;
    return( outs );
}

}
```

Example Teacher.h

- The `Teacher` class also inherits from the `Person`

```
#ifndef TEACHER_H
#define TEACHER_H

#include <iostream>
#include <string>

#include "Person.h"

namespace cs52 {

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

    std::string getDepartment();
```

```
    void setDepartment( std::string dept );

    friend std::ostream& operator<<( std::ostream& outs,
                                     const Teacher& t );
private:


protected:
    std::string my_department;

};

}
#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`

```
#include "Teacher.h"

using namespace cs52;

namespace cs52 {

Teacher::Teacher() {

}

// note the use of an initializer list here
Teacher::Teacher( std::string name, std::string address,
                  std::string dept ) : Person( name, address ),
                                       my_department( dept ) {
}

std::string Teacher::getDepartment() {
    return( my_department );
}

void Teacher::setDepartment( std::string dept ) {
    my_department = dept;
```

```
}

std::ostream& operator<<( std::ostream& outs,
                          const Teacher& t ) {
    Person p = t;
    outs << p;
    outs << " department=" << t.my_department;
    return( outs );
}

}
```

Example Main.cpp

```
#include <iostream>

#include "Person.h"
#include "Teacher.h"
#include "Student.h"


int main() {
    using namespace std;
    using namespace cs52;

    Person p( "Howie", "Los Angeles" );
    Teacher t( "HowieTeacher", "Santa Monica", "Business" );
    Student s( "Howie", "Los Angeles", "102", "3.5" );

    cout << p << endl;
    cout << t << endl;
    cout << s << endl;

    return( 0 );
}
```

Inheritance Syntax

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

```
class DerivedClass : public BaseClass
```

```
{
  // class definition
};
```

- 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

```
class BaseClass
{
public:
  int x;
  BaseClass(int x_, int y_, int z_) : x(x_), y(y_), z(z_) {}
protected:
  int y;
private:
  int z;
}
```

`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`

```
class DerivedClass : public BaseClass
{
  // x is public
  // y is protected
```

```
  // z is not accessible from DerivedClass
public:
  DerivedClass();
}
```

- **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

```
class DerivedClass : protected BaseClass
{
  // x is protected
  // y is protected
  // z is not accessible from DerivedClass
public:
  DerivedClass();
}
```

- **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

```
class DerivedClass : private BaseClass
{
  // x is private
  // y is private
  // z is not accessible from DerivedClass
public:
  DerivedClass();
}
```

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:

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

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

```
class LockableDoor : public Door {
public:
  LockableDoor();
  bool isLocked() const;
  void open();
  void lock();  void unlock();
protected:
  bool thelock;
}
```

- 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:

```
void LockableDoor::open( )
{
  if (!isLocked()) {
    Door::open();
  }
}
```

- 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

```
Door hallDoor;
LockableDoor frontDoor;

hallDoor.open();
frontDoor.lock();
frontDoor.open();
if (!frontDoor.isOpen())
   frontDoor.unlock();
```

Derived classes can become base classes

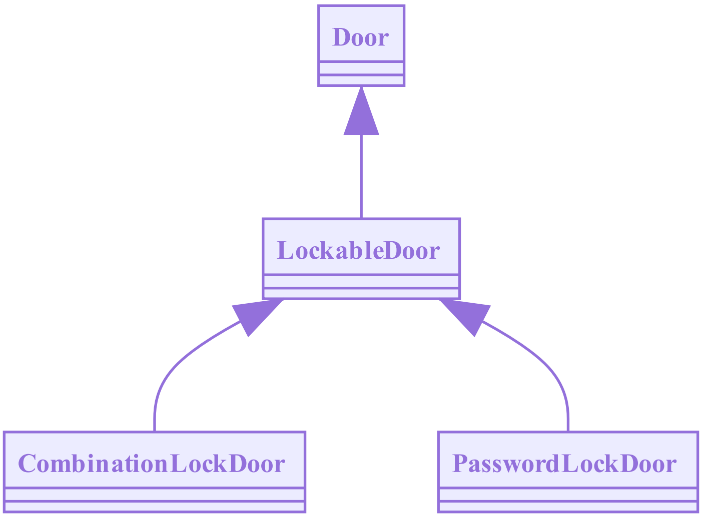

Figure 3: Partial hierarchy of mammals

`CombinationLockDoor`

```
class CombinationLockDoor : public LockableDoor {
public:
  CombinationLockDoor( int combo = 0);
  void unlock( int combo );
protected:
  int thecombination;
}
```

- 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

```
#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:

```

```

protected:
    std::string my_name;
    std::string my_address;

};

}
#endif
```

Example PersonPtr.cpp

```
#include "Person.h"

using namespace cs52;

namespace cs52 {

Person::Person() {

}

// note the use of an initializer list here
Person::Person( std::string name, std::string address ) : my_name( name ),
                                                          my_address( address ) {

}

std::string Person::getName() {
    return( my_name );
}

std::string Person::getAddress() {
    return( my_address );
}

void Person::setName( std::string name ) {
    my_name = name;
}
```

```

void Person::setAddress( std::string address ) {
    my_address = address;
}

std::ostream& operator<<( std::ostream& outs,
                          const Person& p ) {
    outs << "name=";
    outs << p.my_name;
    outs << " address=";
    outs << p.my_address;
    return( outs );
}

std::ostream& operator<<( std::ostream& outs,
                          const Person* p ) {
    if (p == NULL) {
        outs << "NULL pointer";
    }
    else {
        outs << "name=";
        outs << p->my_name;
        outs << " address=";
        outs << p->my_address;
    }
    return( outs );
}

}
```

Example StudentPtr.h

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

```
#ifndef STUDENT_H
#define STUDENT_H

#include <iostream>
#include <string>

#include "Person.h"

```

```cpp
namespace cs52 {

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

    std::string getGPA();
    void setGPA( std::string gpa );
    std::string getID();
    void setID( std::string id );


    friend std::ostream& operator<<( std::ostream& outs,
                                     const Student& s );
    friend std::ostream& operator<<( std::ostream& outs,
                                     const Student* s );

private:


protected:
    std::string my_ID;
    std::string my_GPA;
};

}
#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`

```cpp
#include "Student.h"

using namespace cs52;

namespace cs52 {

Student::Student() {
```

```

}

// note the use of an initializer list here
Student::Student( std::string name, std::string address,
                  std::string id,   std::string gpa ) : Person( name,
                      address ),
                                    my_ID( id ), my_GPA( gpa ) {
}

std::string Student::getGPA() {
    return( my_GPA );
}

void Student::setGPA( std::string gpa ) {
    my_GPA = gpa;
}

std::string Student::getID() {
    return( my_ID );
}

void Student::setID( std::string id ) {
    my_ID = id;
}

std::ostream& operator<<( std::ostream& outs,
                          const Student& s ) {
    Person p = s;
    outs << p;
    outs << " id=" << s.my_ID << " gpa=" << s.my_GPA;
    return( outs );
}

std::ostream& operator<<( std::ostream& outs,
                          const Student* s ) {
    if (s == NULL) {
        outs << "NULL pointer";
    }
    else {
        const Person* p = s;
        outs << p;
        outs << " id=" << s->my_ID << " gpa=" << s->my_GPA;
```

```
    }
    return( outs );
}

}
```

Example TeacherPtr.h

- The `Teacher` class also inherits from the `Person`

```
#ifndef TEACHER_H
#define TEACHER_H

#include <iostream>
#include <string>

#include "Person.h"

namespace cs52 {

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

    std::string getDepartment();
    void setDepartment( std::string dept );

    friend std::ostream& operator<<( std::ostream& outs,
                                     const Teacher& t );
    friend std::ostream& operator<<( std::ostream& outs,
                                     const Teacher* t );

private:


protected:
    std::string my_department;

};

```

```
}
#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`

```
#include "Teacher.h"

using namespace cs52;

namespace cs52 {

Teacher::Teacher() {

}

// note the use of an initializer list here
Teacher::Teacher( std::string name, std::string address,
                  std::string dept ) : Person( name, address ),
                                       my_department( dept ) {
}

std::string Teacher::getDepartment() {
    return( my_department );
}

void Teacher::setDepartment( std::string dept ) {
    my_department = dept;
}

std::ostream& operator<<( std::ostream& outs,
                          const Teacher& t ) {
    Person p = t;
    outs << p;
    outs << " department=" << t.my_department;
    return( outs );
}

std::ostream& operator<<( std::ostream& outs,
                          const Teacher* t ) {
```

```
    if (t == NULL) {
        outs << "NULL pointer";
    }
    else {
        const Person* p = t;
        outs << p;
        outs << " department=" << t->my_department;
    }
    return( outs );
}

}
```

Example MainPtr.cpp

```
#include <iostream>

#include "Person.h"
#include "Teacher.h"
#include "Student.h"


int main() {
    using namespace std;
    using namespace cs52;

    Person* p = new Person( "Howie", "Los Angeles" );
    Teacher* t = new Teacher( "HowieTeacher", "Santa Monica", "Business
       " );
    Student* s = new Student( "Howie", "Los Angeles", "102", "3.5" );

    cout << p << endl;
    cout << t << endl;
    cout << s << endl;

    delete( p );
    delete( t );
    delete( s );

    return( 0 );
}
```

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

```
class Monitor {

};

class Computer {
private:
  Monitor theMonitor; // a monitor is part-of a computer
};

// a mac is-a computer
class Mac : public Computer {

};
```

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`

```
class PasswordLockDoor : public LockableDoor {
public:
  PasswordLockDoor(const char c[]="");
  void unlock( const char c[]="");
protected:
  string thepassword;
```

```
}
```

- 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:

```
typedef Door* DoorPtr;
DoorPtr p = new Door(); // dynamically allocating a door
p->open();  // calls Door::open
...
p = new LockableDoor();
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

```
class Door {
public:
  Door();
  bool isOpen() const;
  virtual void open();
```

```
    void close();
protected:
    bool isShut;
}
```

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

```
#ifndef AUTO_H
#define AUTO_H
#include "Settings.h"

namespace cs52 {

class Auto {
public:
    Auto( );

#ifdef USEVIRTUALFUNCTIONS
    virtual void insertKey();
    virtual void turn();
    virtual void drive();
#else
    void insertKey();
    void turn();
    void drive();
#endif
};

}

#endif
```

Example Auto.cpp

```cpp
#include "Auto.h"
#include <iostream>

namespace cs52 {

Auto::Auto() {
  // empty
}

void Auto::insertKey() {
  using namespace std;
  cout << "AUTO--inserting the key" << endl;
}

void Auto::turn() {
  using namespace std;
  cout << "AUTO--turning the key" << endl;
}

void Auto::drive() {
  using namespace std;
  cout << "AUTO--driving the car" << endl;
}

}
```

Example Honda.h

- The `Honda` class inherits from the `Auto`

```cpp
#ifndef HONDA_H
#define HONDA_H
#include "Settings.h"
#include "Auto.h"

namespace cs52 {

class Honda : public Auto {
public:
    Honda( );

```

```
#ifdef USEVIRTUALFUNCTIONS
    virtual void insertKey();
    virtual void turn();
    virtual void drive();
#else
    void insertKey();
    void turn();
    void drive();
#endif
};

}

#endif
```

Example Honda.cpp

```
#include "Honda.h"
#include <iostream>

namespace cs52 {

Honda::Honda() : Auto() {
  // empty
}

void Honda::insertKey() {
  using namespace std;
  cout << "HONDA--waking up the mouse..." << endl;
}

void Honda::turn() {
  using namespace std;
  cout << "HONDA--feeding the mouse..." << endl;
}

void Honda::drive() {
  using namespace std;
  cout << "HONDA--mouse is turning the wheels..." << endl;
}

}
```

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

```
/// uncomment this to create virtual methods
#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
- This is late binding at runtime!

```
#include "Auto.h"
#include "Honda.h"
#include <iostream>


int main() {
  using namespace std;
  using namespace cs52;

  cout << "------AUTO------" << endl;
  Auto a;
  a.insertKey();
  a.turn();
  a.drive();

  cout << "------HONDA------" << endl;
  Honda h;
  h.insertKey();
  h.turn();
  h.drive();

  cout << "doing the same thing with pointer variables..." << endl;
  Auto * ptrAuto = NULL;
  ptrAuto = &a;
```

```
    cout << "------PTRAUTO POINTING AT AN AUTO------" << endl;
    ptrAuto->insertKey();
    ptrAuto->turn();
    ptrAuto->drive();

    ptrAuto = &h;
    cout << "------PTRAUTO POINTING AT AN HONDA------" << endl;
    // This is where the magic happens.
    // remember ptrAuto is a pointer to an Auto, not to a Honda
    // But since Auto has these marked as virtual functions,
    // the derived class's (Honda) functions are called!
    ptrAuto->insertKey();
    ptrAuto->turn();
    ptrAuto->drive();

      return 0;
}
```