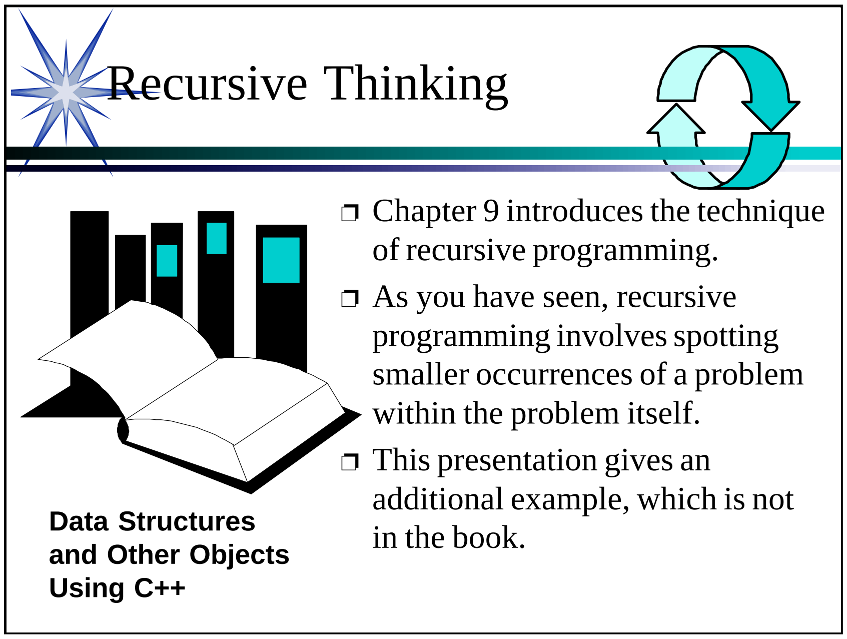


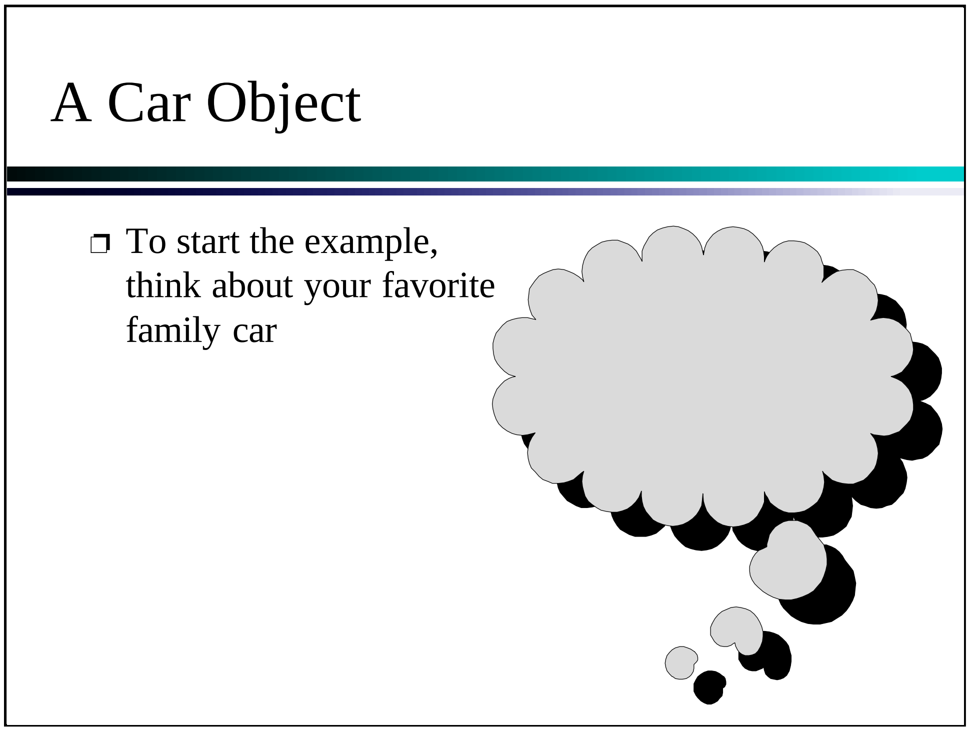
Recursive Thinking

- Chapter 9 introduces the technique of recursive programming.
- As you have seen, recursive programming involves spotting smaller occurrences of a problem within the problem itself.
- This presentation gives an additional example, which is not in the book.

Data Structures and Other Objects Using C++

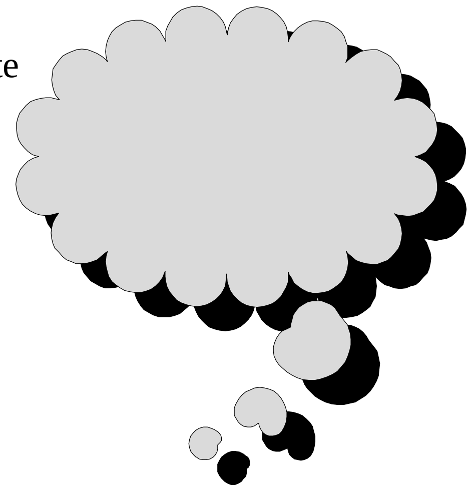
This lecture demonstrates a typical pattern that arises in recursive functions. The lecture can be given shortly before or shortly after the students have read Section 9.1.

By the way, the book delays the introduction of recursion until just before trees. Our reasoning here is that recursive functions with linked lists can sometimes be bad programming practice because the run-time stack must grow to the length of the linked list (and iterative algorithms are usually just as easy). On the other hand, the depth of recursive tree algorithms is merely the depth of the tree (and equivalent iterative algorithms can be harder to program).



A Car Object

- To start the example, think about your favorite family car

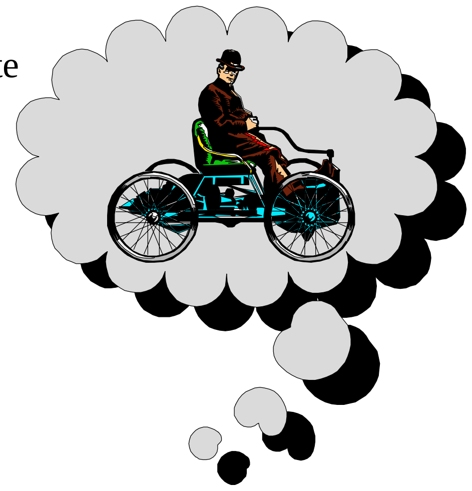


In this lecture, we're going to design a recursive function which manipulates a new object called a Car. In order to explain the Car that I have in mind, think about your favorite family car...

NOTE: In Powerpoint, the next few slides will automatically appear every few seconds...

A Car Object

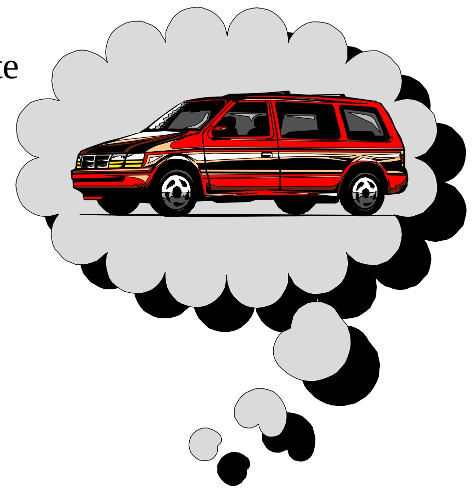
- ❒ To start the example, think about your favorite family car



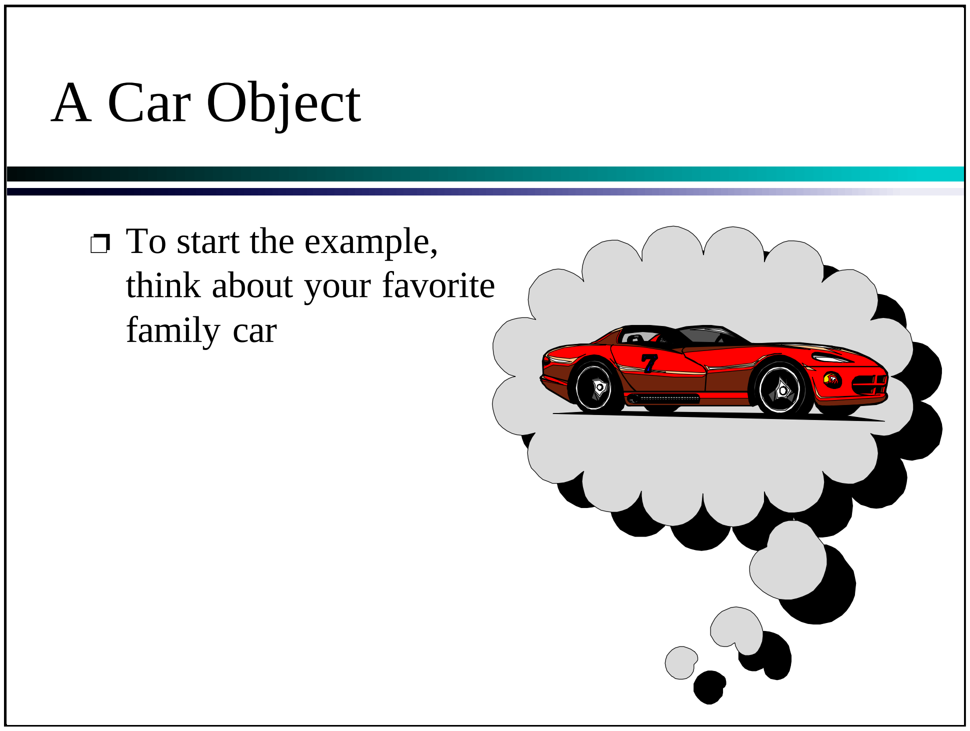
...let's try for something a bit more modern that this...

A Car Object

- To start the example, think about your favorite family car

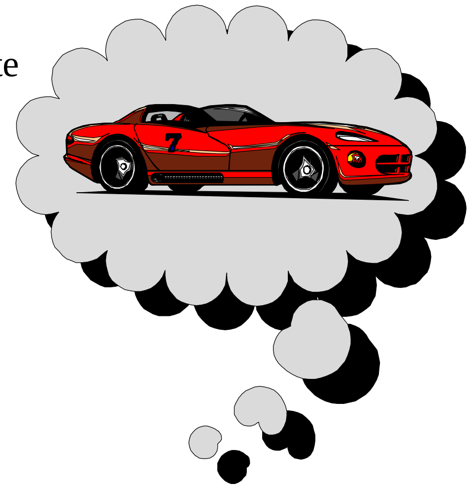


...no, that's too conservative...

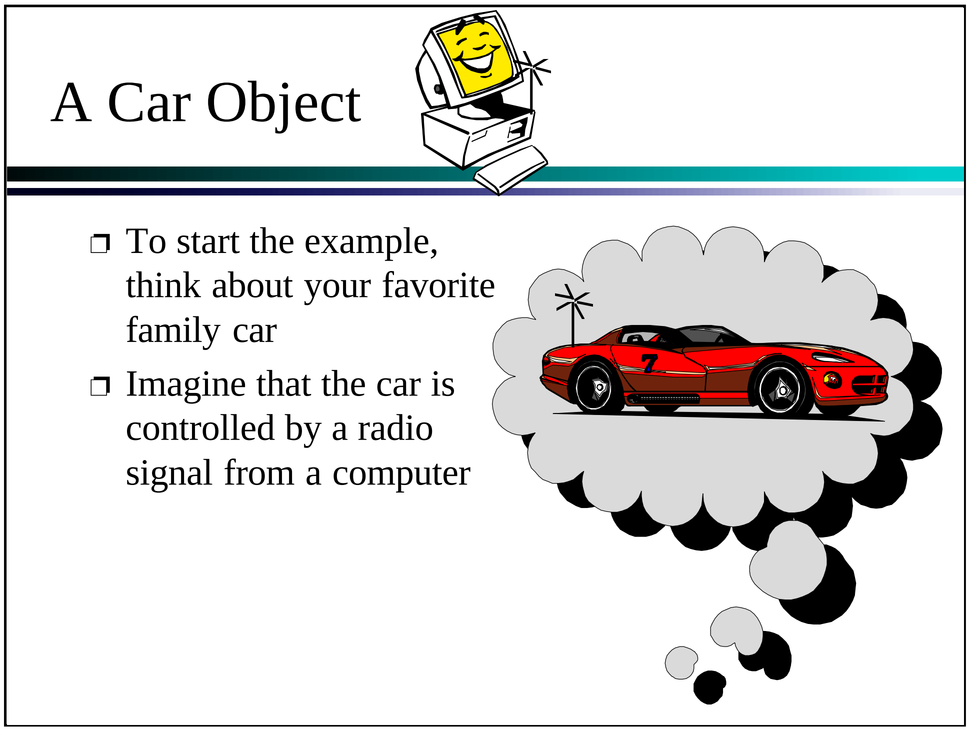


A Car Object

- To start the example, think about your favorite family car

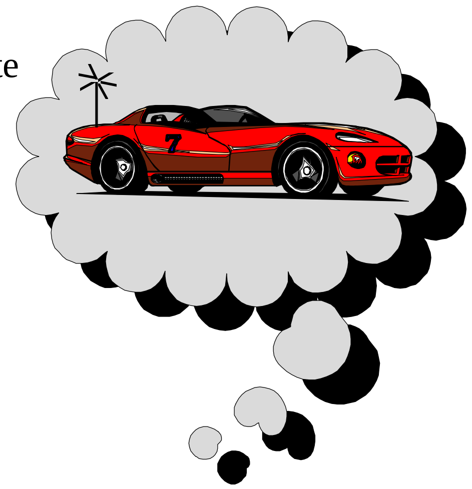


...that's better!

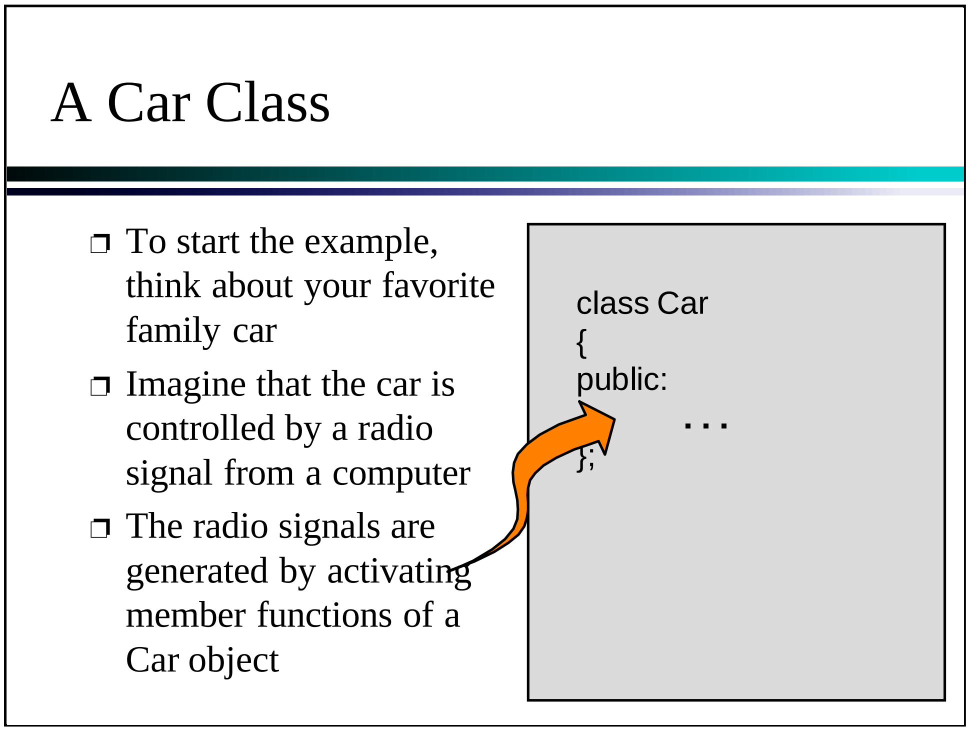


A Car Object

- To start the example, think about your favorite family car
- Imagine that the car is controlled by a radio signal from a computer



I want you to imagine that this family car is being controlled by a radio signal coming from your computer.



A Car Class

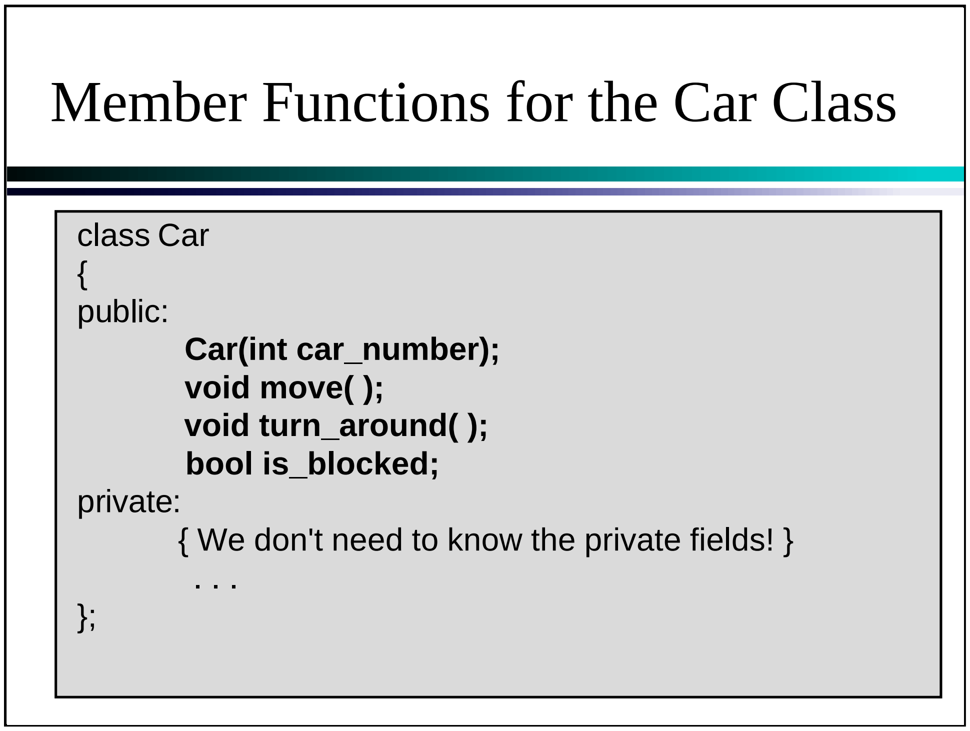
- To start the example, think about your favorite family car
- Imagine that the car is controlled by a radio signal from a computer
- The radio signals are generated by activating member functions of a Car object

```
class Car
{
public:
        . . .
};
```

The radio signals themselves are generated by writing a program which uses a new object type called a Car. Each time one of the car methods is activated, the computer sends a radio signal to control the car.

This may seem a bit far-fetched, but such radio-controlled hardware really does exist, and the hardware is controlled by programs which could have objects such as these. A similar example is an automatic baggage system at an airport, which is controlled by a computer.

Also, remember that in order to understand this example, you don't need to know all the details of <u>how</u> the Car's methods work. All you need to know is <u>what</u> each of the Car's methods accomplishes. I have four particular methods in mind, and I'll describe <u>what</u> each method does.



Member Functions for the Car Class

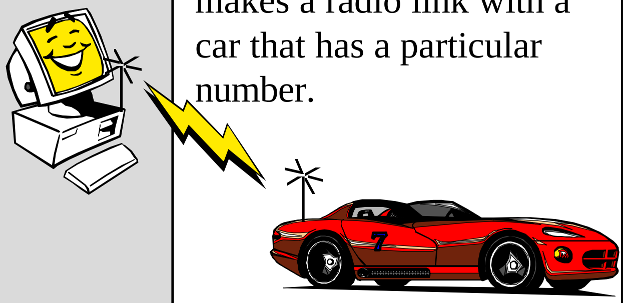
```
class Car
{
public:
            Car(int car_number);
            void move( );
            void turn_around( );
            bool is_blocked;
private:
           { We don't need to know the private fields! }
             . . .
};
```

Here are the prototypes of the four member functions. Let's look at each method in turn...

The Constructor

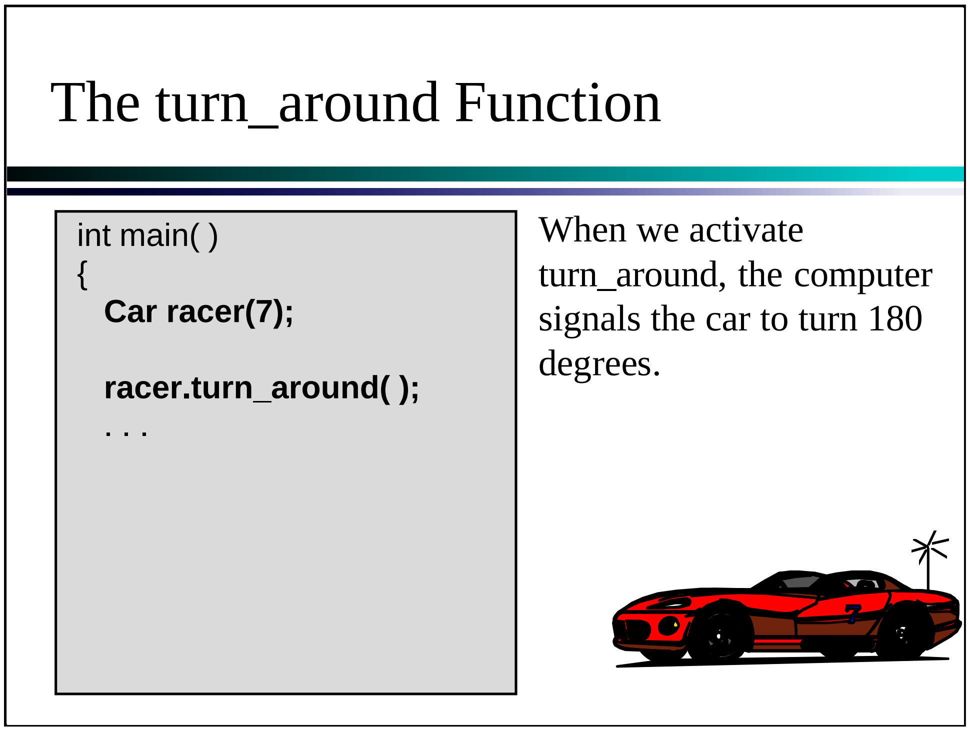
```
int main( )
{
   Car racer(7);

   . . .
```

When we declare a Car and activate the constructor, the computer makes a radio link with a car that has a particular number.



This example demonstrates the usage of the constructor. In this example, we have declared a Car called racer and the activated the constructor with the argument 7.

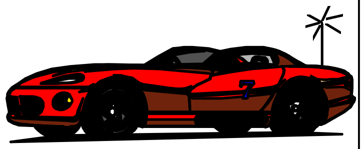
The argument 7 just indicates which of several cars we want to control. Each car needs to have a unique number. In this example, we are making a radio connection to "Car Number 7". Any future radio signals generated by activating racer's methods will control "Car Number 7".



The turn_around Function

```
int main( )
{
   Car racer(7);

   racer.turn_around( );
   . . .
```

When we activate turn_around, the computer signals the car to turn 180 degrees.



After the connection is made, we can activate other member functions of racer. In this example, we are activating

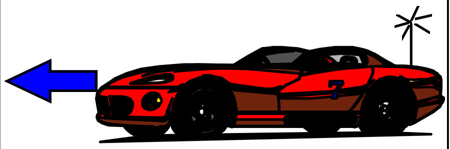
racer.turn_around( );

The function sends a radio signal to the car, telling it to turn 180 degrees. (In case you hadn't noticed, the car has a very small turning radius!)

The move Function

```
int main( )
{
   Car racer(7);

   racer.turn_around( );
   racer.move( );
   . . .
```

When we activate move, the computer signals the car to move forward one foot.



Here's an example of the third Car member function in action. When we activate

racer.move( );

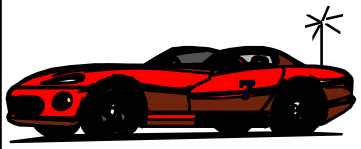
the result is a radio signal which tells the car to move forward one foot.

NOTE: In Powerpoint, the next slide automatically appears, moving the car forward "one foot".

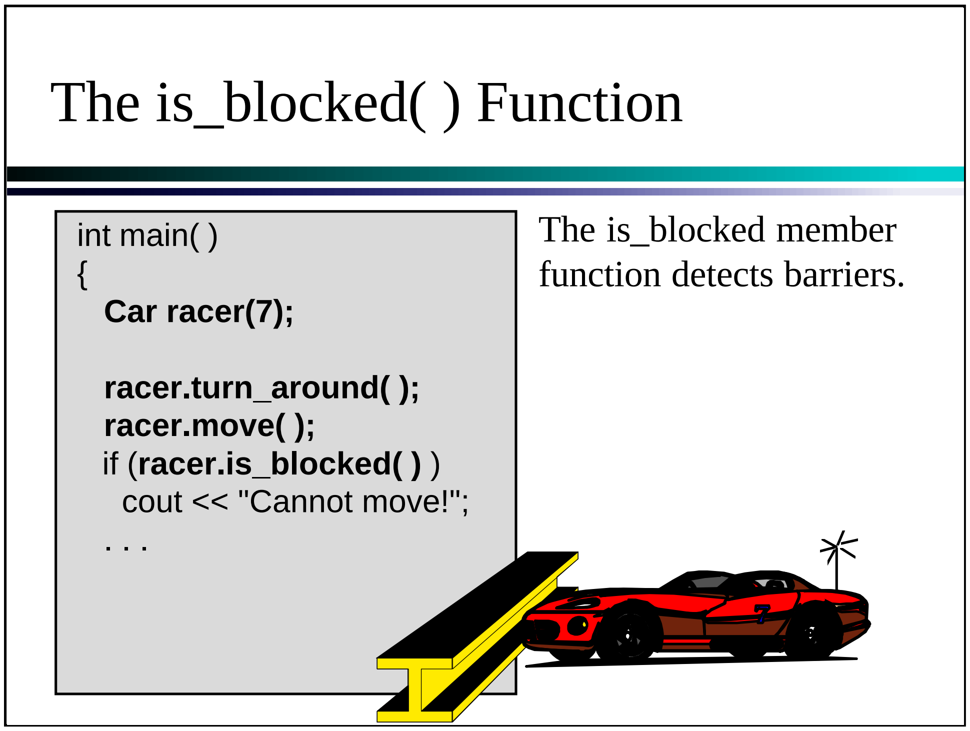
The move Function

```
int main( )
{
   Car racer(7);

   racer.turn_around( );
   racer.move( );
   . . .
```

When we activate move, the computer signals the car to move forward one foot.



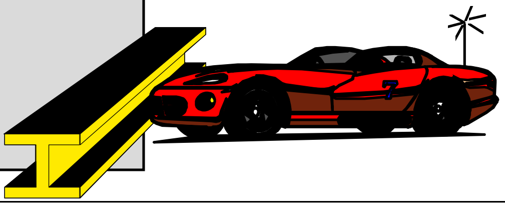
As you can see, the car has moved forward by a foot.



The is_blocked( ) Function

```
int main( )
{
   Car racer(7);

   racer.turn_around( );
   racer.move( );
   if (racer.is_blocked( ) )
     cout << "Cannot move!";
   . . .
```

The is_blocked member function detects barriers.



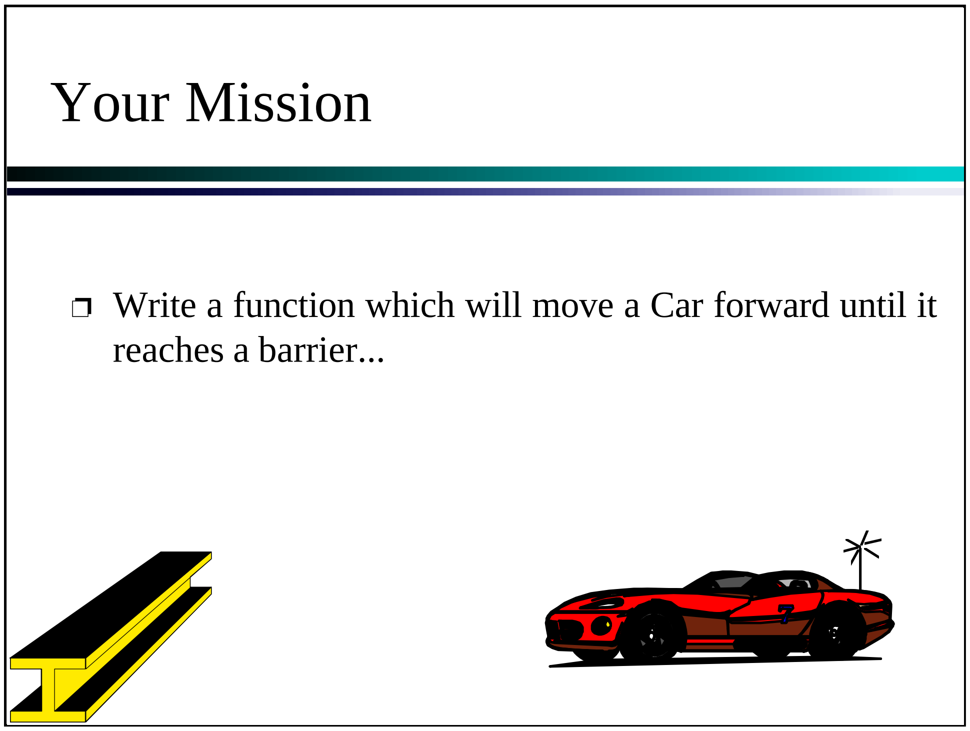
The last Car member function is a boolean function called is_blocked. The function returns true if there is some barrier in front of the car, making it impossible to move. The function returns false if the front of the car is unblocked, making a move possible.

A question: What will racer.is_blocked return in this example?

Answer: True, because there is a definite barrier in front of the car.

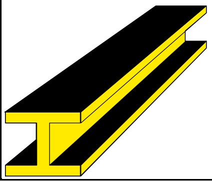
Another question: Can you state a good precondition for the move member function?

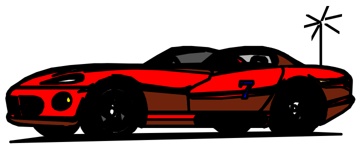
Answer: is_blocked must return false.



Your Mission

- ❒ Write a function which will move a Car forward until it reaches a barrier...

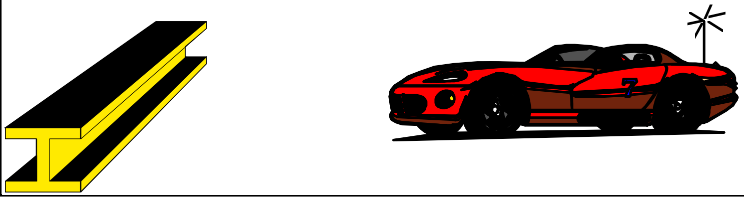




Now we can examine the problem that I have in mind. I want to write a function which will move a Car forward until it reaches a barrier...

Your Mission

- ❐ Write a function which will move a Car forward until it reaches a barrier...

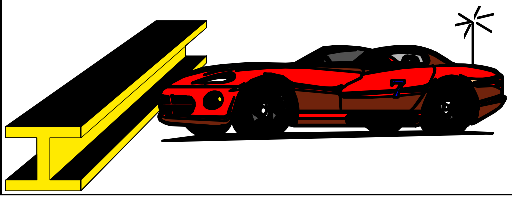


...as shown here.

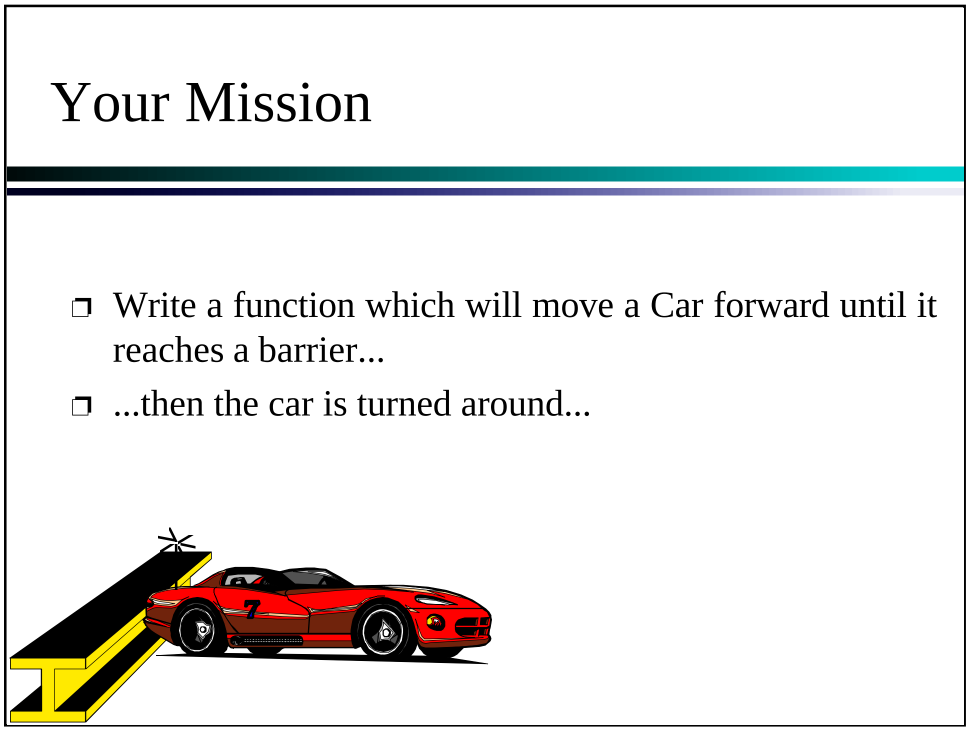
NOTE: In Powerpoint, this slide automatically appears after six seconds. Each of the next few slides automatically appears after a few more seconds.

Your Mission

- ❐ Write a function which will move a Car forward until it reaches a barrier...

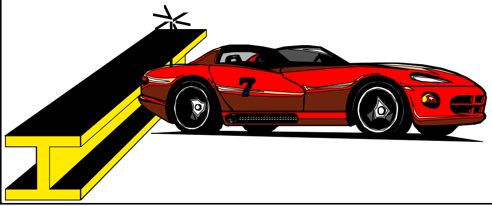


Here, you can see that the barrier has been reached. But that is not the end of the function's work. The function must next...

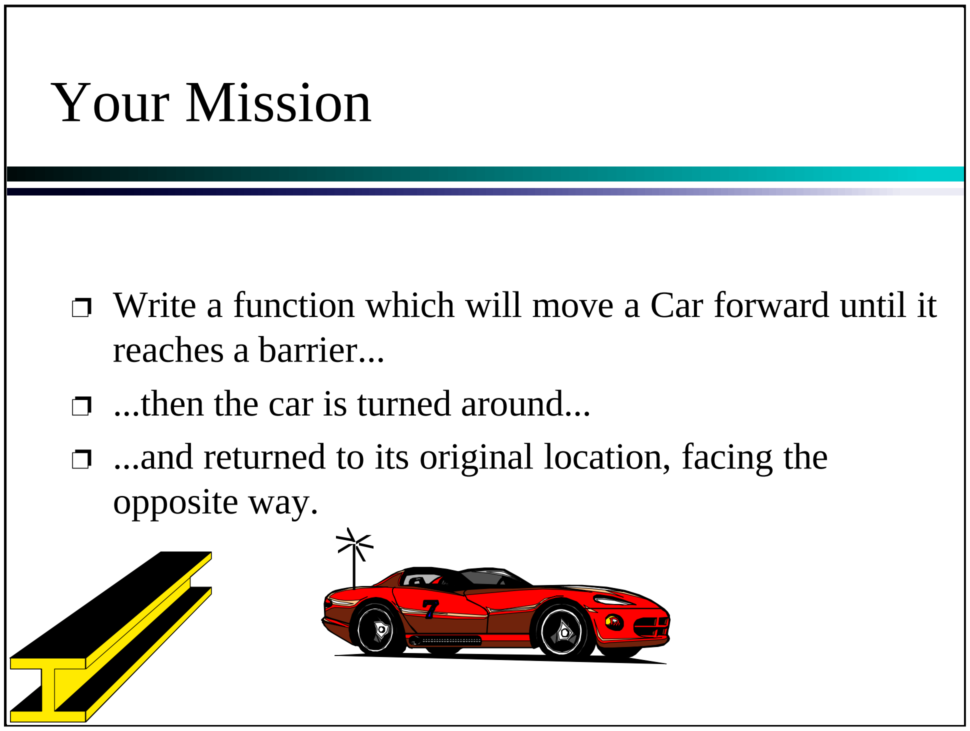


Your Mission

- ❐ Write a function which will move a Car forward until it reaches a barrier...
- ❐ ...then the car is turned around...

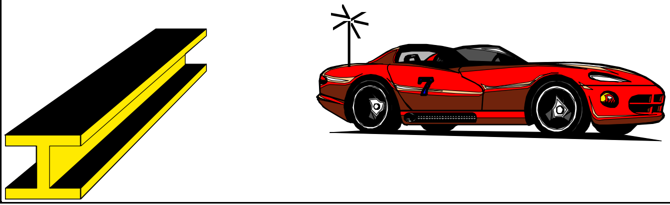


...turn the car around...



Your Mission

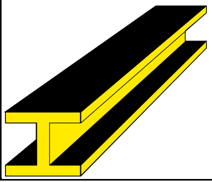
- ❐ Write a function which will move a Car forward until it reaches a barrier...
- ❐ ...then the car is turned around...
- ❐ ...and returned to its original location, facing the opposite way.

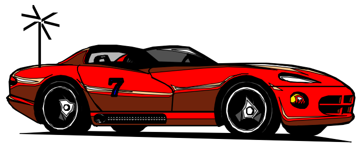


...and move the car back to its original position, facing the opposite direction.

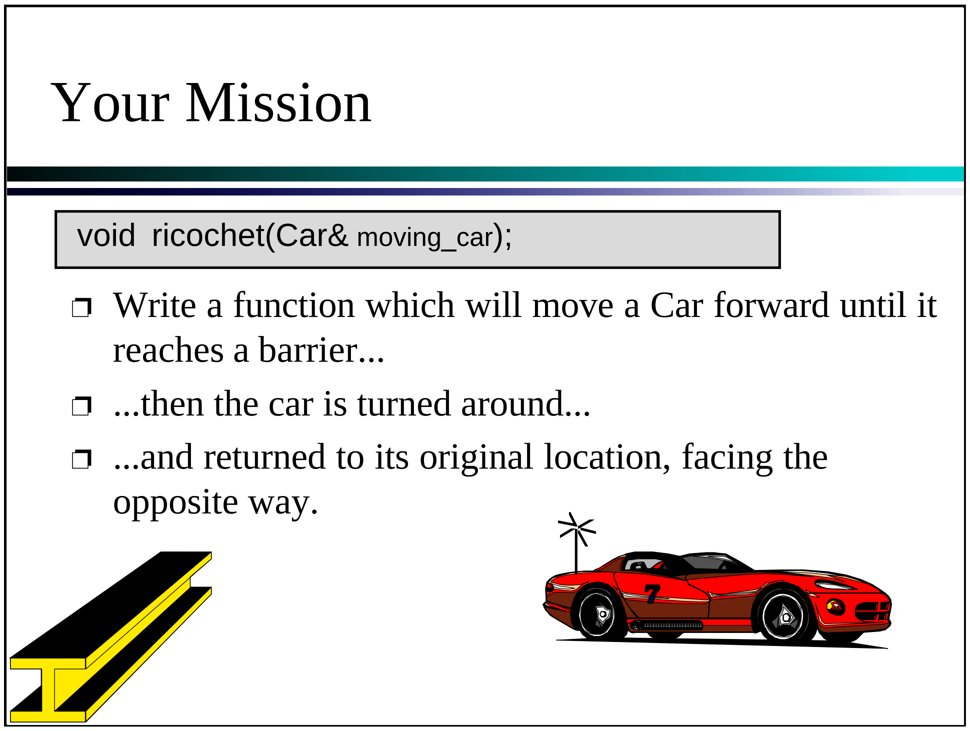
Your Mission

- ❒ Write a function which will move a Car forward until it reaches a barrier...
- ❒ ...then the car is turned around...
- ❒ ...and returned to its original location, facing the opposite way.





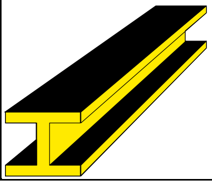
This is the end of the function's work.

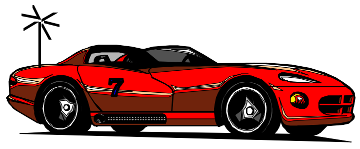


Your Mission

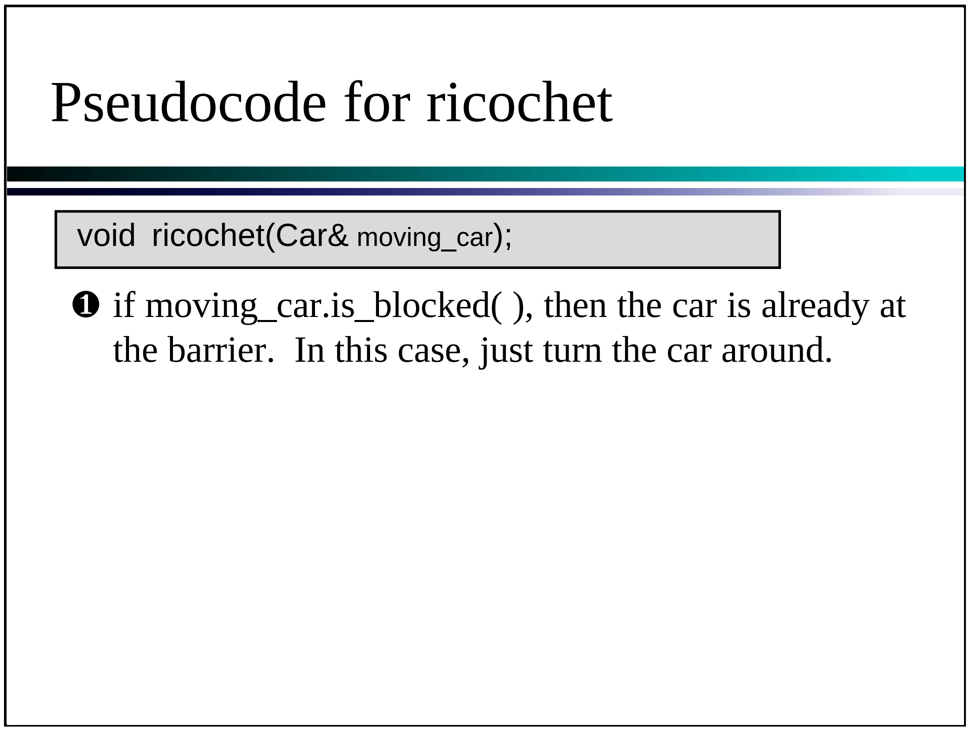
```
void  ricochet(Car& moving_car);
```

- Write a function which will move a Car forward until it reaches a barrier...
- ...then the car is turned around...
- ...and returned to its original location, facing the opposite way.





The function, called ricochet, has the heading shown here. There is one parameter, called moving_car, which is the Car that the function manipulates. We will assume that the radio connection has already been made for this car, so all the function has to do is issue the necessary move and turn_around commands.



Pseudocode for ricochet

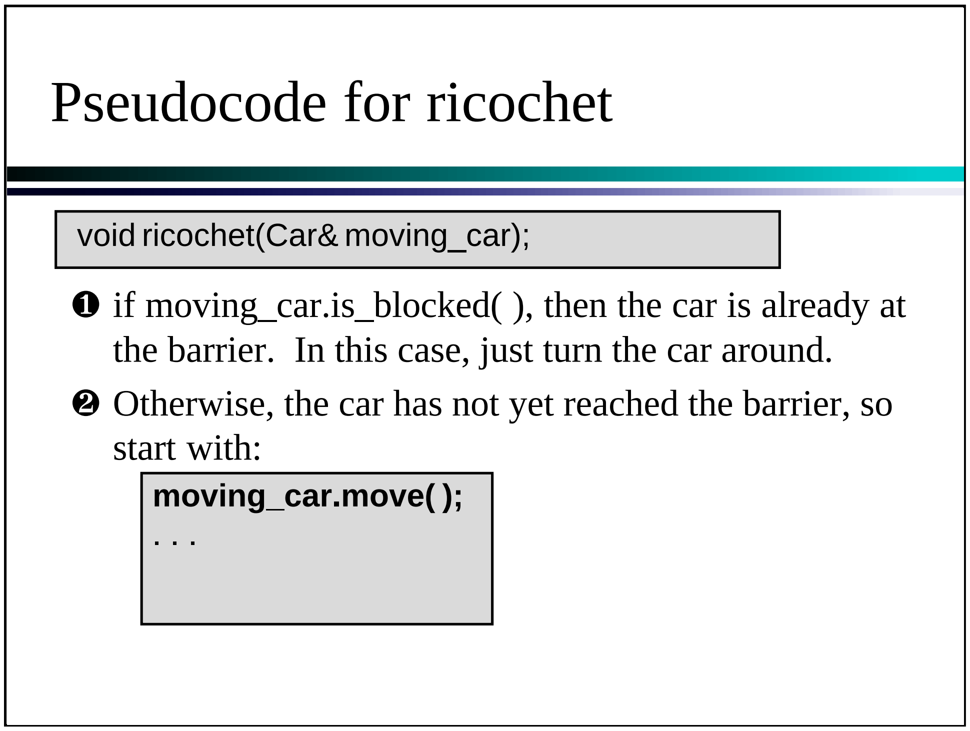
```
void  ricochet(Car& moving_car);
```

❶ if moving_car.is_blocked( ), then the car is already at the barrier. In this case, just turn the car around.

The first action the function takes is to check for a very simple case: the case where the car is already blocked.

In this case, no movement is needed. All that the function needs to do is turn the car around, and leave it in the same place.

NOTE: The lecturer can pretend to be the car, blocked at a wall, to demonstrate this simple case.



Pseudocode for ricochet

```
void ricochet(Car& moving_car);
```

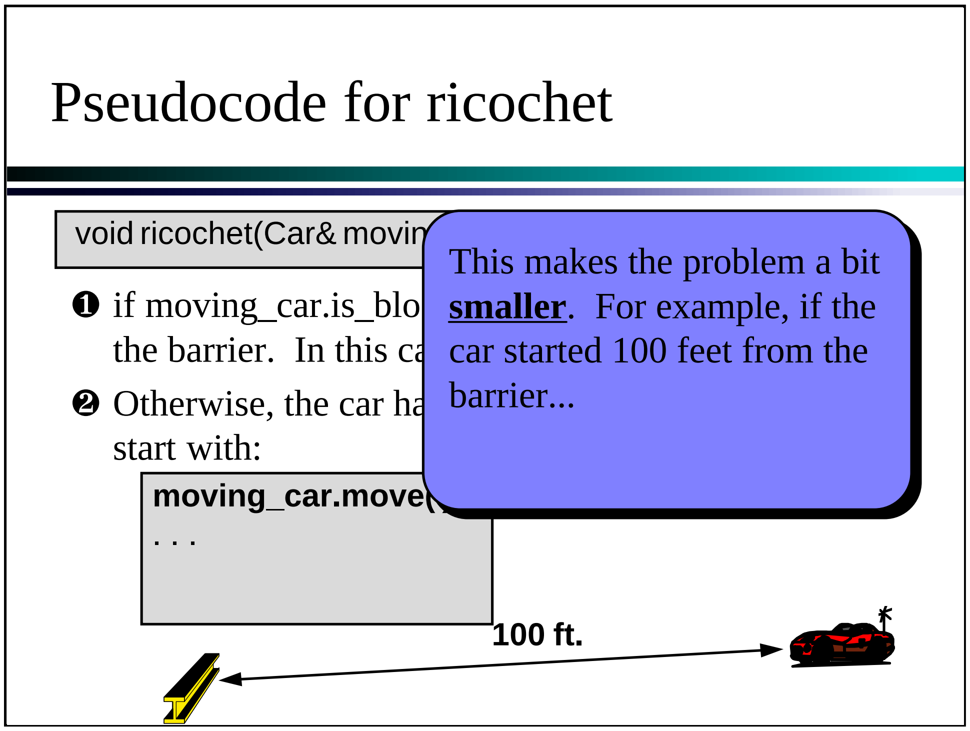
❶ if moving_car.is_blocked( ), then the car is already at the barrier. In this case, just turn the car around.

❷ Otherwise, the car has not yet reached the barrier, so start with:

```
moving_car.move( );
. . .
```

On the other hand, if the car is not blocked, then some movement is needed. The function begins the movement by moving the car forward just one foot.

Notice that the move activation is not in a loop. We're just moving one foot, with the primary goal being to make a smaller version of the original problem. In fact, once the car has moved just a single foot forward, we do have a smaller version of the original problem...



Pseudocode for ricochet

```
void ricochet(Car& movin
```

❶ if moving_car.is_blo
the barrier. In this ca

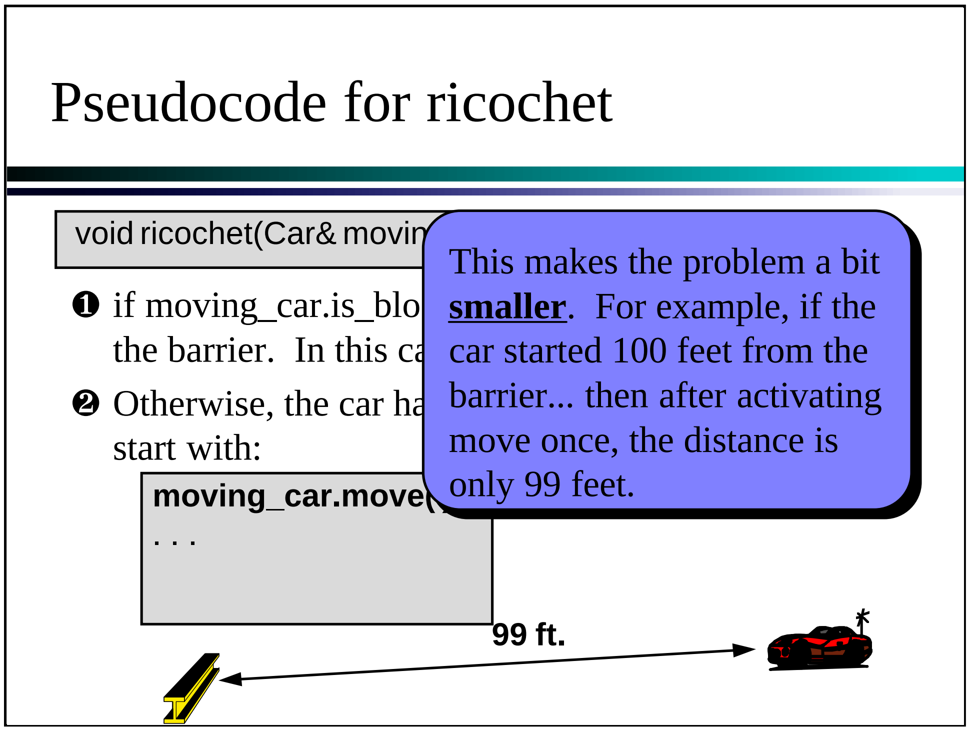
❷ Otherwise, the car ha
start with:

```
moving_car.move(
. . .
```

This makes the problem a bit **smaller**. For example, if the car started 100 feet from the barrier...

...as shown in this example.

In this example, the car starts 100 feet from a barrier.



Pseudocode for ricochet

```
void ricochet(Car& movin
```

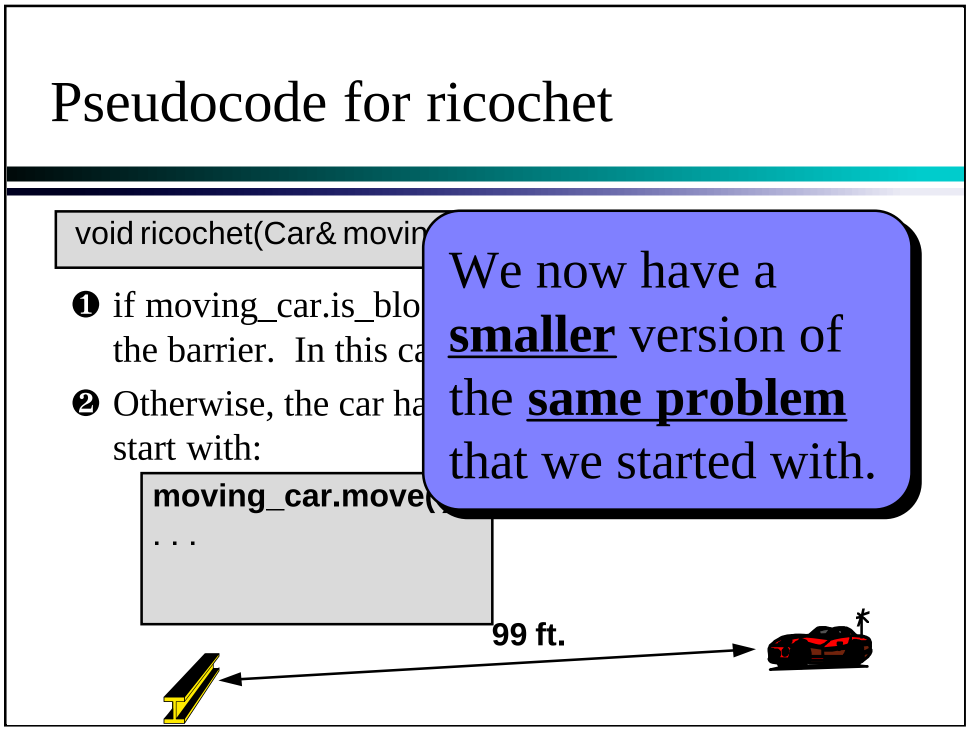
❶ if moving_car.is_blo
the barrier. In this ca

❷ Otherwise, the car ha
start with:

```
moving_car.move(
. . .
```

This makes the problem a bit **smaller**. For example, if the car started 100 feet from the barrier... then after activating move once, the distance is only 99 feet.

After moving just one foot, the car is 99 feet from the barrier.



Pseudocode for ricochet

```
void ricochet(Car& movin
```

❶ if moving_car.is_blo
the barrier. In this ca

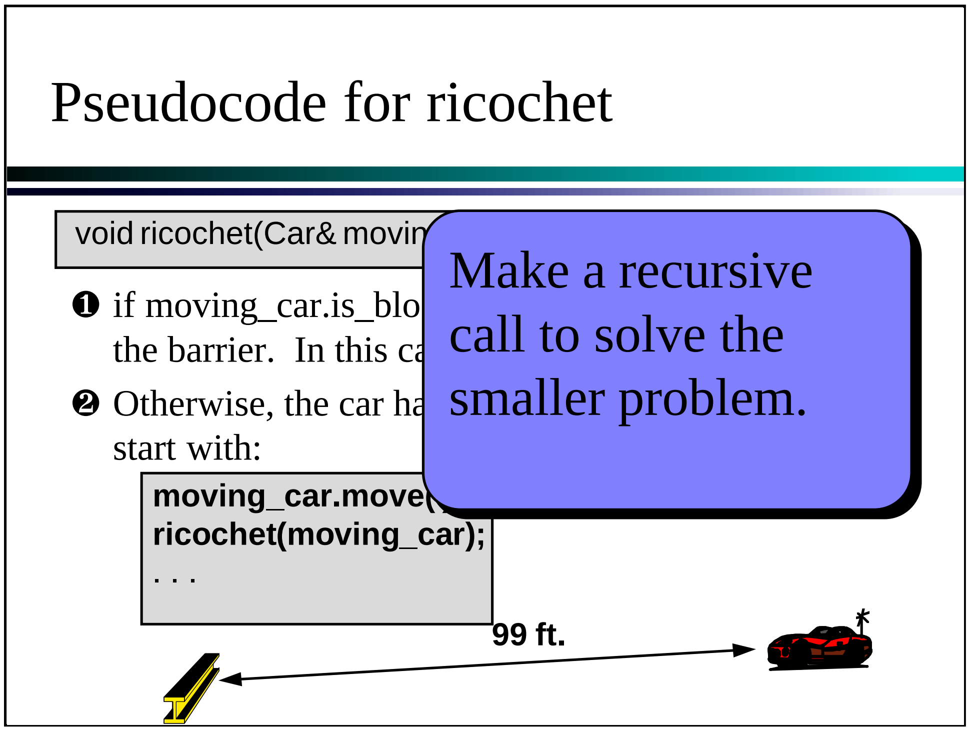
❷ Otherwise, the car ha
start with:

```
moving_car.move(
. . .
```

We now have a **smaller** version of the **same problem** that we started with.

99 ft.

Since 99 feet is less than 100 feet, we have a smaller version of the original problem.



Pseudocode for ricochet

void ricochet(Car& movin

❶ if moving_car.is_blo
the barrier. In this ca

❷ Otherwise, the car ha
start with:

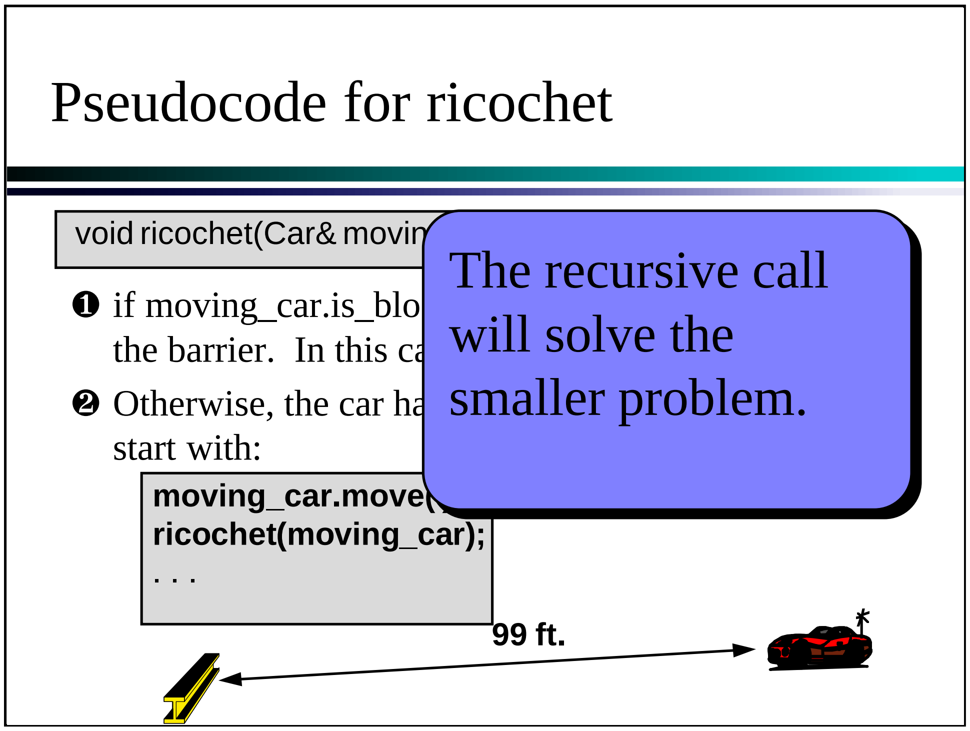
```
moving_car.move(
ricochet(moving_car);
. . .
```

Make a recursive call to solve the smaller problem.

Once a smaller problem has been created, we can make the key step of any recursive function:

The ricochet function calls itself to solve the smaller problem!

The act of a function calling itself is recursion, or in other words a recursive call. In order for recursion to succeed, the recursive call must be asked to solve a problem which is somehow smaller than the original problem. The precise meaning of "smaller" is given in Section 9.3, but for now you can use your intuition about what a "smaller" problem is.



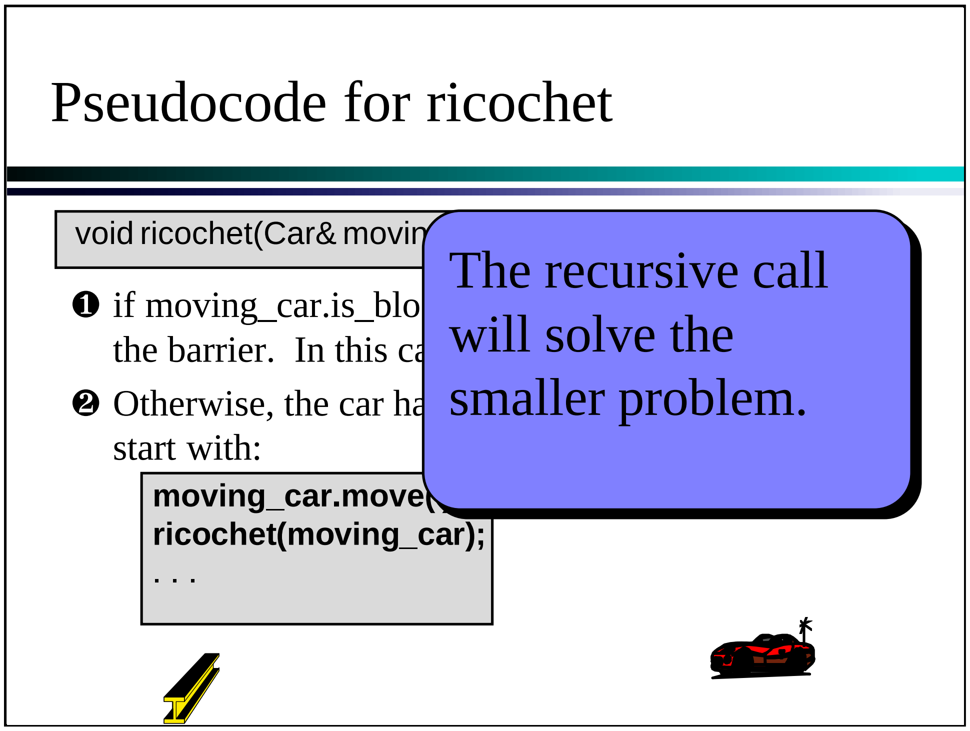
Pseudocode for ricochet

```
void ricochet(Car& movin
❶ if moving_car.is_blo
  the barrier.  In this ca
❷ Otherwise, the car ha
  start with:
    moving_car.move(
    ricochet(moving_car);
    . . .
```

The recursive call will solve the smaller problem.

The key idea to remember is this: When you make a recursive call to solve a smaller problem, the recursive call will successfully solve that smaller problem.

In this case, that means that the recursive call will move the car 99 feet to the barrier...



Pseudocode for ricochet

```
void ricochet(Car& movin
```

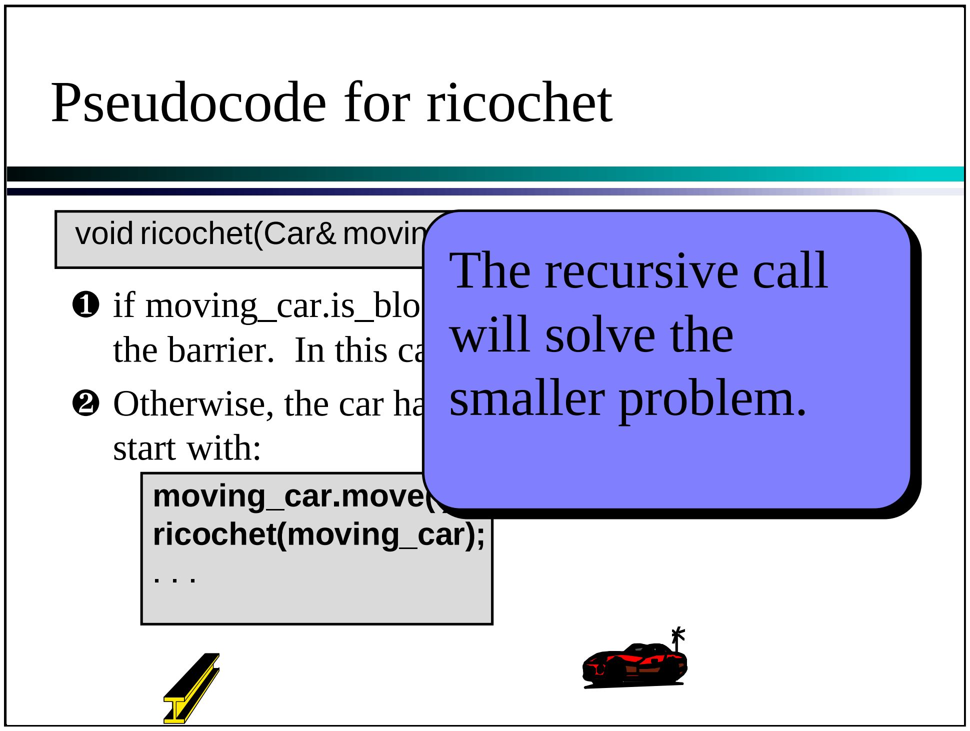
❶ if moving_car.is_blo
the barrier. In this ca

❷ Otherwise, the car ha
start with:

```
moving_car.move(
ricochet(moving_car);
. . .
```

The recursive call will solve the smaller problem.

...as shown here and in the next few slides.



Pseudocode for ricochet

```
void ricochet(Car& movin
```

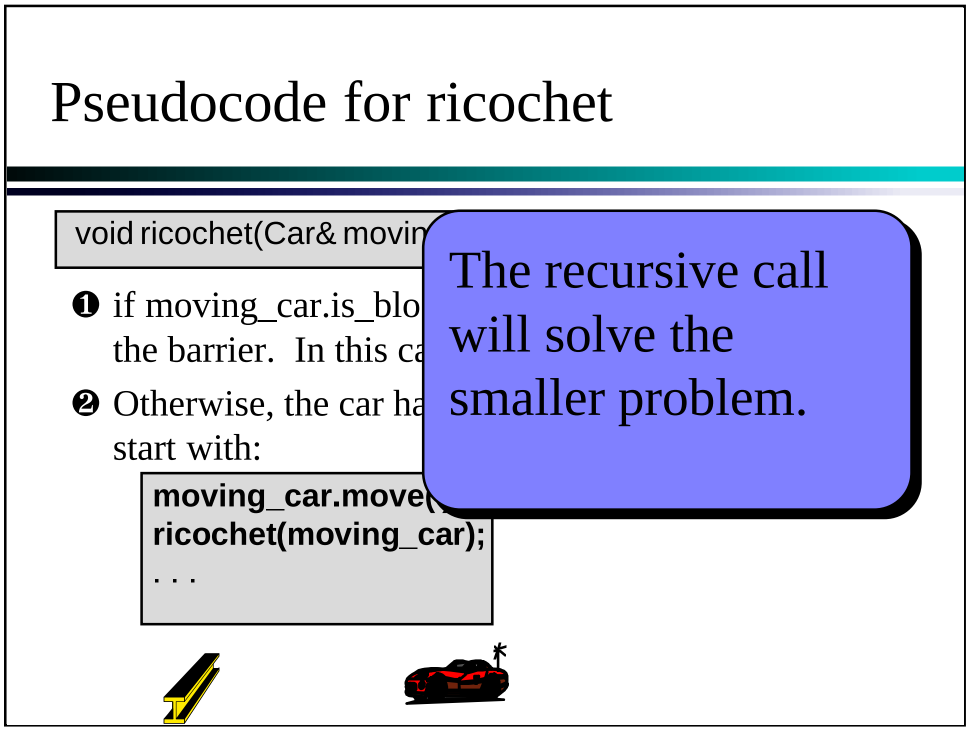
❶ if moving_car.is_blo
the barrier. In this ca

❷ Otherwise, the car ha
start with:

```
moving_car.move(
ricochet(moving_car);
. . .
```

The recursive call will solve the smaller problem.

This is still work being done in the recursive call.



Pseudocode for ricochet

```
void ricochet(Car& movin
```

❶ if moving_car.is_blo
the barrier. In this ca

❷ Otherwise, the car ha
start with:

```
moving_car.move(
ricochet(moving_car);
. . .
```

The recursive call will solve the smaller problem.

Still in that recursive call!

Pseudocode for ricochet

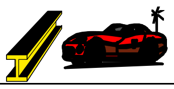
```
void ricochet(Car& movin
```

❶ if moving_car.is_blo
the barrier. In this ca

❷ Otherwise, the car ha
start with:

```
moving_car.move(
ricochet(moving_car);
. . .
```

The recursive call will solve the smaller problem.



The work of the recursive call has taken the car to the barrier...

Pseudocode for ricochet

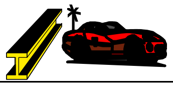
```
void ricochet(Car& movin
```

❶ if moving_car.is_blo
the barrier. In this ca

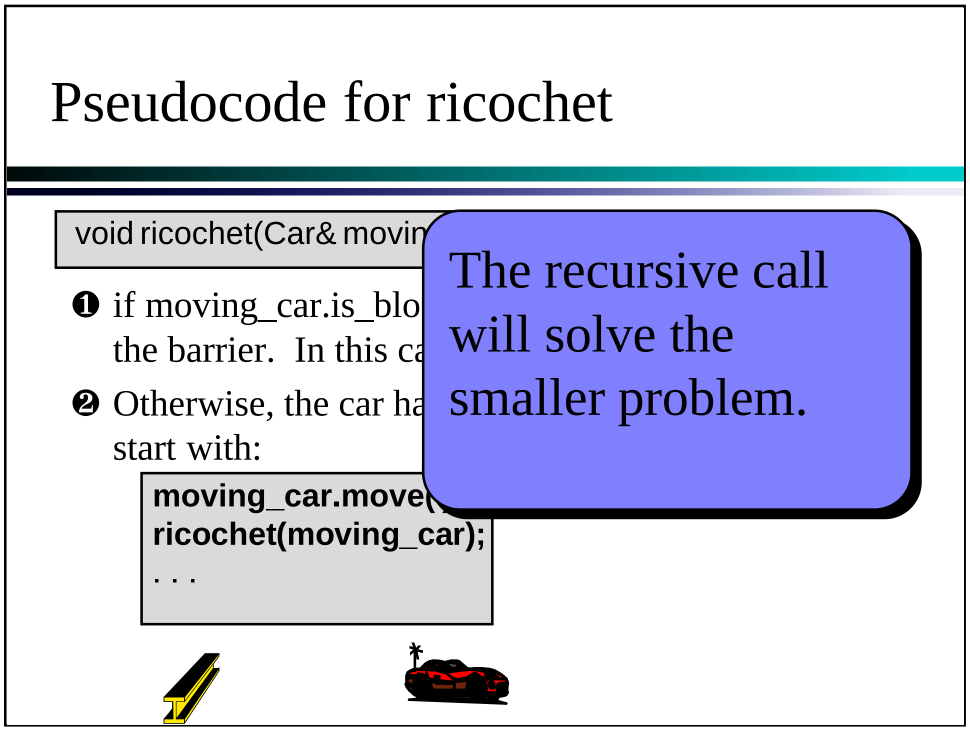
❷ Otherwise, the car ha
start with:

```
moving_car.move(
ricochet(moving_car);
. . .
```

The recursive call will solve the smaller problem.



...and turned the car around.



Pseudocode for ricochet

```
void ricochet(Car& movin
```

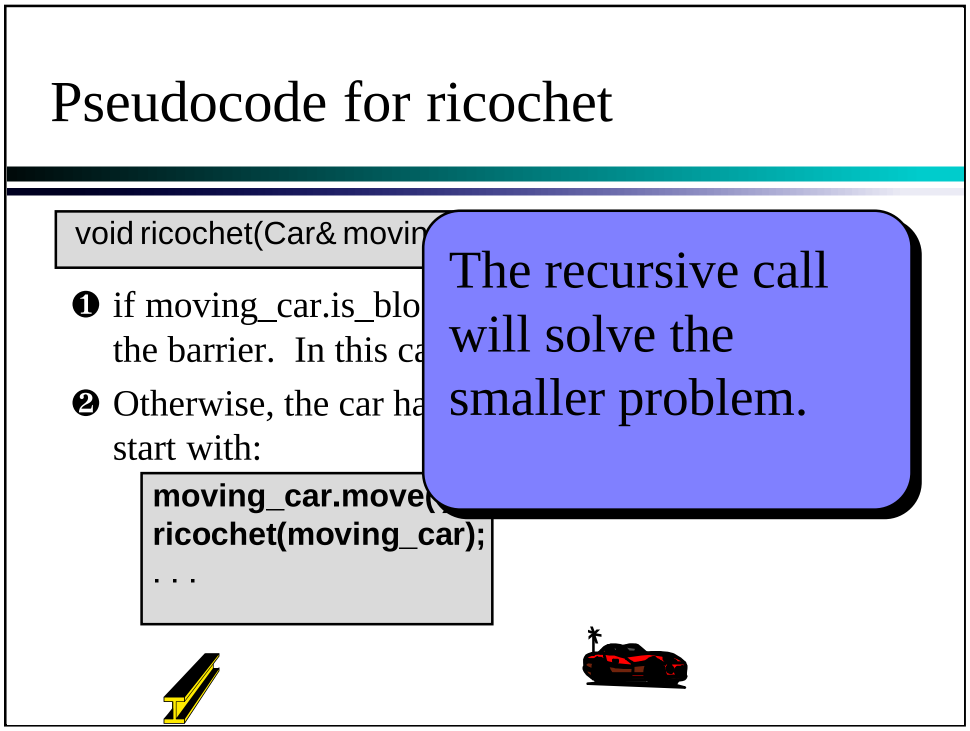
❶ if moving_car.is_blo
the barrier. In this ca

❷ Otherwise, the car ha
start with:

```
moving_car.move(
ricochet(moving_car);
. . .
```

The recursive call will solve the smaller problem.

The recursive call is still working to solve the smaller problem...



Pseudocode for ricochet

```
void ricochet(Car& movin
```

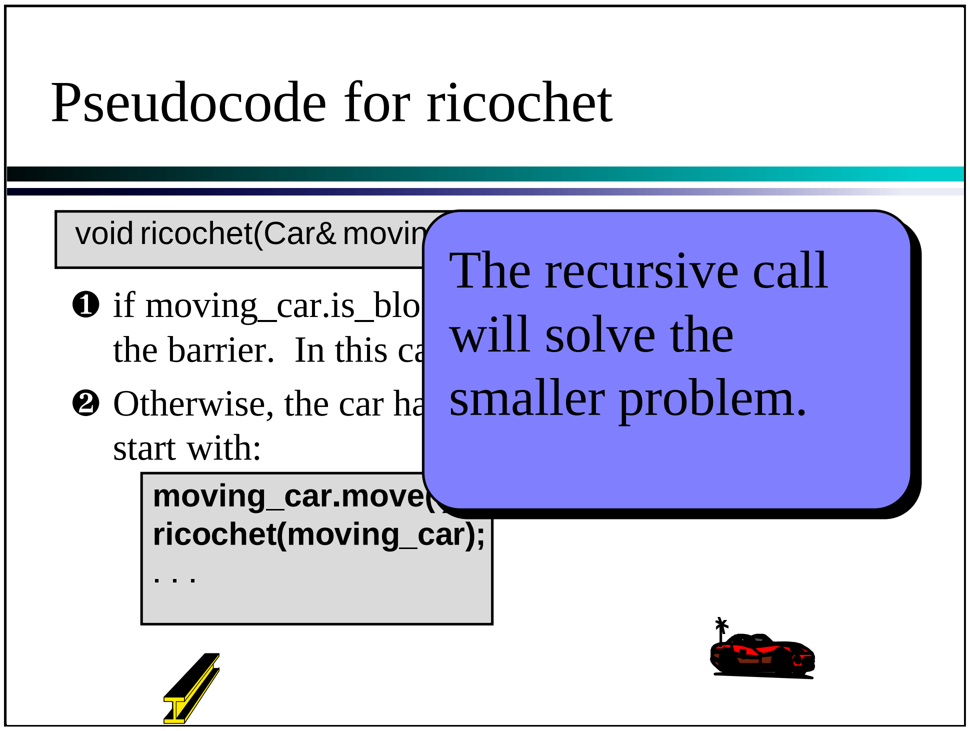
❶ if moving_car.is_blo
the barrier. In this ca

❷ Otherwise, the car ha
start with:

```
moving_car.move(
ricochet(moving_car);
. . .
```

The recursive call will solve the smaller problem.

...eventually the recursive call will bring the car back...



Pseudocode for ricochet

```
void ricochet(Car& movin
```

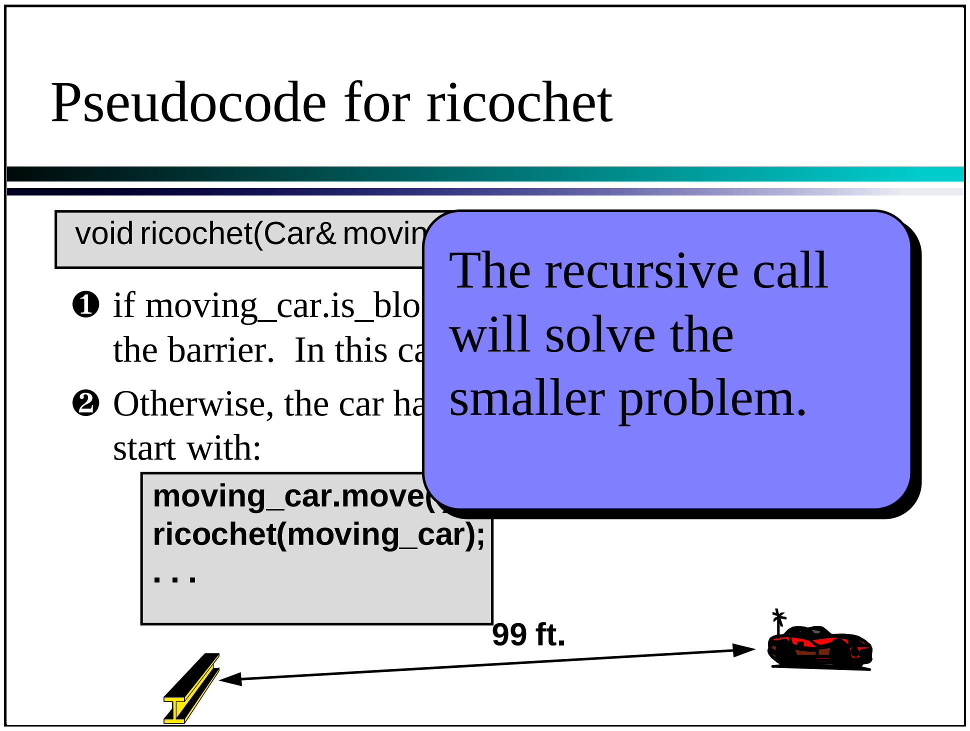
❶ if moving_car.is_blo
the barrier. In this ca

❷ Otherwise, the car ha
start with:

```
moving_car.move(
ricochet(moving_car);
. . .
```

The recursive call will solve the smaller problem.

...to the position where the recursive call was made.



Pseudocode for ricochet

```
void ricochet(Car& movin
```

❶ if moving_car.is_blo
the barrier. In this ca

❷ Otherwise, the car ha
start with:

```
moving_car.move(
ricochet(moving_car);
. . .
```

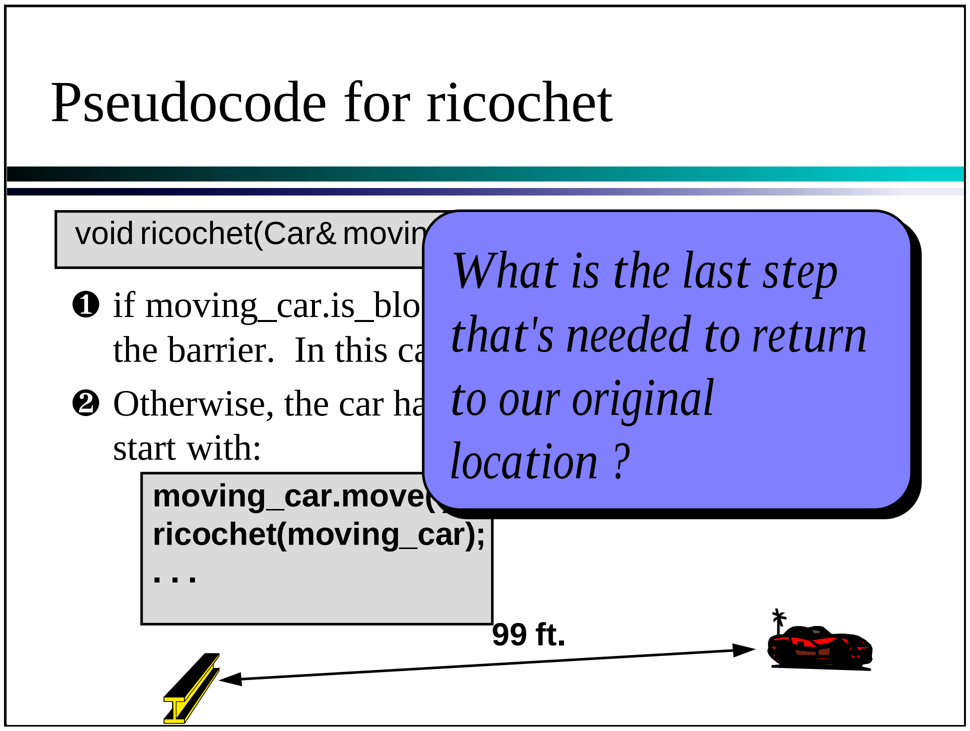
The recursive call will solve the smaller problem.

At last! The recursive call has finished its work, solving the smaller problem. In the solution of the smaller problem, the recursive call moved the car forward to the barrier, turned the car around, and moved the car back to the starting position of the smaller problem, but facing the opposite direction

Another key point: You don't need to know all the details of how that recursive call worked. Just have confidence that the recursive call will solve the smaller problem. You can safely have this confidence provided that:

1. When the problem gets small enough, there won't be more recursive calls, and

2. Each recursive call makes the problem "smaller".



Pseudocode for ricochet

```
void ricochet(Car& movin
```

❶ if moving_car.is_blo
the barrier. In this ca

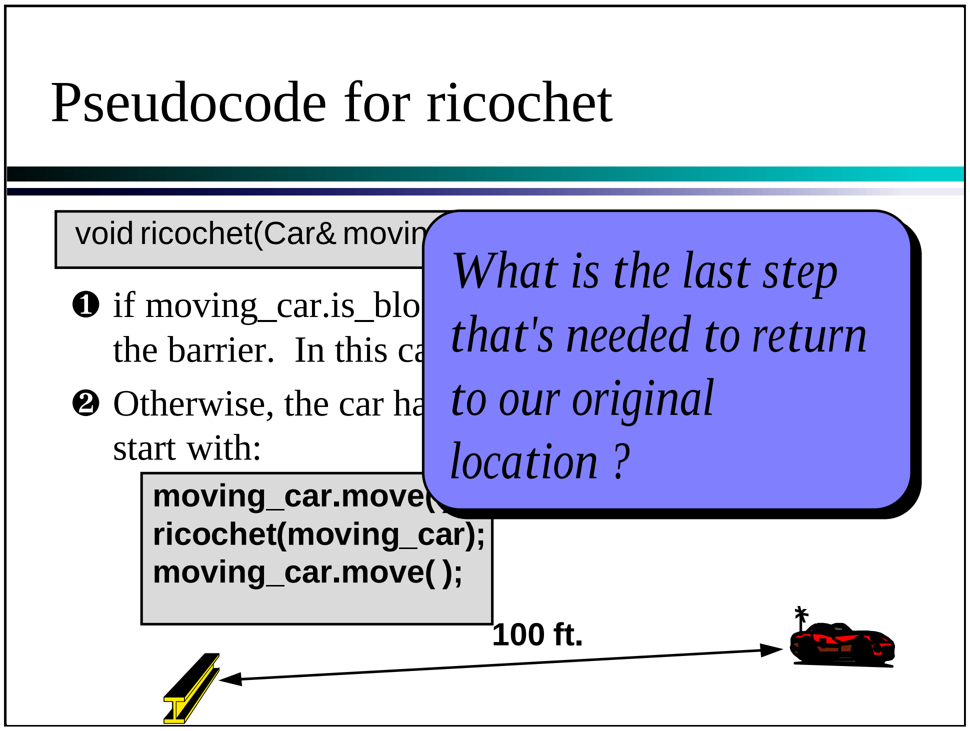
❷ Otherwise, the car ha
start with:

```
moving_car.move(
ricochet(moving_car);
. . .
```

What is the last step that's needed to return to our original location ?

After the recursive call solves the smaller problem, there is one more step that must be taken to solve the original problem.

Question: What is that remaining step?



Pseudocode for ricochet

```
void ricochet(Car& movin
```

❶ if moving_car.is_blo
the barrier. In this ca

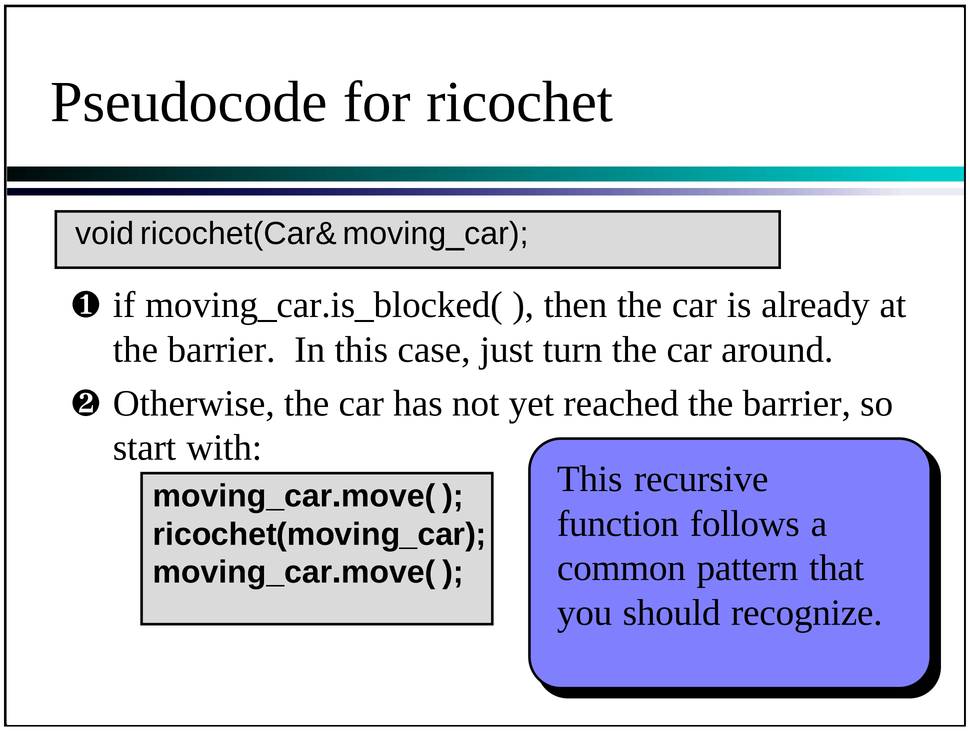
❷ Otherwise, the car ha
start with:

```
moving_car.move(
ricochet(moving_car);
moving_car.move( );
```

What is the last step that's needed to return to our original location ?

100 ft.

Answer: The car must be moved one more step to bring it back to its original position.



Pseudocode for ricochet

```
void ricochet(Car& moving_car);
```

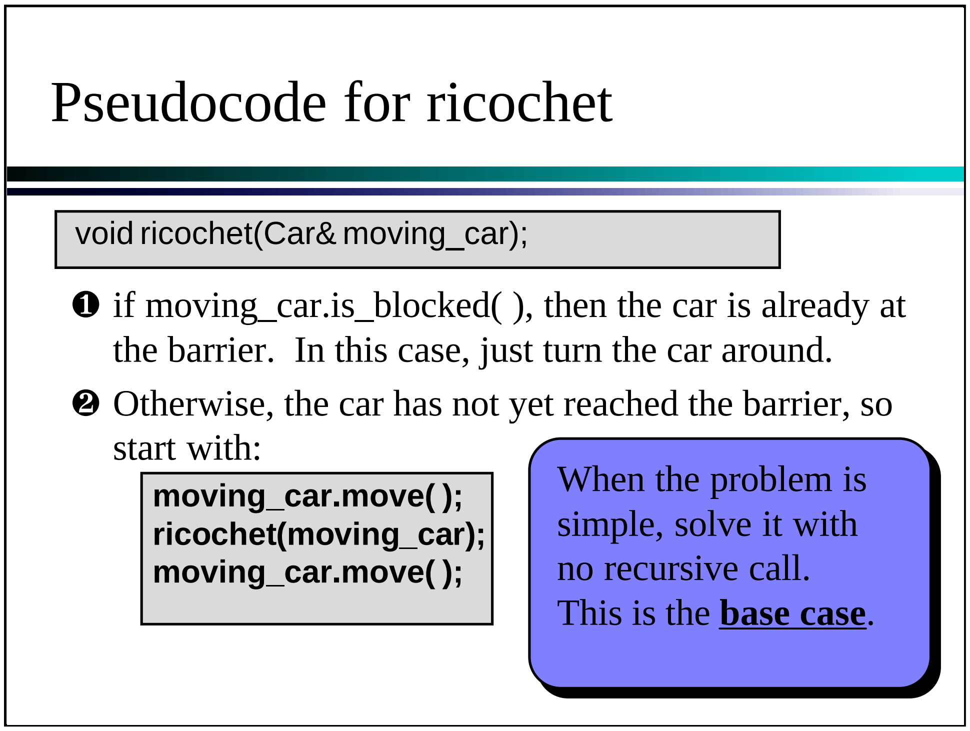
❶ if moving_car.is_blocked( ), then the car is already at the barrier. In this case, just turn the car around.

❷ Otherwise, the car has not yet reached the barrier, so start with:

```
moving_car.move( );
ricochet(moving_car);
moving_car.move( );
```

This recursive function follows a common pattern that you should recognize.

The pattern followed by this recursive function is a common one that you should be able to recognize and implement for similar problems.



Pseudocode for ricochet

```
void ricochet(Car& moving_car);
```

❶ if moving_car.is_blocked( ), then the car is already at the barrier. In this case, just turn the car around.

❷ Otherwise, the car has not yet reached the barrier, so start with:

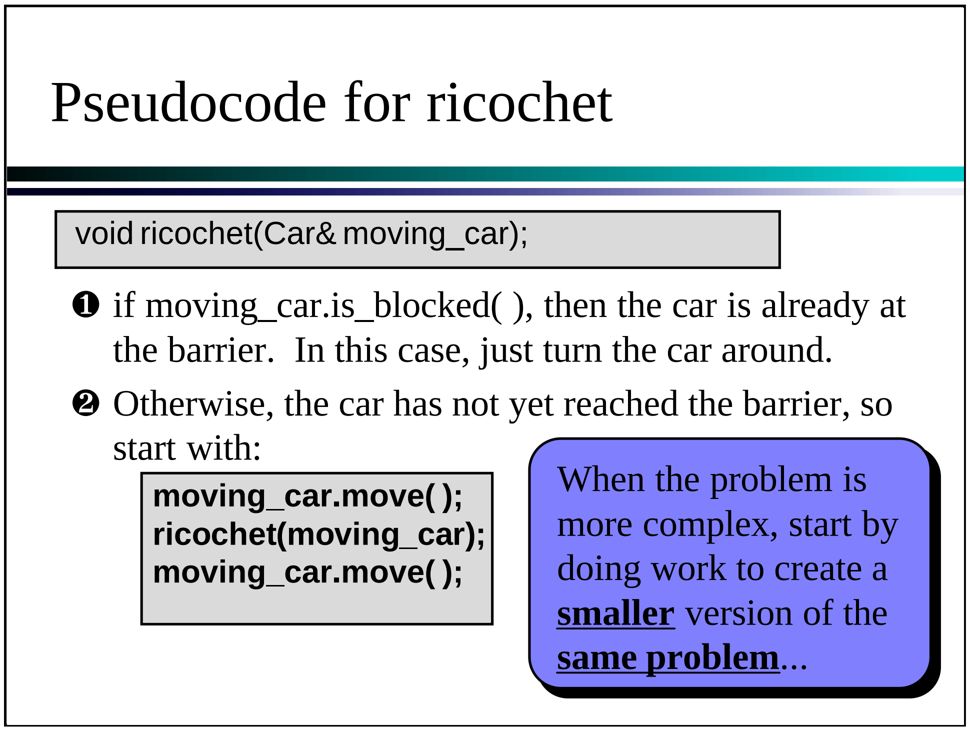
```
moving_car.move( );
ricochet(moving_car);
moving_car.move( );
```

When the problem is simple, solve it with no recursive call.
This is the **base case**.

The start of the pattern is called the base case. This is the case where the problem is so simple that it can be solved with no recursion.

Question: What do you suppose would happen if we forgot to include a base case?

Answer: In theory, the recursion would never end. Each recursive call would make another recursive call which would make another recursive call, and so on forever. In practice, each recursive call uses some of the computer's memory, and eventually you will run out of memory. At that point, a run-time error occurs stating "Run-time stack overflow" or perhaps "Stack-Heap collision." Both messages mean that you have run out of memory to make function calls.



Pseudocode for ricochet

```
void ricochet(Car& moving_car);
```

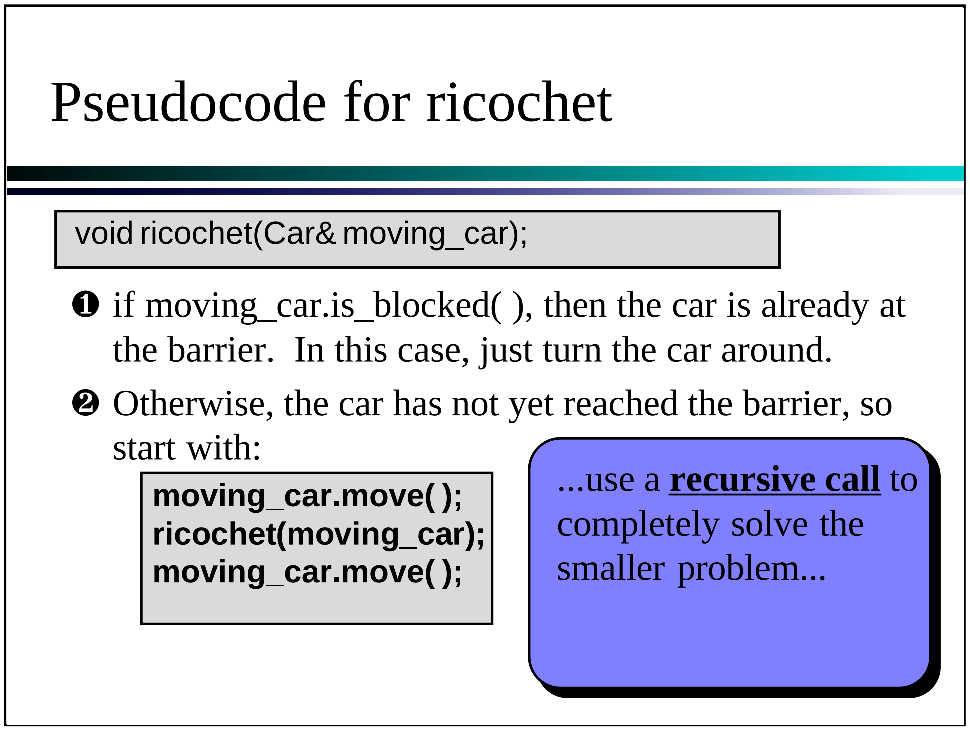
❶ if moving_car.is_blocked( ), then the car is already at the barrier. In this case, just turn the car around.

❷ Otherwise, the car has not yet reached the barrier, so start with:

```
moving_car.move( );
ricochet(moving_car);
moving_car.move( );
```

When the problem is more complex, start by doing work to create a **smaller** version of the **same problem**...

If the base case does not apply, then the algorithm will work by creating a smaller version of the original problem.



Pseudocode for ricochet

```
void ricochet(Car& moving_car);
```

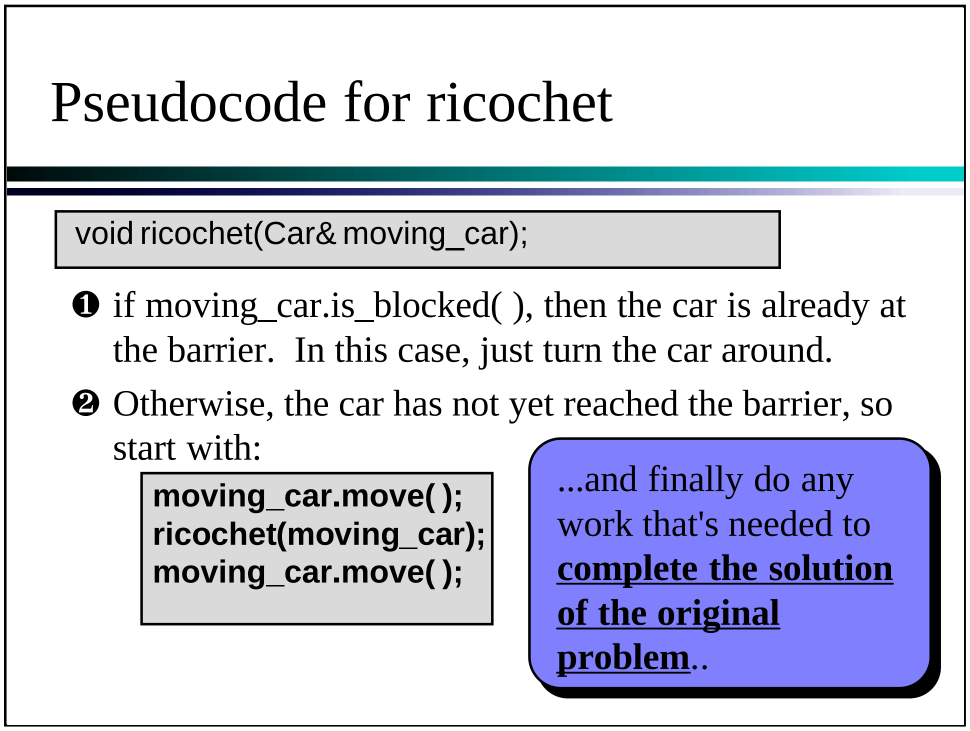
❶ if moving_car.is_blocked( ), then the car is already at the barrier. In this case, just turn the car around.

❷ Otherwise, the car has not yet reached the barrier, so start with:

```
moving_car.move( );
ricochet(moving_car);
moving_car.move( );
```

...use a **recursive call** to completely solve the smaller problem...

The smaller problem is solved with a recursive call.



Pseudocode for ricochet

```
void ricochet(Car& moving_car);
```

❶ if moving_car.is_blocked( ), then the car is already at the barrier. In this case, just turn the car around.

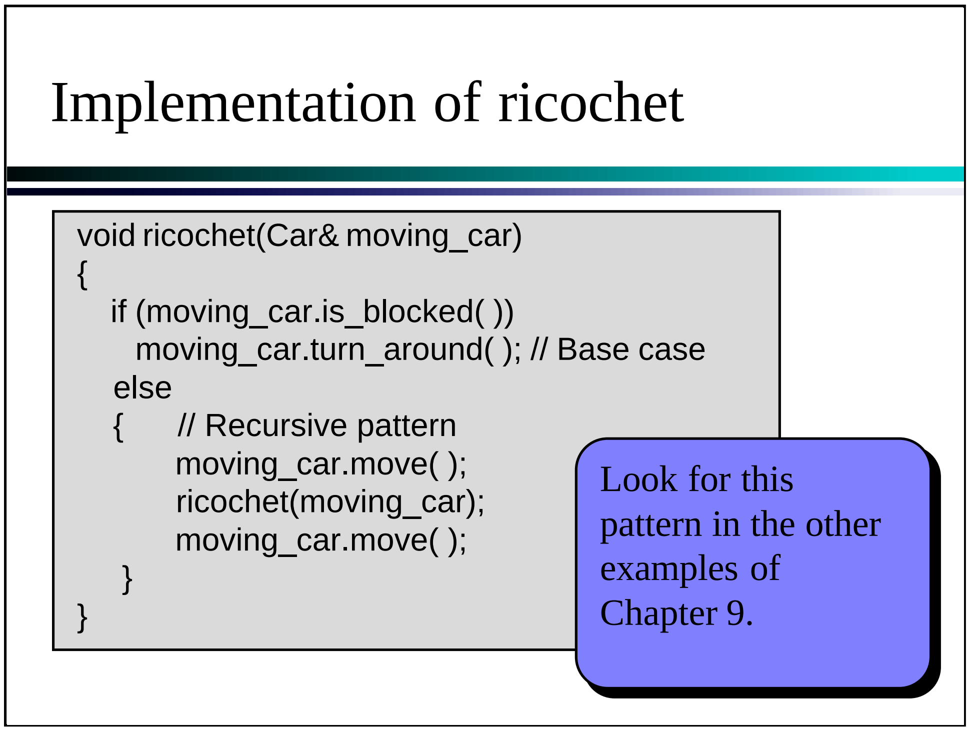
❷ Otherwise, the car has not yet reached the barrier, so start with:

```
moving_car.move( );
ricochet(moving_car);
moving_car.move( );
```

...and finally do any work that's needed to **complete the solution of the original problem**..

Generally a bit of extra work is done before the recursive call, in order to create the smaller problem. Work is often needed after the recursive call too, to finish solving the larger problem.

Programmer's Hint: If you find that no work is needed after the recursive call, then you have a simple kind of recursion called "tail recursion". The word "tail" refers to the fact that the recursive call is the final thing the algorithm does. Tail recursion is nearly always a bad idea. The reason is that tail recursive algorithms can generally be written much simpler with a loop and no recursive call.

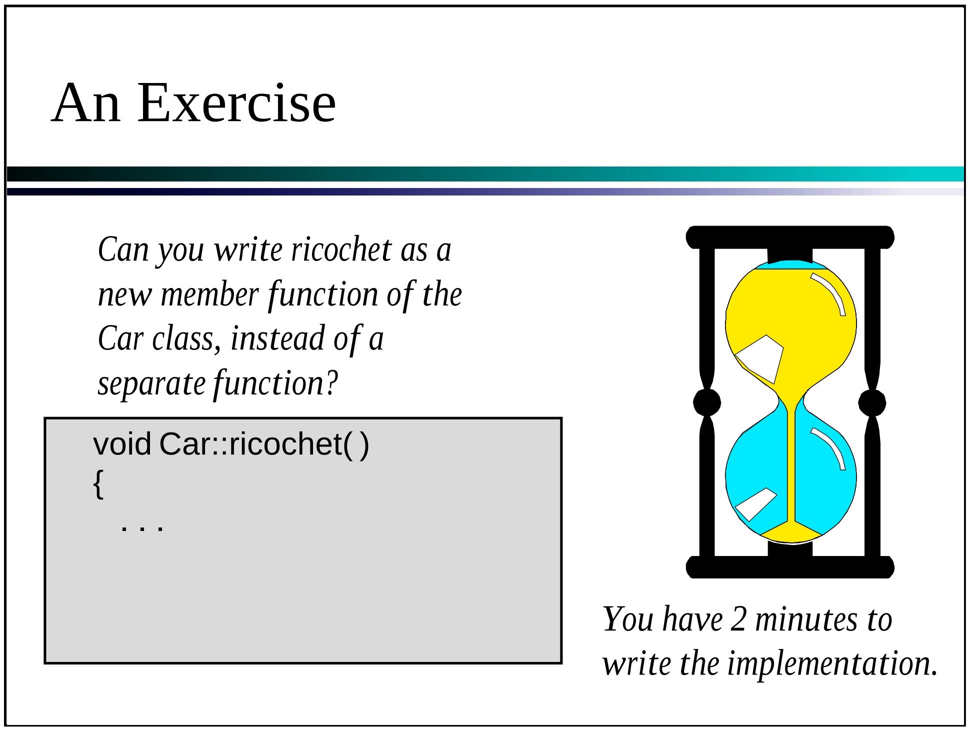


Implementation of ricochet

```
void ricochet(Car& moving_car)
{
   if (moving_car.is_blocked( ))
     moving_car.turn_around( ); // Base case
   else
   {      // Recursive pattern
          moving_car.move( );
          ricochet(moving_car);
          moving_car.move( );
    }
}
```

Look for this pattern in the other examples of Chapter 9.

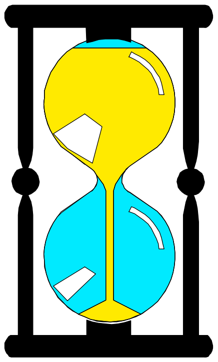
You'll find this same pattern in other problems of Chapter 9, such as the recursive functions to write the digits of a number vertically.



An Exercise

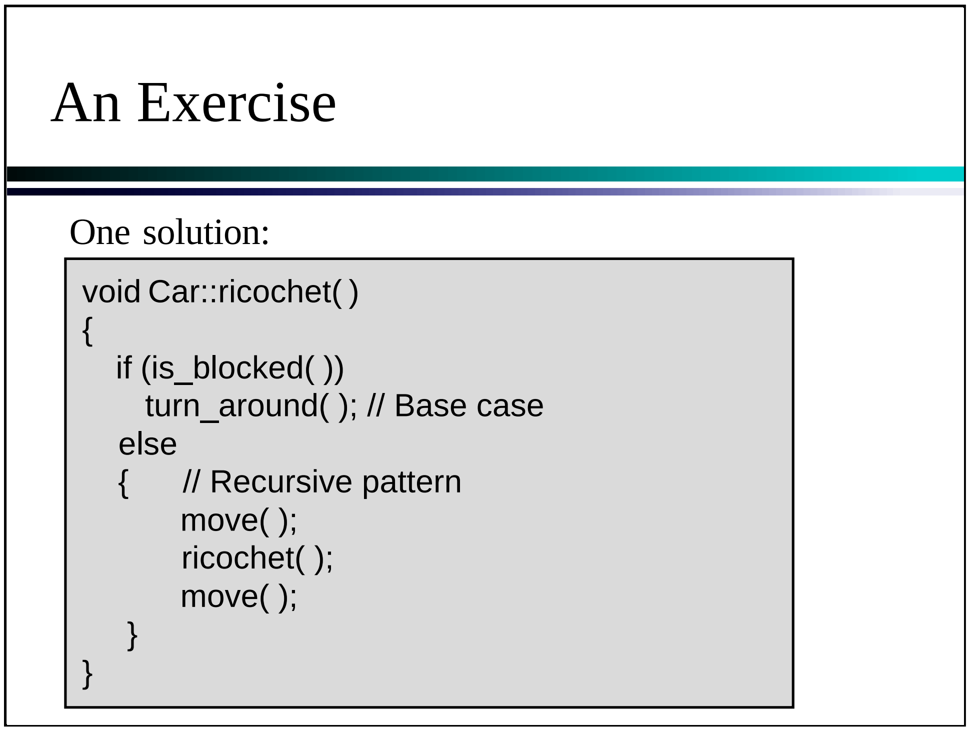
Can you write ricochet as a new member function of the Car class, instead of a separate function?

```
void Car::ricochet( )
{
   . . .
```



You have 2 minutes to write the implementation.

Time for another quiz . . .



An Exercise

One solution:

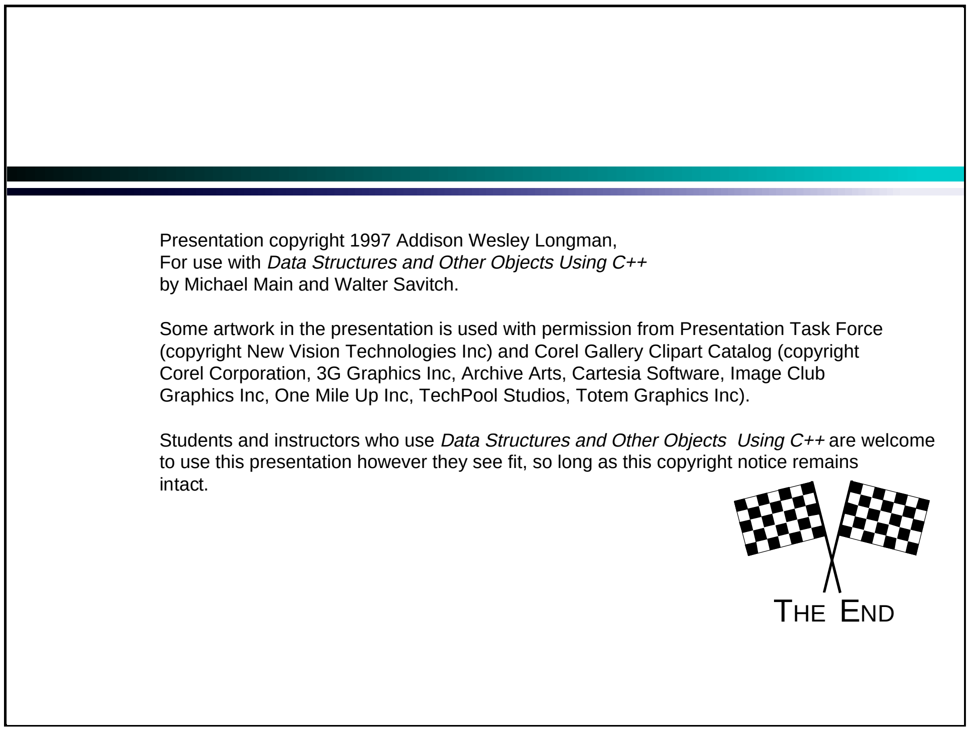
```
void Car::ricochet( )
{
   if (is_blocked( ))
      turn_around( ); // Base case
   else
   {     // Recursive pattern
         move( );
         ricochet( );
         move( );
    }
}
```

Two key points:

1. The recursive member function does not need a parameter, since it must be activated by a particular car. For example, if you have a car named racer, then you can activate racer's ricochet function with:

```
racer.ricochet( );
```

2. The recursive member function activates itself. Do you see where this happens?



THE END

Feel free to send your ideas to:

Michael Main

main@colorado.edu