CSC212
Data Structure

COMPUTER SCIENCE
CITY COLLEGE OF NEW YORK

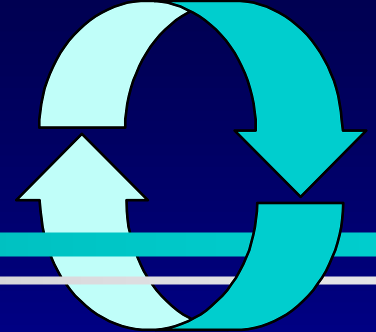
Lecture 11
Recursive Thinking

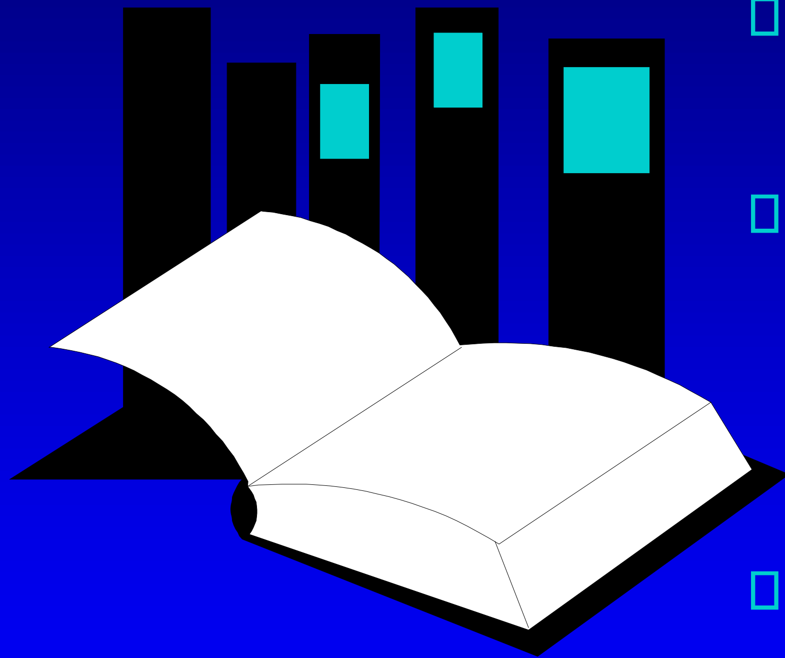
Instructor: George Wolberg
Department of Computer Science
City College of New York

Outline of This Lecture

- Start with an Example of Recursion
  - “racing car” – not in the textbook
  - using slides (provided by the authors)
- Recursive Thinking: General Form
- Tracing Recursive Calls
  - using blackboard to show the concepts
- A Closer Look at Recursion
  - activation record and runtime stack

Recursive Thinking



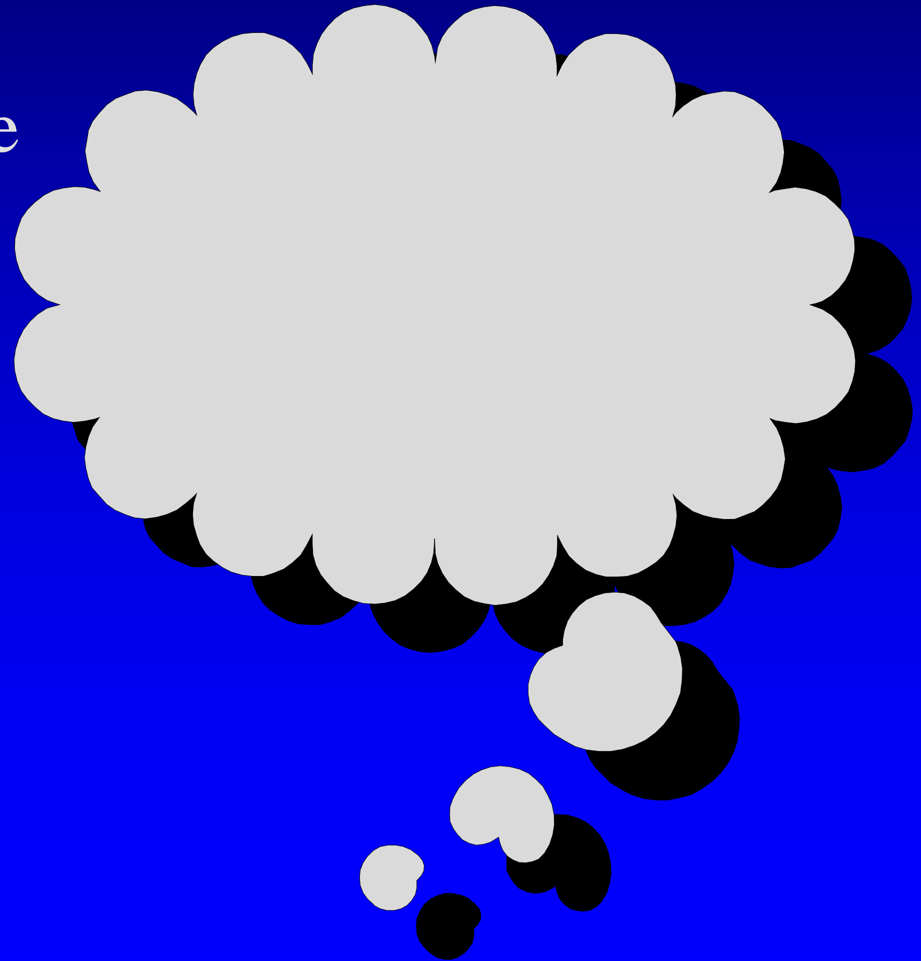


- Chapter 9 introduces the technique of recursive programming.
- As you have seen, recursive programming involves spotting smaller occurrences of a problem within the problem itself.
- This presentation gives an additional example, which is not in the book.

Data Structures and Other Objects Using C++

A Car Object

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



A Car Object

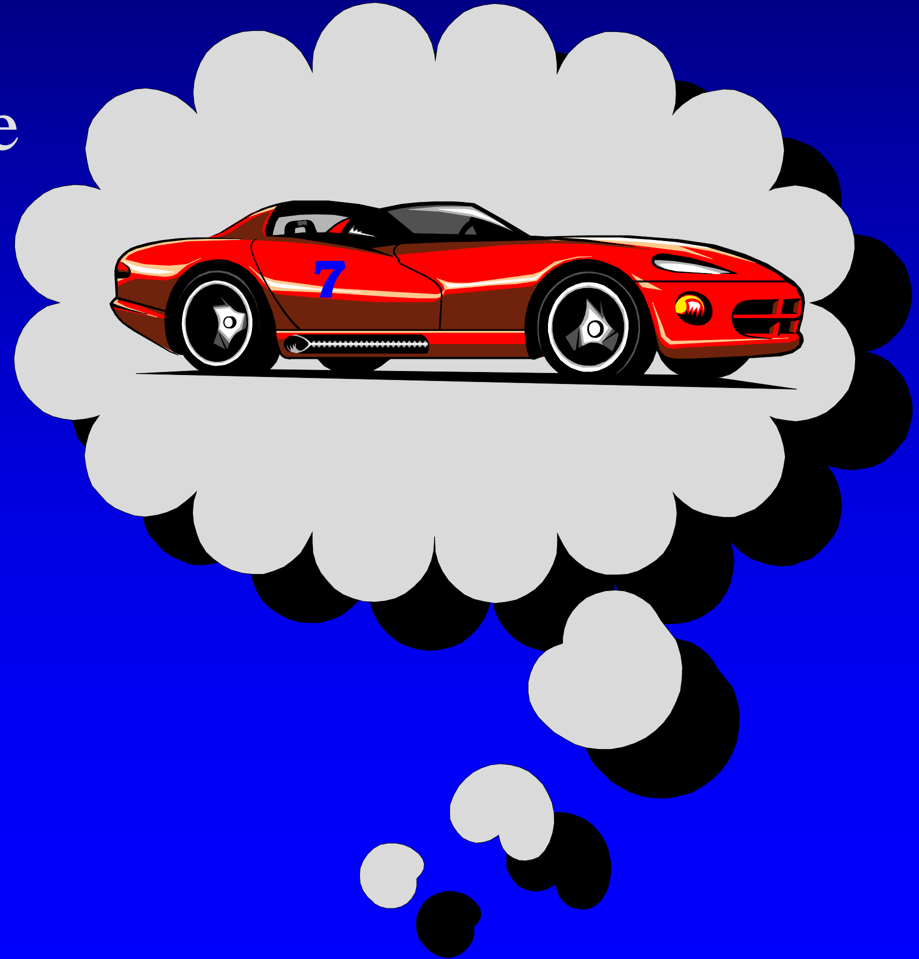
- To start the example, think about your favorite family car

A Car Object

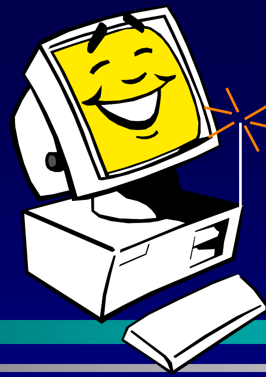
- To start the example, think about your favorite family car

A Car Object

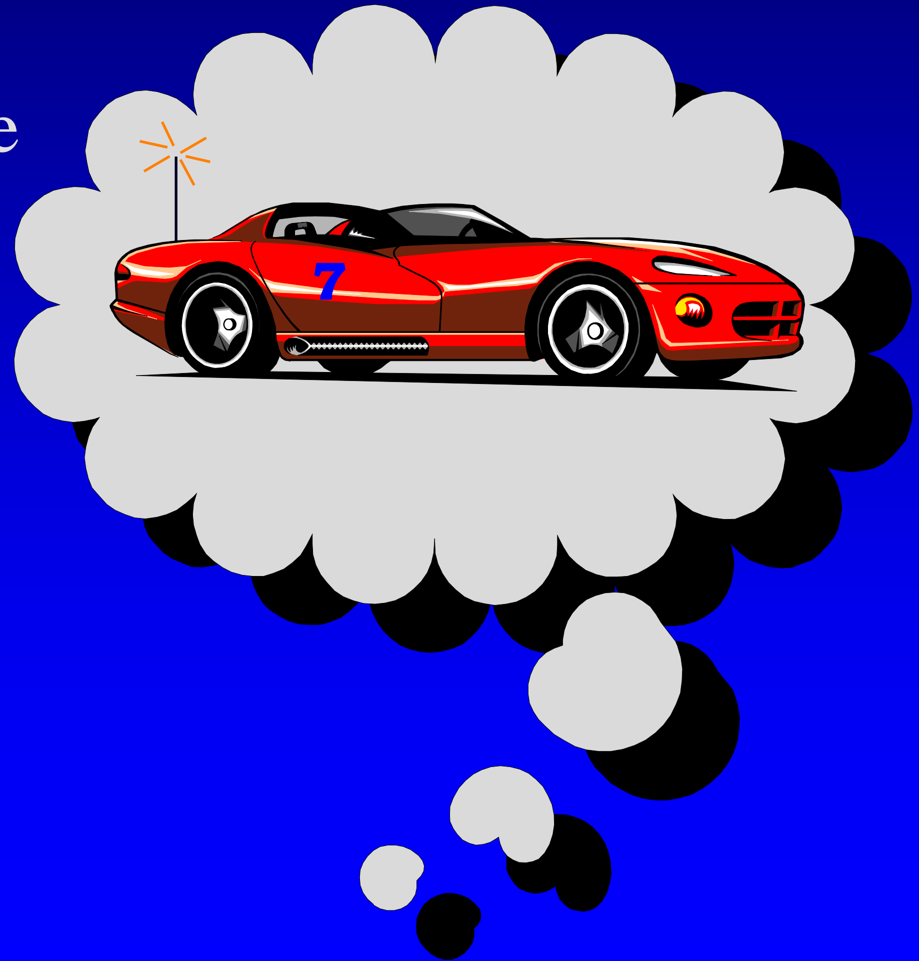
- To start the example, think about your favorite family car



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



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

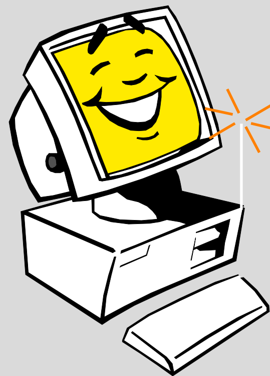
```
class Car
{
public:
        . . .
};
```

Member Functions for the Car Class

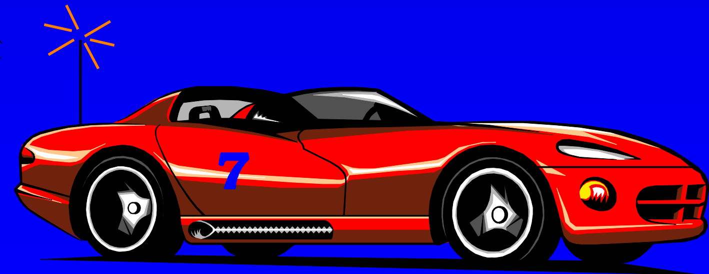
```
class Car
{
public:
        Car(int car_number);
        void move( );
        void turn_around( );
        bool is_blocked( );
private:
        { We don't need to know the private fields! }
         . . .
};
```

The Constructor

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

   . . .
```



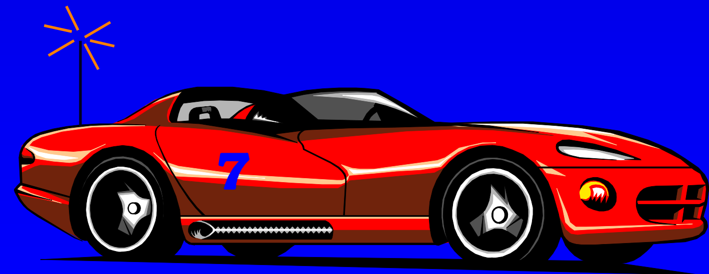
When we declare a Car and activate the constructor, the computer makes a radio link with a car that has a particular number.



The turn_around Function

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

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

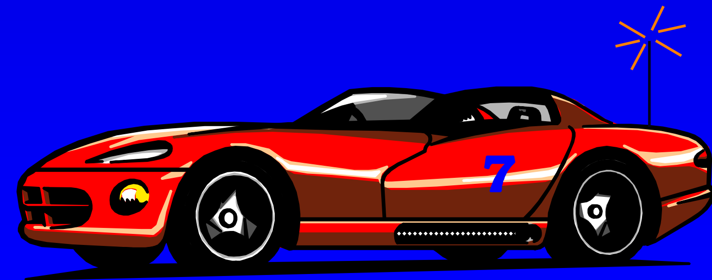
When we activate turn_around, the computer signals the car to turn 180 degrees.



The turn_around Function

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

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

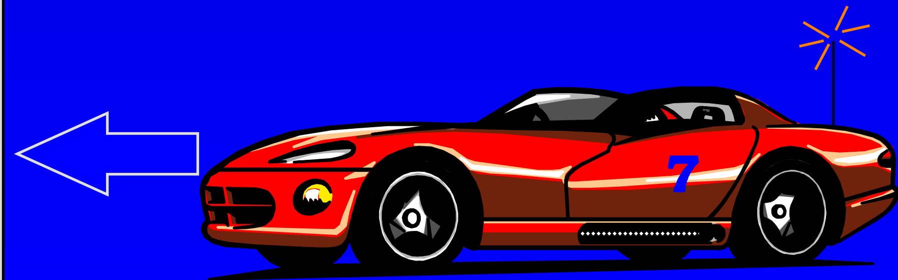
When we activate turn_around, the computer signals the car to turn 180 degrees.



The move Function

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

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

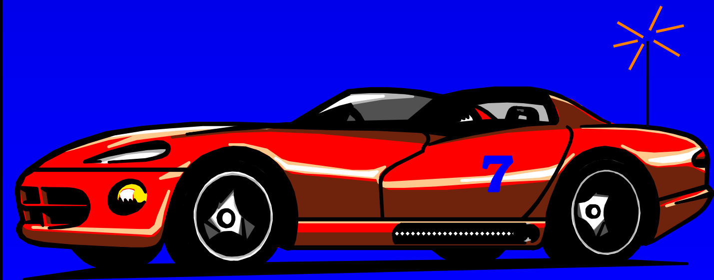
When we activate move, the computer signals the car to move forward one foot.



The move Function

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

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

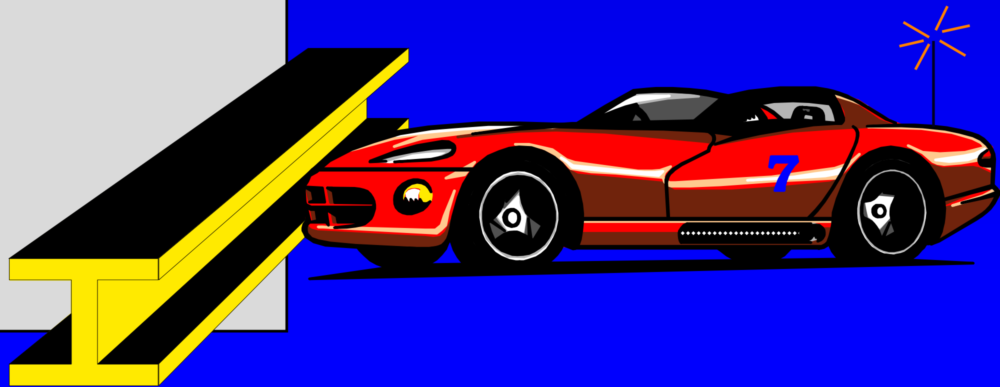
When we activate move, the computer signals the car to move forward one foot.



The is_blocked( ) Function

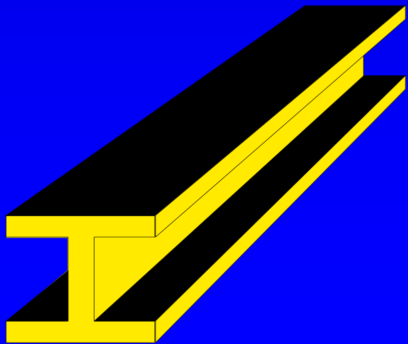
```
int main( )
{
   Car racer(7);

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

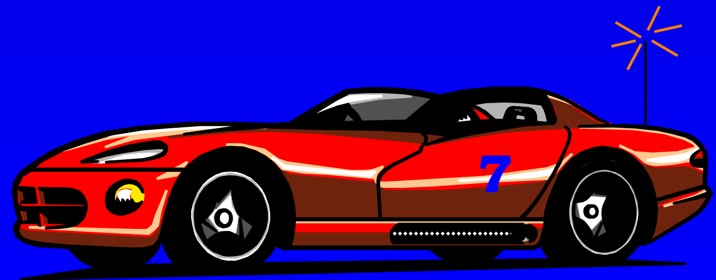
The is_blocked member function detects barriers.



Your Mission

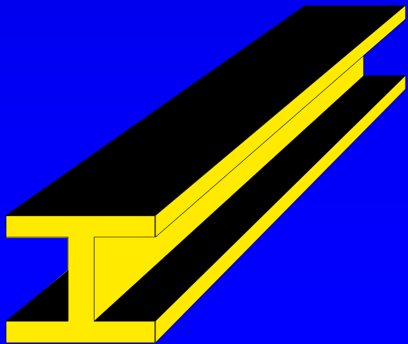
- Write a function which will move a Car forward until it reaches a barrier...

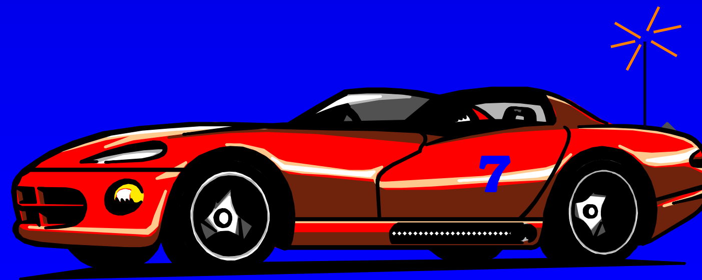




Your Mission

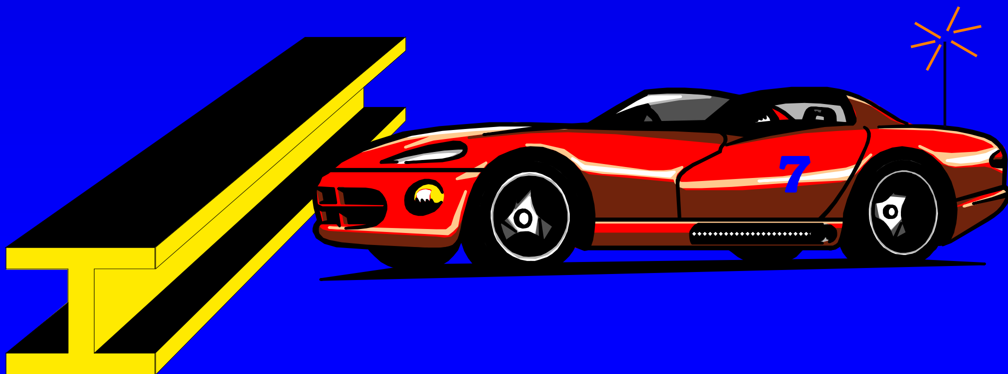
- Write a function which will move a Car forward until it reaches a barrier...





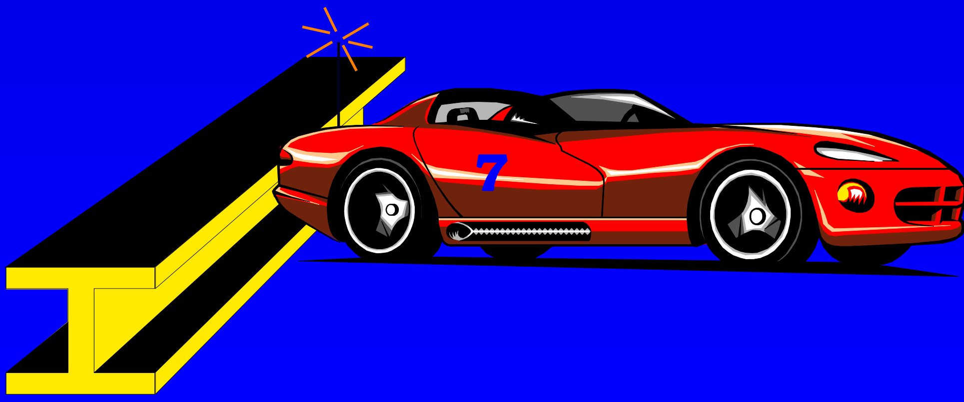
Your Mission

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



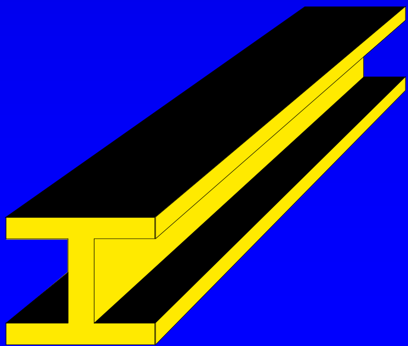
Your Mission

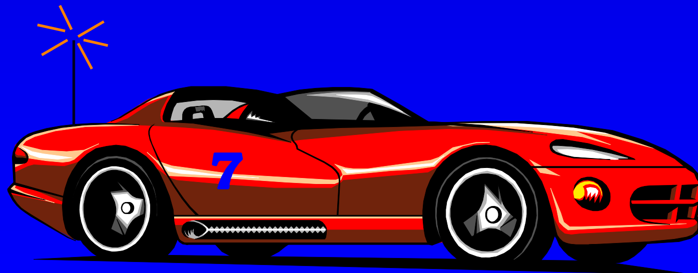
- Write a function which will move a Car forward until it reaches a barrier...
- ...then the car is turned around...



Your Mission

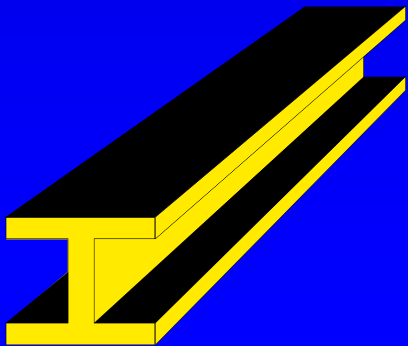
- Write a function which will move a Car forward until it reaches a barrier...
- ...then the car is turned around...
- ...and returned to its original location, facing the opposite way.

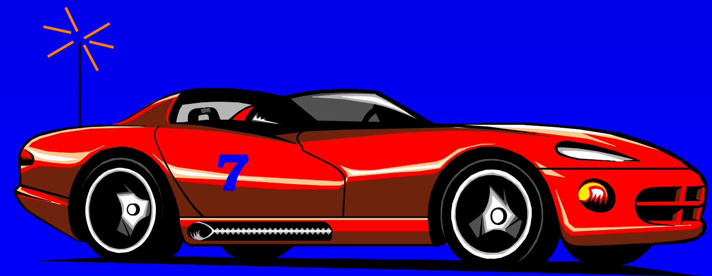




Your Mission

- 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.

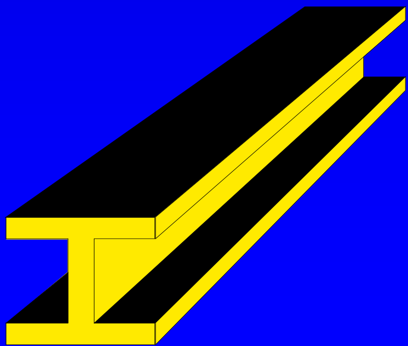


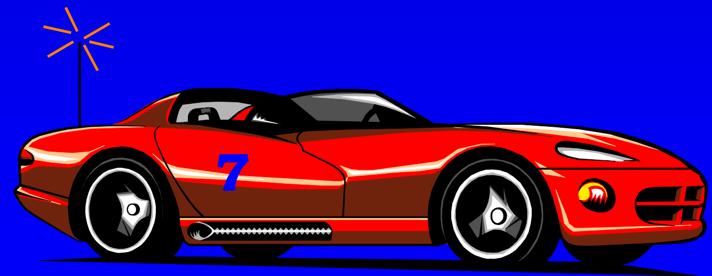


Your Mission

```
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.





Pseudocode for ricochet

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

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

Pseudocode for ricochet

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

- if moving_car.is_blocked( ), then the car is already at the barrier. In this case, just turn the car around.
- Otherwise, the car has not yet reached the barrier, so start with:

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

Pseudocode for ricochet

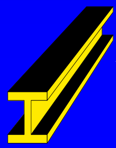
```
void ricochet(Car& movin
```

- if moving_car.is_blo the barrier. In this ca
- 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...

100 ft.



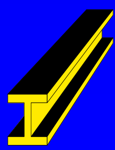
Pseudocode for ricochet

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

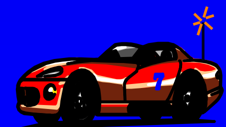
- if moving_car.is_blo
  the barrier. In this ca
- Otherwise, the car ha
  start with:

```
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.



99 ft.



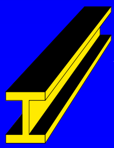
Pseudocode for ricochet

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

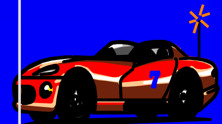
- if moving_car.is_blo
  the barrier. In this ca
- Otherwise, the car ha
  start with:

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

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



99 ft.



Pseudocode for ricochet

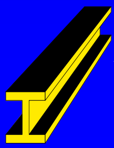
```
void ricochet(Car& movin
```

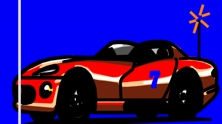
- if moving_car.is_blo
  the barrier. In this ca
- Otherwise, the car ha
  start with:

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

Make a recursive call to solve the smaller problem.

99 ft.





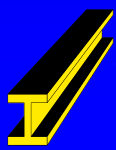
Pseudocode for ricochet

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

- if moving_car.is_blo
  the barrier. In this ca
- Otherwise, the car ha
  start with:

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

The recursive call will solve the smaller problem.



99 ft.

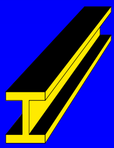
Pseudocode for ricochet

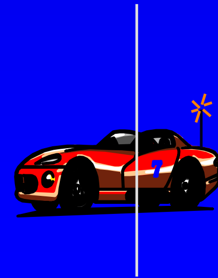
```
void ricochet(Car& movin
```

- if moving_car.is_blo
  the barrier. In this ca
- Otherwise, the car ha
  start with:

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

The recursive call will solve the smaller problem.





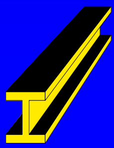
Pseudocode for ricochet

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

- if moving_car.is_blo
  the barrier. In this ca
- Otherwise, the car ha
  start with:

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

The recursive call will solve the smaller problem.



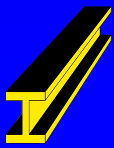
Pseudocode for ricochet

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

- if moving_car.is_blo the barrier. In this ca
- Otherwise, the car ha start with:

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

The recursive call will solve the smaller problem.



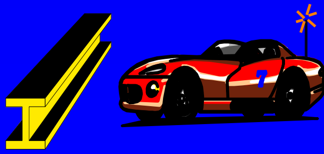
Pseudocode for ricochet

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

- if moving_car.is_blo
  the barrier. In this ca
- Otherwise, the car ha
  start with:

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

The recursive call will solve the smaller problem.



Pseudocode for ricochet

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

- if moving_car.is_blo
  the barrier. In this ca
- Otherwise, the car ha
  start with:

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

The recursive call will solve the smaller problem.

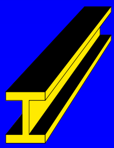
Pseudocode for ricochet

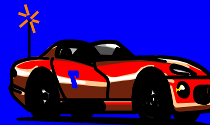
```
void ricochet(Car& movin
```

- if moving_car.is_blo the barrier. In this ca
- Otherwise, the car ha start with:

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

The recursive call will solve the smaller problem.





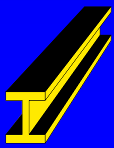
Pseudocode for ricochet

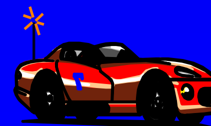
```
void ricochet(Car& movin
```

- if moving_car.is_blo
  the barrier. In this ca
- Otherwise, the car ha
  start with:

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

The recursive call will solve the smaller problem.





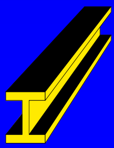
Pseudocode for ricochet

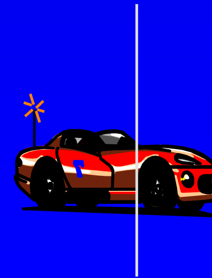
```
void ricochet(Car& movin
```

- if moving_car.is_blo
  the barrier. In this ca
- Otherwise, the car ha
  start with:

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

The recursive call will solve the smaller problem.





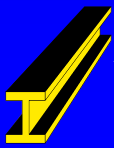
Pseudocode for ricochet

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

- if moving_car.is_blo
  the barrier. In this ca
- Otherwise, the car ha
  start with:

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

The recursive call will solve the smaller problem.



99 ft.

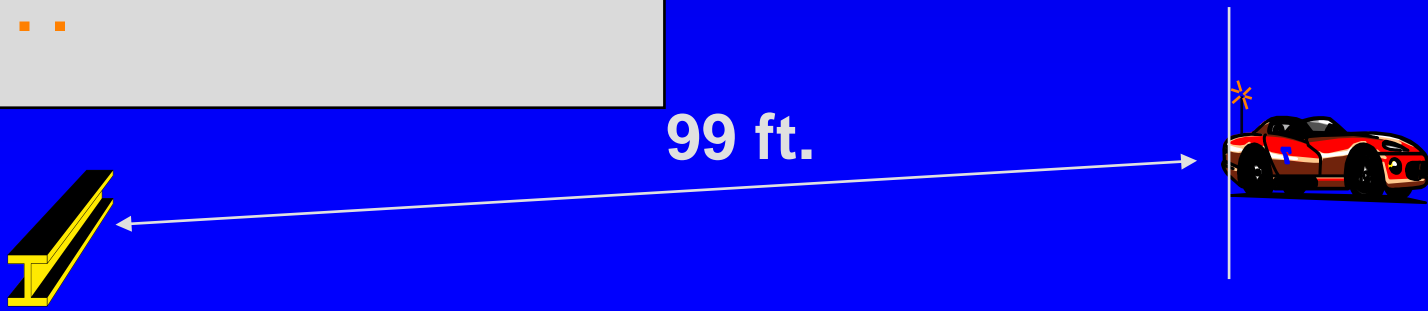
Pseudocode for ricochet

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

- if moving_car.is_blo
  the barrier. In this ca
- 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 ?



Pseudocode for ricochet

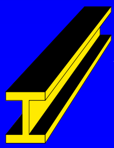
```
void ricochet(Car& movin
```

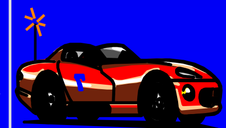
- if moving_car.is_blo
  the barrier. In this ca
- Otherwise, the car ha
  start with:

```
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.





Pseudocode for ricochet

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

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

Pseudocode for ricochet

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

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

Pseudocode for ricochet

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

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

Pseudocode for ricochet

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

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

Pseudocode for ricochet

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

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

Implementation of ricochet

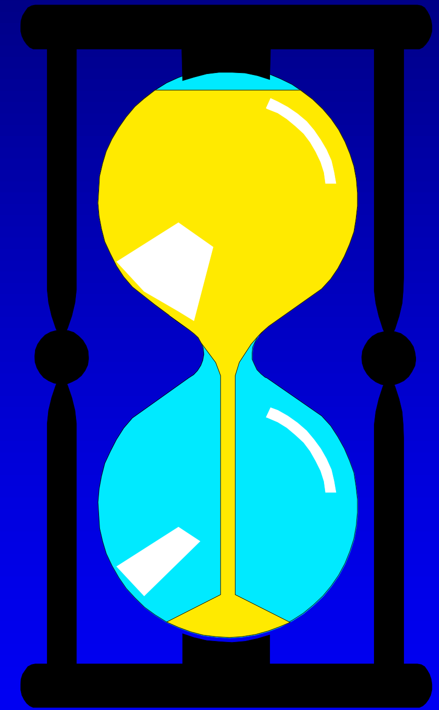
```
void ricochet(Car& moving_car)
{
    if (moving_car.is_blocked( ))
      moving_car.turn_around( ); // Base case
    else
    {      // Recursive pattern
           moving_car.move( );
           ricochet(moving_car);
           moving_car.move( );
     }
}
```

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

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.

An Exercise

One solution:

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

Recursive Thinking: General Form

- Recursive Calls
  - Suppose a problem has one or more cases in which some of the subtasks are simpler versions of the original problem. These subtasks can be solved by recursive calls
- Stopping Cases /Base Cases
  - A function that makes recursive calls must have one or more cases in which the entire computation is fulfilled without recursion. These cases are called stopping cases or base cases

Tracing Recursive Calls: Ricochet

Do it by hand if car is 4 feet away from the barrier

```
void Car::ricochet( )
{
   if (is_blocked( ))
A.       turn_around( ); // Base case
   else
   {     // Recursive pattern
B.          move( );
C.          ricochet( );
D.          move( );
E }
}
```

A Close Look at Ricochet Recursion

- The recursive case and the stopping case
- Activation record
  - The return location only in this example – other information is kept in the object racer
- The running stack
  - The collection of the activation records is stored in a stack data structure

Example 2: Write Number Vertically

- Task
  - Write a non-negative integer to the screen with its decimal digits stacked vertically
  - for example:

Input → Output:

1234

1

2

3

4

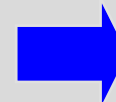
A possible function

Write an integer number vertically

```
void write_vertical (unsigned int number)
// precondition: number >=0
// Postcondition: The digits of number have been written, stacked vertically.
{  assert(number>=0);
    do
   {

        cout << number % 10 << endl;        // Write a digit
         number = number / 10;
    } while (number !=0);
}
```

Input

1234



Output:

4

3

2

1

Approach 1: using a stack

Write an integer number vertically

```
void stack_write_vertical (unsigned int number)
// Postcondition: The digits of number have been written, stacked vertically.
{
  stack<int> s;
   do
   {
        s.push(number % 10);        // push a digit in the stack
        number = number / 10;
    } while (number !=0);
   while (!(s.empty()))
   {
        cout << s.top()<< endl;        //print a digit from the stack
        s.pop();
    }
}
```

Approach 2: Using Recursion

Write an integer number vertically

```
void recursive_write_vertical(unsigned int number)
// Postcondition: The digits of number have been written, stacked vertically.
{
    if (number < 10)  // stopping case
       cout << number << endl; // Write the one digit
    else // including recursive calls
    {
       recursive_write_vertical(number/10);          // Write all but the last digit
       cout << number % 10 << endl;        // Write the last digit
    }
}
```

Tracing Recursive Calls

Write an integer number vertically

```
void recursive_write_vertical_2(unsigned int number)
// Postcondition: The digits of number have been written, stacked vertically.
{
   if (number < 10)  // stopping case
A     cout << number << endl; // Write the one digit
   else // including recursive calls
   {
B      recursive_write_vertical(number/10);         // Write all but the last digit
C      cout << number % 10 << endl;       // Write the last digit
D  }
}
```

A Closer Look at the Recursion

- Recursive Function
  - Recursive calls
  - Stopping (Base) cases
- Run-time Stack
  - the collection of activation records is stored in the stack
- Activation Record - a special memory block including
  - return location of a function call
  - values of the formal parameters and local variables

Recursive Thinking: General Form

- Recursive Calls
  - Suppose a problem has one or more cases in which some of the subtasks are simpler versions of the original problem. These subtasks can be solved by recursive calls
- Stopping Cases /Base Cases
  - A function that makes recursive calls must have one or more cases in which the entire computation is fulfilled without recursion. These cases are called stopping cases or base cases

Self-Tests and More Complicated Examples

- An Extension of write_vertical (page 436)
  - handles all integers including negative ones
  - Hints: you can have more than one recursive calls or stopping cases in your recursive function
- Homework
  - Reading: Section 9.1
  - Self-Test: Exercises 1-8
  - Advanced Reading: Section 9.2
  - Assignment 5 online

super_write_vertical

Write any integer number vertically

```
void super_write_vertical(int number)
// Postcondition: The digits of the number have been written, stacked vertically.
// If number is negative, then a negative sign appears on top.
// Library facilities used: iostream.h, math.h
{
    if (number < 0)
    {
        cout << '-' << endl;                // print a negative sign
        super_write_vertical(abs(number));  // abs computes absolute value
        // This is Spot #1 referred to in the text.
    }
    else if (number < 10)
        cout << number << endl;             // Write the one digit
    else
    {
        super_write_vertical(number/10);    // Write all but the last digit
        // This is Spot #2 referred to in the text.
        cout << number % 10 << endl;        // Write the last digit
    }
}
```