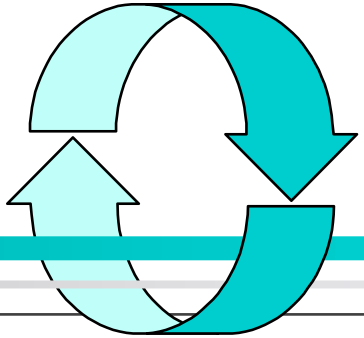
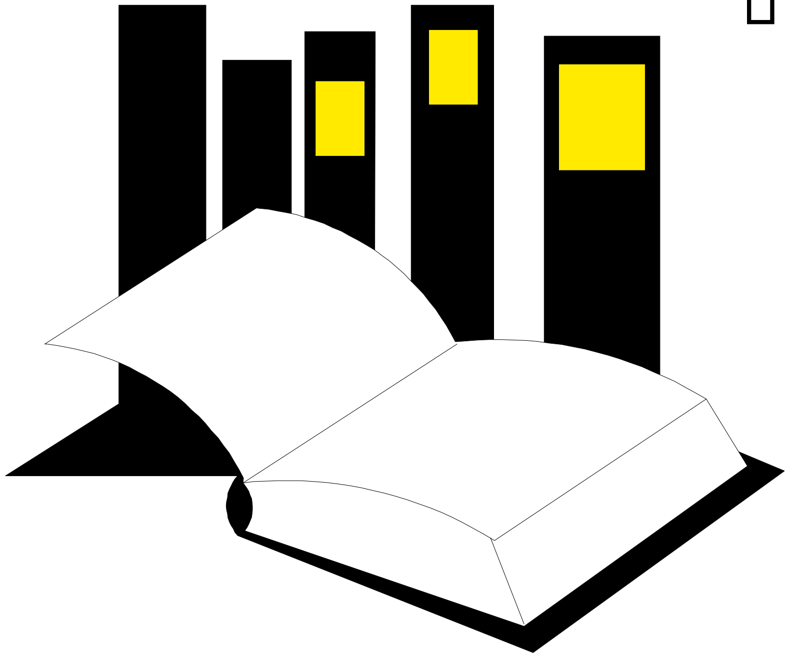
DATA STRUCTURES AND ALGORITHMS

DR SAMABIA TEHSIN

BS (AI)

Recursive Thinking





- Recursive programming involves spotting smaller occurrences of a problem within the problem itself.

Overview

Recursion: a definition in terms of itself.

Recursion in algorithms:

Natural approach to **some** (not all) problems

A *recursive algorithm* uses itself to solve one or more smaller identical problems

Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first

Recursion is a technique that solves a problem by solving a smaller problem of the same type

Recursive Methods Must Eventually Terminate

A recursive method must have

at least one base, or stopping, case.

A base case does not execute a recursive call
- stops the recursion

Each successive call to itself must be a "smaller version of itself”
- an argument that describes a smaller problem
- a base case is eventually reached

Key Components of a Recursive Algorithm Design

1. What is a smaller ***identical*** problem(s)?
   - Decomposition
2. How are the answers to smaller problems combined to form the answer to the larger problem?
   - Composition
3. Which is the smallest problem that can be solved easily (without further decomposition)?
   - Base/stopping case

When you turn this into a program, you end up with functions that call themselves (*recursive functions*)

```
int f(int x)
{
 int y;

 if(x==0)
    return 1;
 else {
    y = 2 * f(x-1);
    return y+1;
 }
}
```

Problems defined recursively

There are many problems whose solution can be defined recursively

Example: *n factorial*

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! * n & \text{if } n > 0 \end{cases}$$

(*recursive* solution)

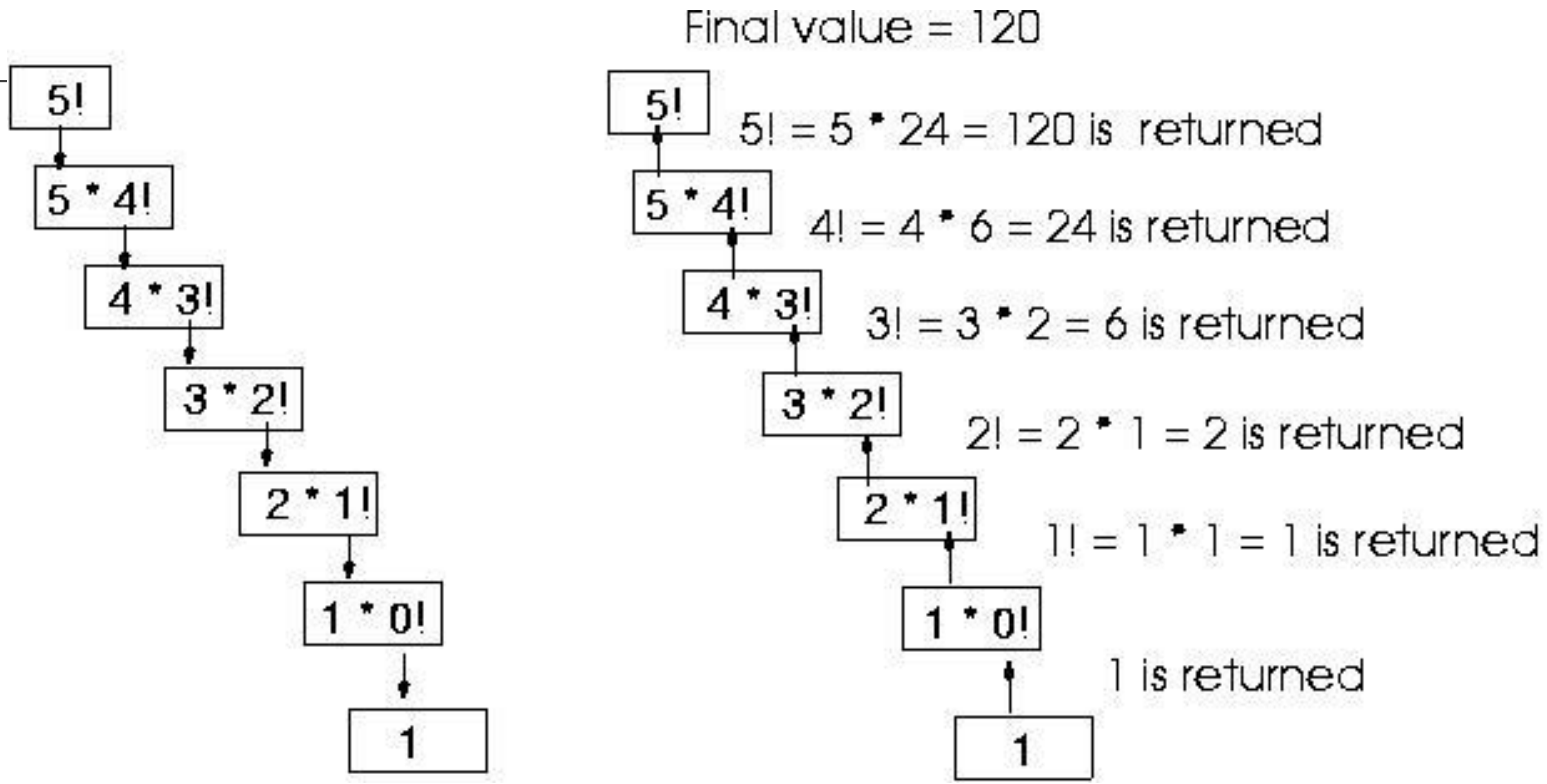
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1*2*3*\ldots*(n-1)*n & \text{if } n > 0 \end{cases}$$

(*closed form* solution)

Coding the factorial function

Recursive implementation

```
int Factorial(int n)
{
 if (n==0)  // base case
   return 1;
 else
   return n * Factorial(n-1);
}
```



Final value = 120
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1 * 0!
1
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1! = 1 * 1 = 1 is returned
1 * 0!
1 is returned
1

Coding the factorial function (cont.)

Iterative implementation

```
int Factorial(int n)
{
 int fact = 1;

 for(int count = 2; count <= n; count++)
   fact = fact * count;

 return fact;
}
```

Another example:
n choose *k* (combinations)

Given *n* things, how many different sets of size *k* can be chosen?

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \qquad 1 < k < n \qquad \textit{(recursive solution)}$$

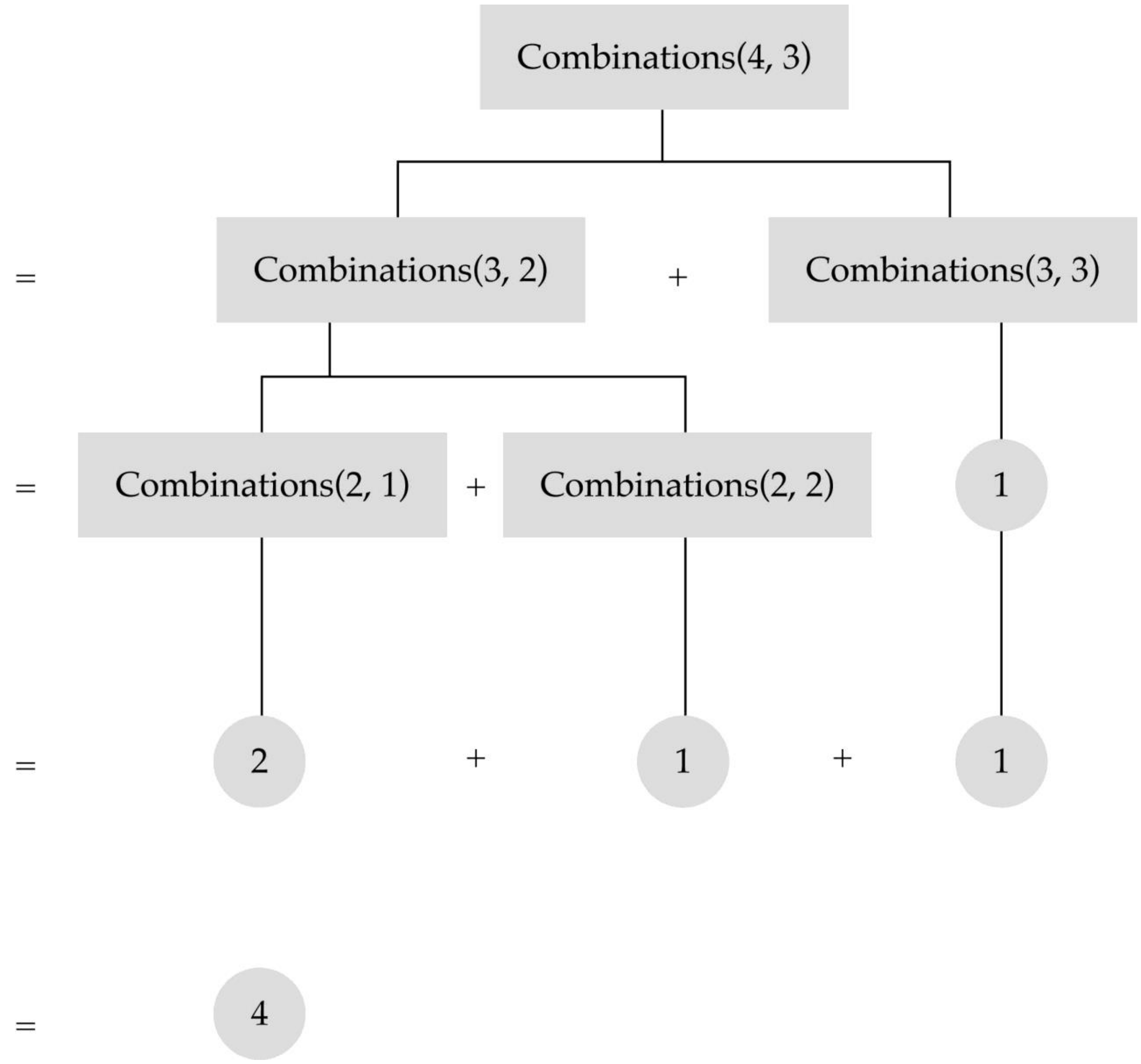
$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 1 < k < n \qquad \textit{(closed-form solution)}$$

with base cases:

$$\binom{n}{1} = n \quad (k = 1), \qquad \binom{n}{n} = 1 \quad (k = n)$$

n choose k (combinations)

```
int Combinations(int n, int k)
{
 if(k == 1)  // base case 1
   return n;
 else if (n == k)  // base case 2
   return 1;
 else
   return(Combinations(n-1, k) + Combinations(n-1, k-1));
}
```


Combinations(4, 3)
= Combinations(3, 2) + Combinations(3, 3)
= Combinations(2, 1) + Combinations(2, 2) 1
= 2 + 1 + 1
= 4

Recursion vs. iteration

Iteration can be used in place of recursion

- An iterative algorithm uses a *looping construct*
- A recursive algorithm uses a *branching structure*

Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions

Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

Fibonacci Numbers

The *N*th Fibonacci number is the sum of the previous two Fibonacci numbers

0, 1, 1, 2, 3, 5, 8, 13, …

Recursive Design:

- Decomposition & Composition
  - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
- Base case:
  - fibonacci(1) = 0
  - fibonacci(2) = 1

`fibonacci` Method

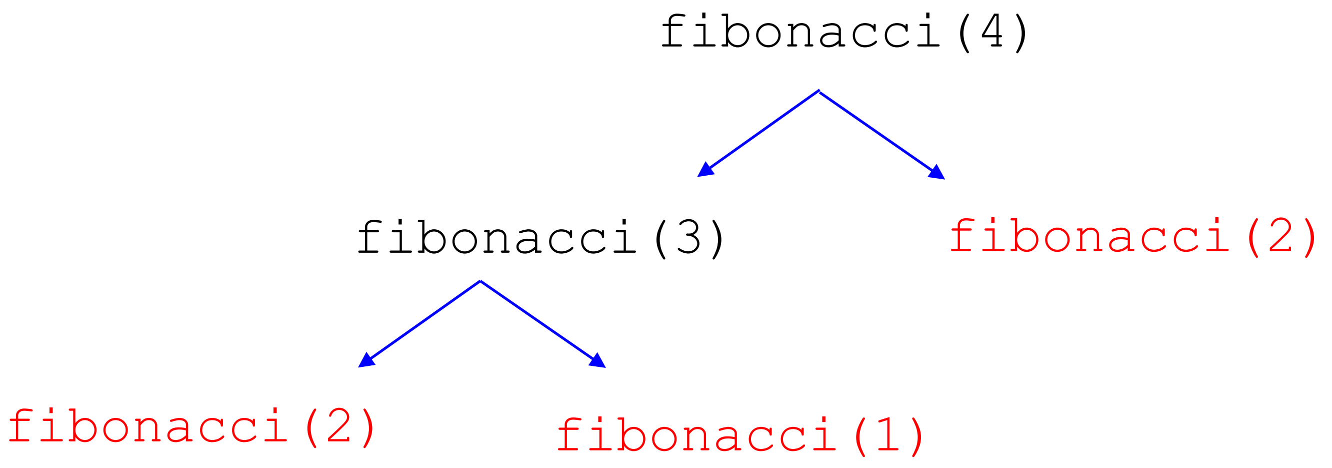
```
int fibonacci(int n)
{
   int fib;
   if (n > 2)
     fib = fibonacci(n-1) + fibonacci(n-2);
   else if (n == 2)
     fib = 1;
   else
     fib = 0;
   return fib;
}
```

Execution Trace (decomposition)

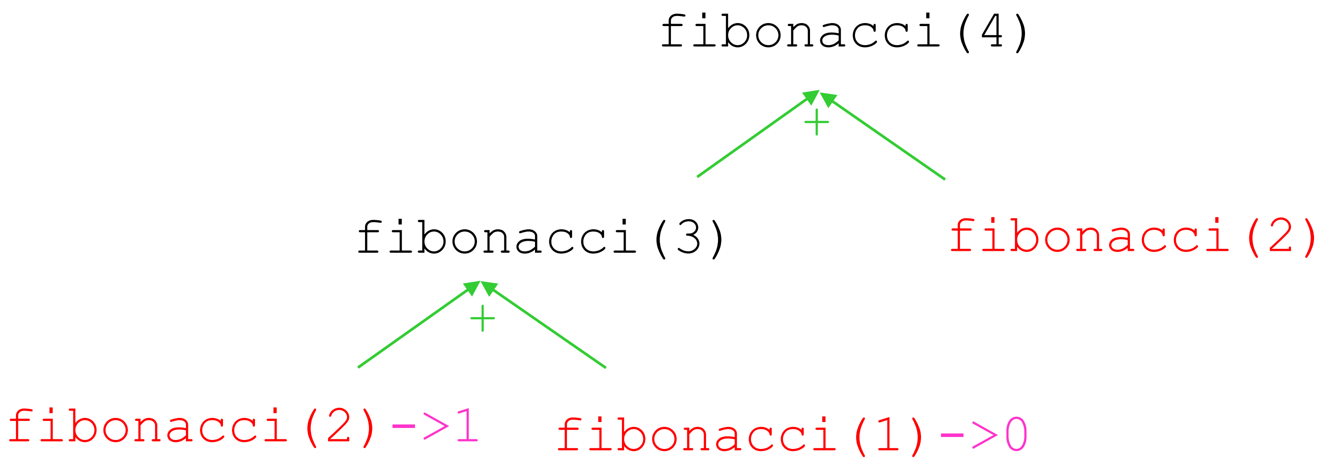
fibonacci(4)

fibonacci(3)　　　fibonacci(2)

Execution Trace (decomposition)



Execution Trace (composition)



Execution Trace (composition)

fibonacci(4)

+

fibonacci(3)->1 fibonacci(2)->1

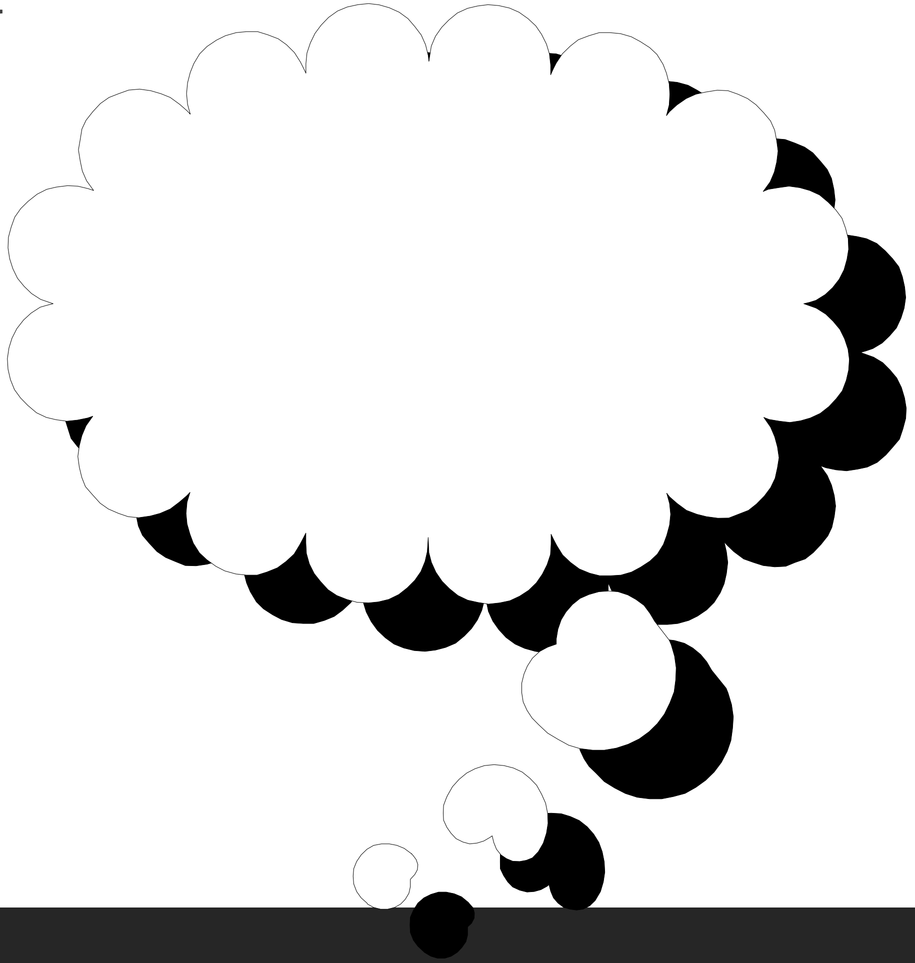
Execution Trace (composition)

```
fibonacci(4)->2
```

Case Study

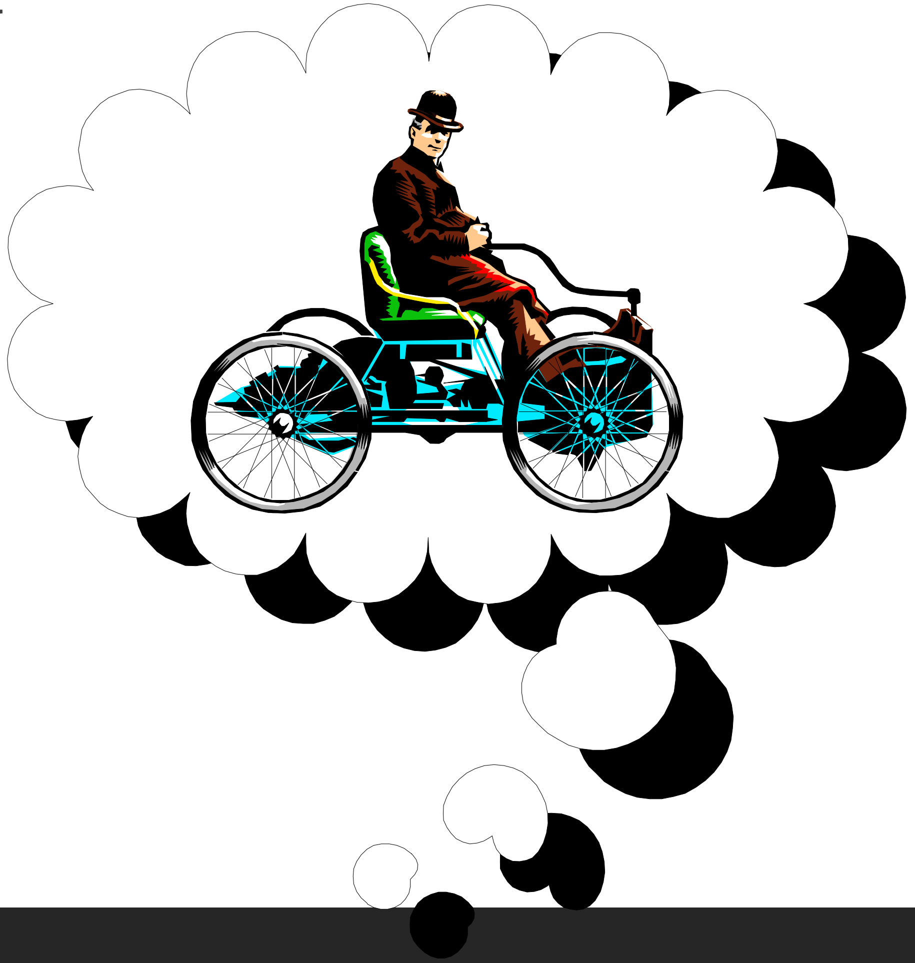
A Car Object

To start the example, think about your favorite family car



A Car Object

To start the example, think about your favorite family car

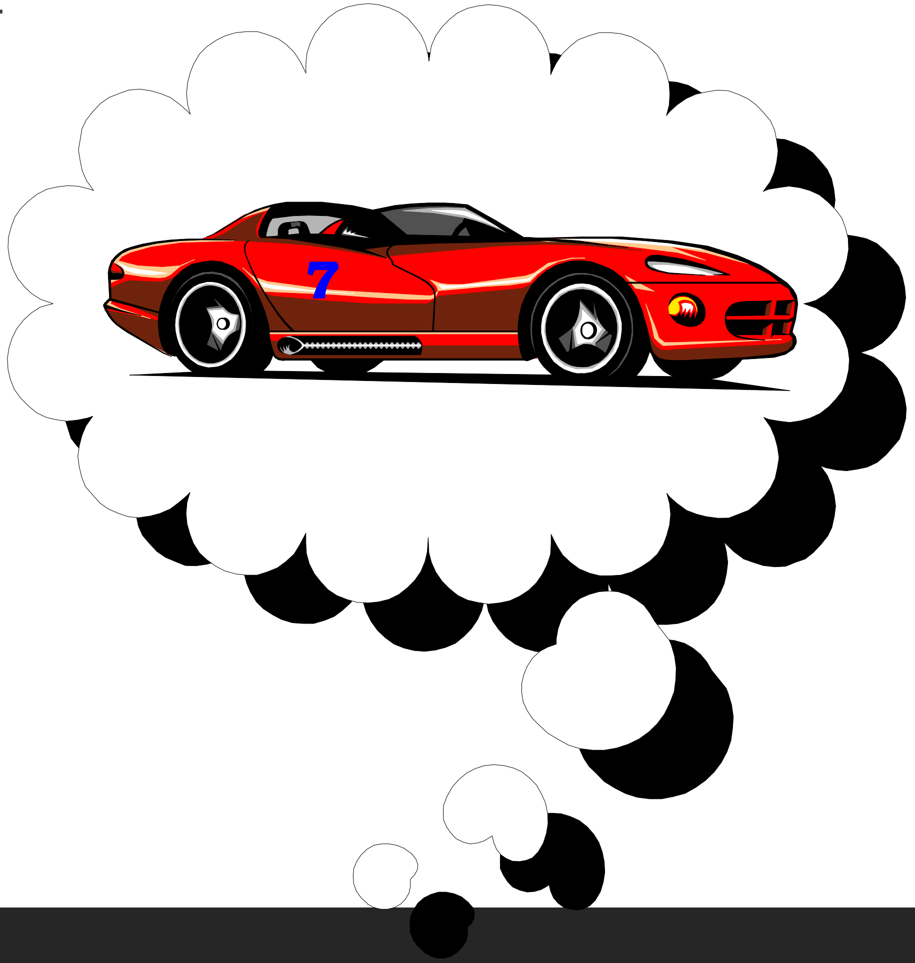


A Car Object

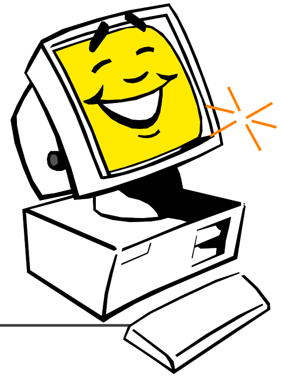
To start the example, think about your favorite family car

A Car Object

To start the example, think about your favorite family car

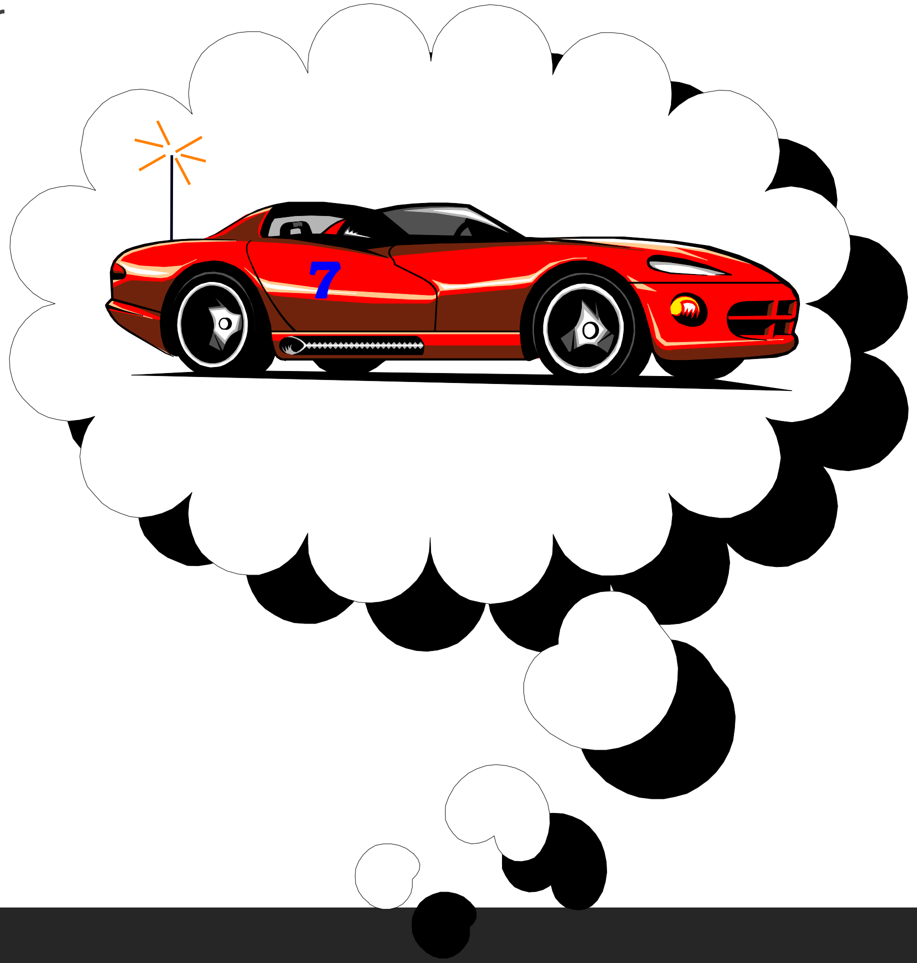


A Car Object



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

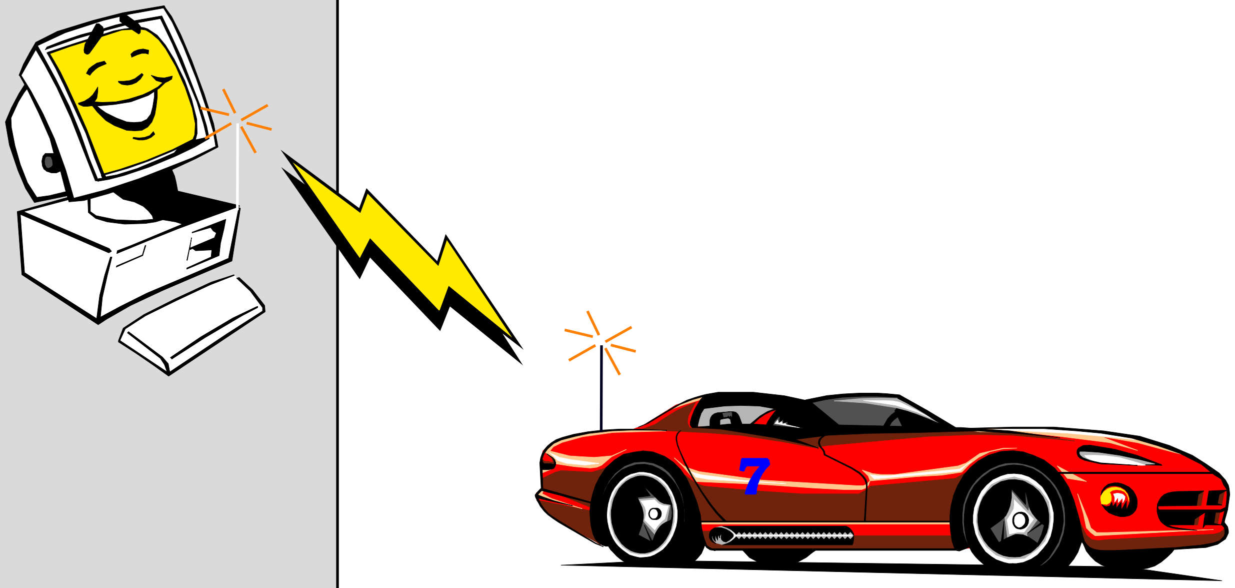
```
class car
{
public:
        car(int car_number);
        void move( );
        void turn_around( );
        bool is_blocked;
private:
        { We don't need to know the private fields! }
         . . .
};
```

The Constructor

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


   . . .
```

When we declare a Car

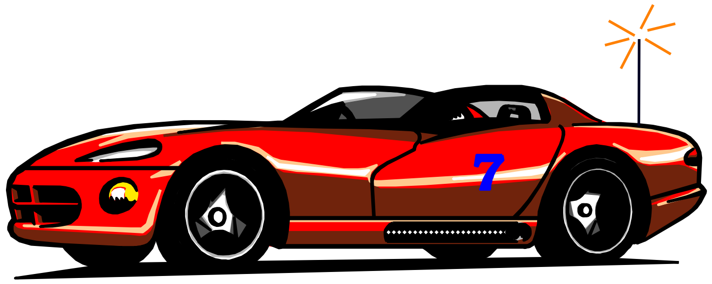
and activate the constructor, the computer makes a radio link with a car that has a particular number.



The turn_around Function

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

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

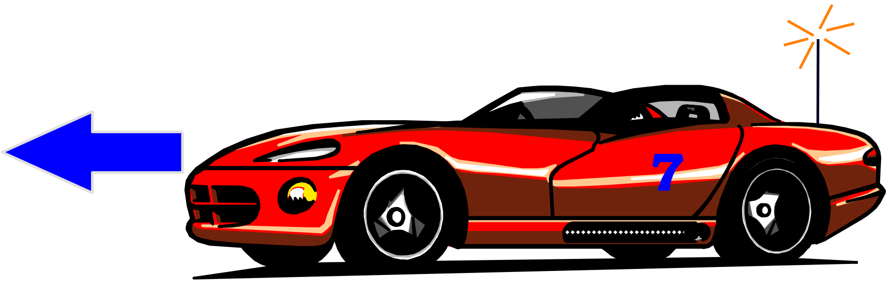
When we activate turn_around, the computer signals the car to turn 180 degrees.



The move Function

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

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

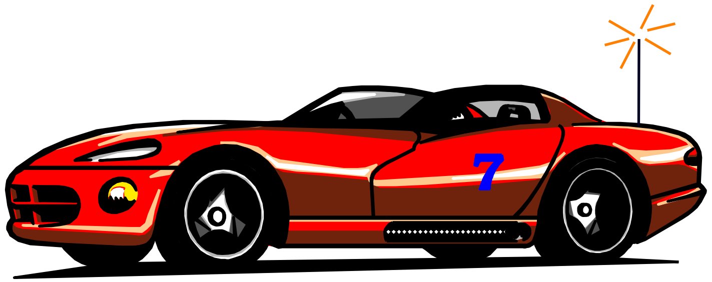
When we activate move, the computer signals the car to move forward one foot.



The move Function

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

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

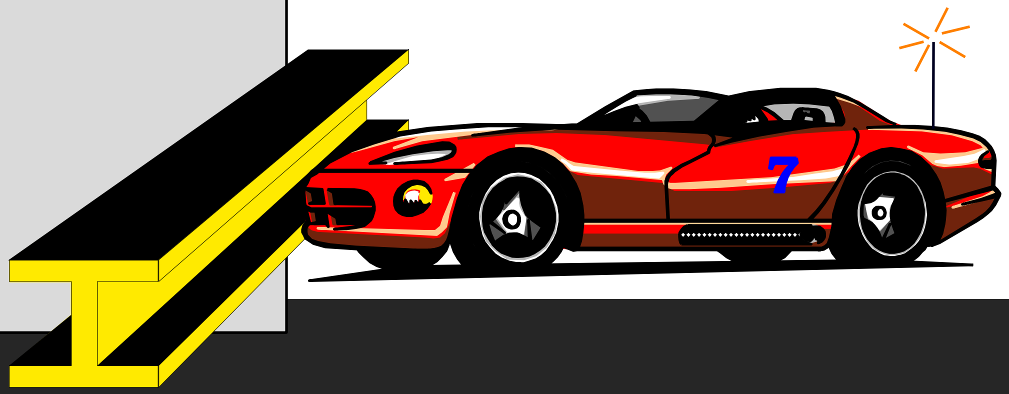
When we activate move, the computer signals the car to move forward one foot.



The is_blocked( ) Function

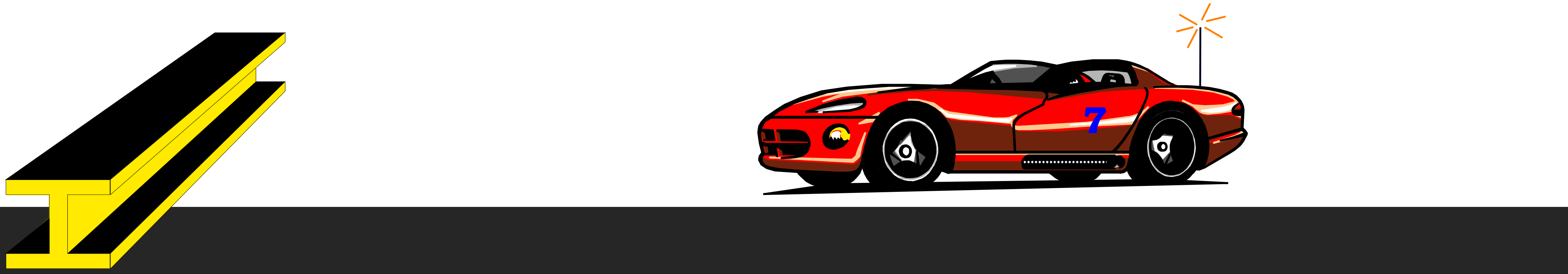
```
int main( )
{
   car racer(7);

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

The is_blocked member function detects barriers.



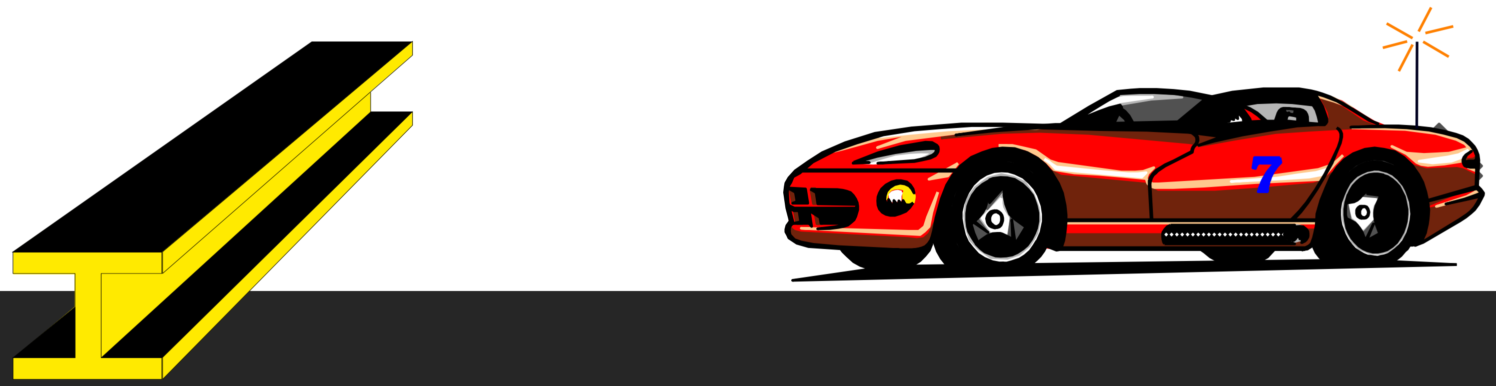
Your Mission

Write a function which will move a car forward until it reaches a barrier…



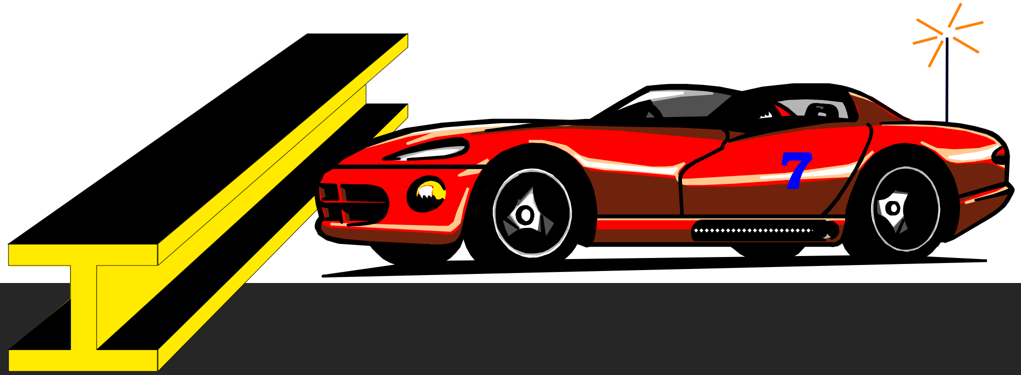
Your Mission

Write a function which will move a car forward until it reaches a barrier…



Your Mission

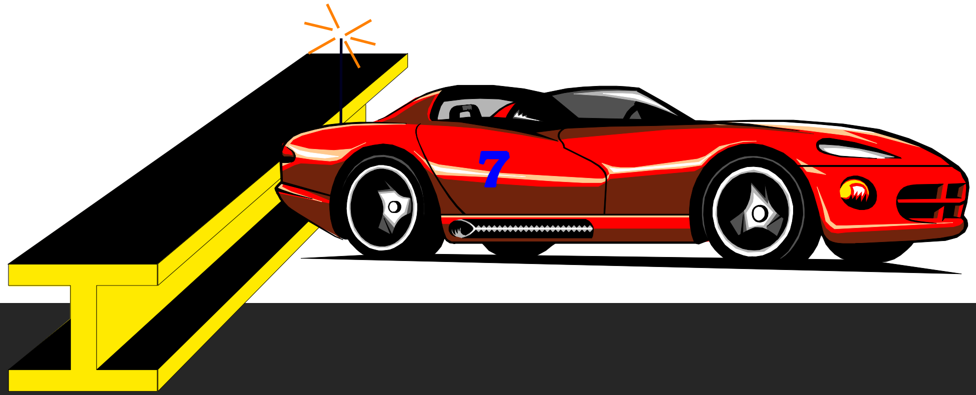
Write a function which will move a car forward until it reaches a barrier…



Your Mission

Write a function which will move a car forward until it reaches a barrier…

…then the car is turned around…

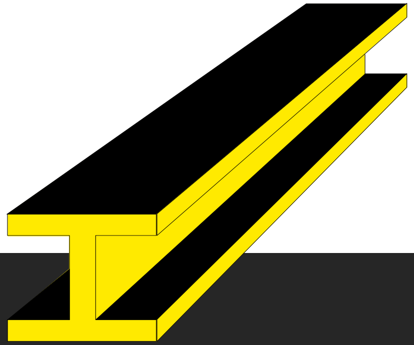


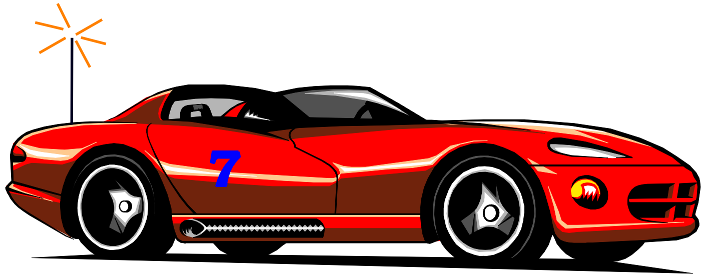
Your Mission

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.



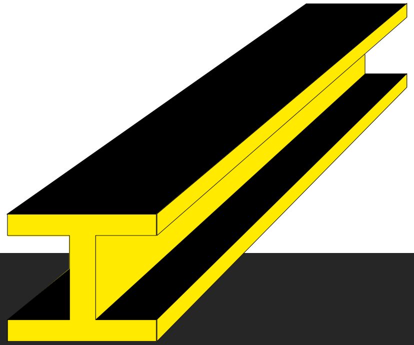


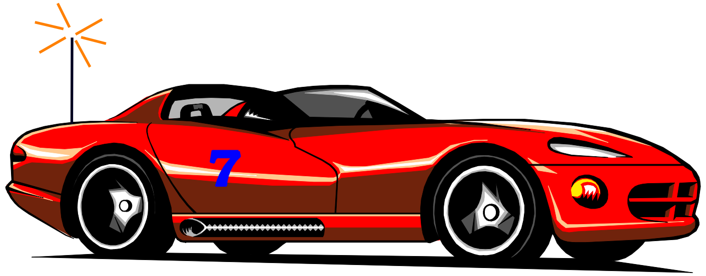
Your Mission

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.





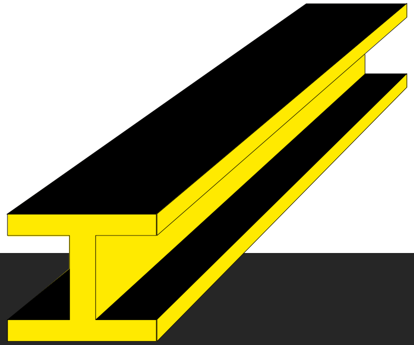
Your Mission

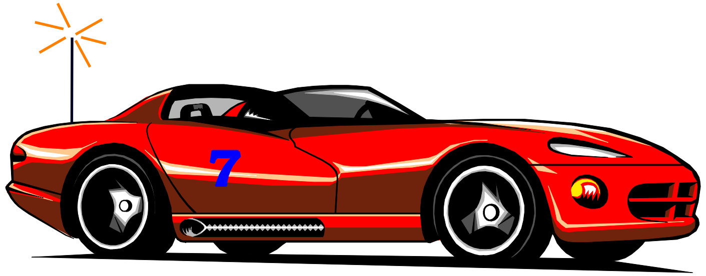
```
void  reflexion(car& moving_car);
```

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 reflexion

```
void  reflexion(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 reflexion

```
void reflexion(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 reflexion

```
void reflexion(car& moving
```

- if moving_car.is_blocked( ), the … e,
  just turn the car around.
- Otherwise, the car has not yet

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

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

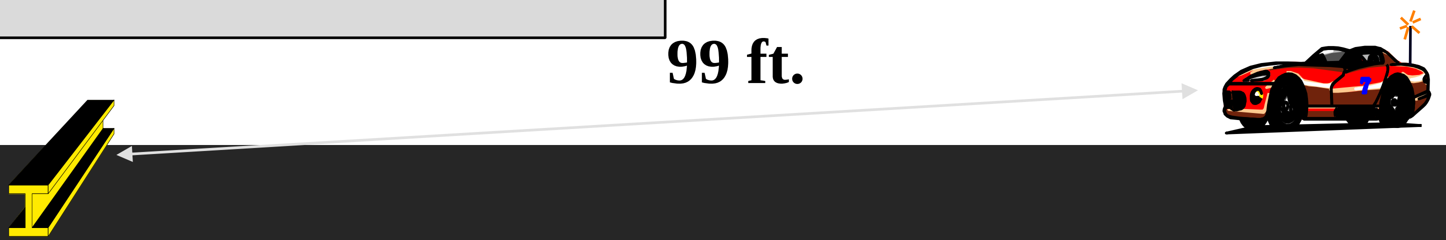
Pseudocode for reflexion

```
void reflexion(car& moving
```

- if moving_car.is_blocked( ), the ... e,
  just turn the car around.
- Otherwise, the car has not yet

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.

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



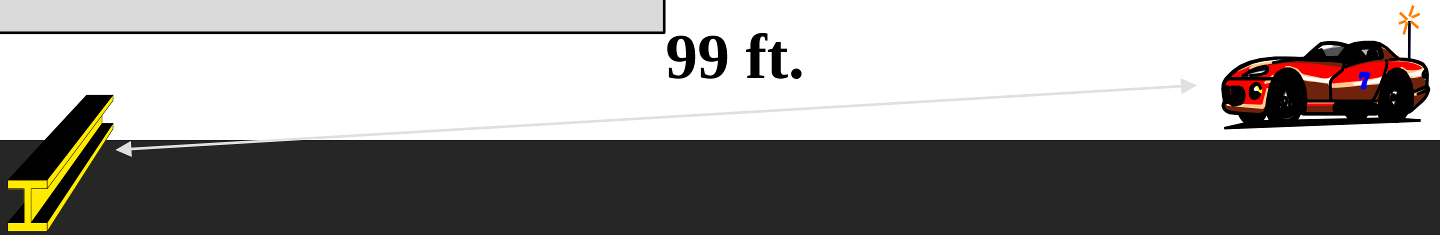
Pseudocode for reflexion

```
void reflexion(car& moving
```

- if moving_car.is_blocked( ), the ... e,
  just turn the car around.
- Otherwise, the car has not yet

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

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



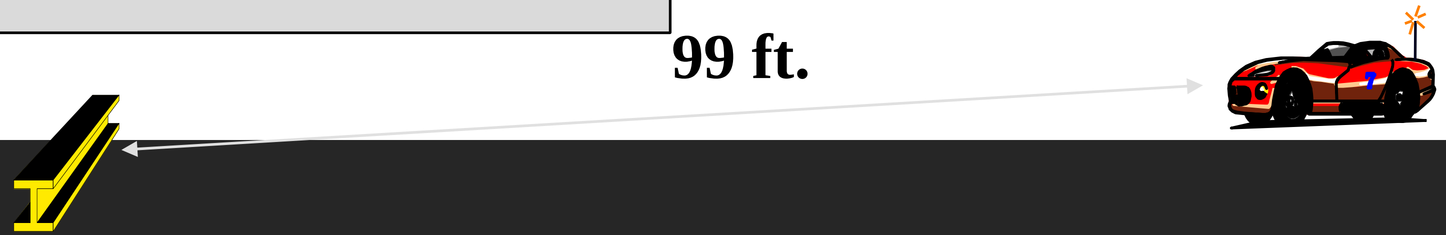
Pseudocode for reflexion

```
void reflexion(car& moving
```

- if moving_car.is_blocked( ), the ... e,
  just turn the car around.
- Otherwise, the car has not yet

Make a recursive call to solve the smaller problem.

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



Pseudocode for reflexion

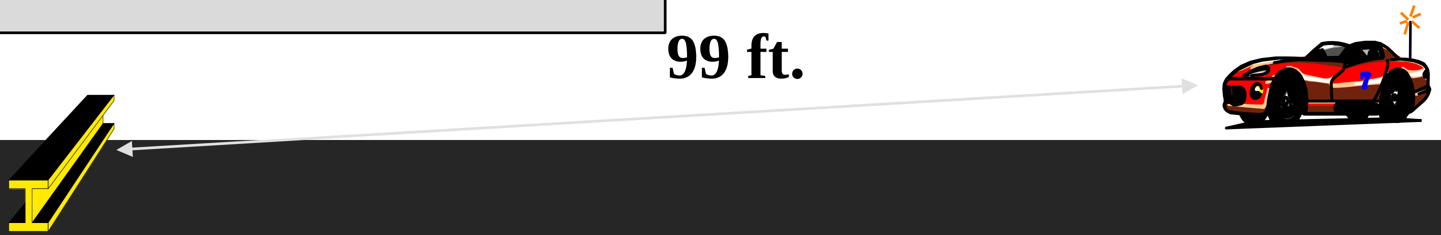
```
void reflexion(car& moving
```

- if moving_car.is_blocked( ), the e, just turn the car around.
- Otherwise, the car has not yet

The recursive call will solve the smaller problem.

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

99 ft.



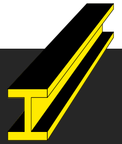
Pseudocode for reflexion

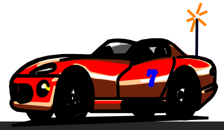
void reflexion(car& moving

- if moving_car.is_blocked( ), the
  just turn the car around.
- Otherwise, the car has not yet

The recursive call will solve the smaller problem.

moving_car.move( );
reflexion(moving_car);
. . .





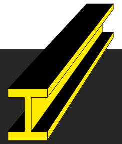
Pseudocode for reflexion

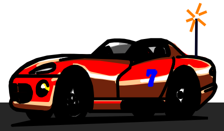
```
void reflexion(car& moving
```

- if moving_car.is_blocked( ), the … e, just turn the car around.
- Otherwise, the car has not yet

The recursive call will solve the smaller problem.

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





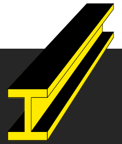
Pseudocode for reflexion

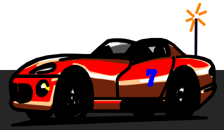
```
void reflexion(car& moving
```

- if moving_car.is_blocked( ), the ... just turn the car around.
- Otherwise, the car has not yet

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

The recursive call will solve the smaller problem.





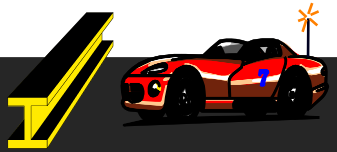
Pseudocode for reflexion

```
void reflexion(car& moving
```

- if moving_car.is_blocked( ), the
  just turn the car around.
- Otherwise, the car has not yet

The recursive call will solve the smaller problem.

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



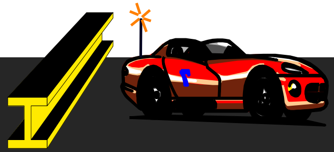
Pseudocode for reflexion

void reflexion(car& moving

- if moving_car.is_blocked( ), the ... e, just turn the car around.
- Otherwise, the car has not yet

The recursive call will solve the smaller problem.

moving_car.move( );
reflexion(moving_car);
. . .



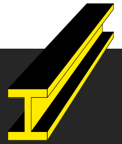
Pseudocode for reflexion

```
void reflexion(car& moving
```

- if moving_car.is_blocked( ), the
  just turn the car around.
- Otherwise, the car has not yet

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

The recursive call will solve the smaller problem.



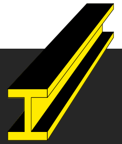
Pseudocode for reflexion

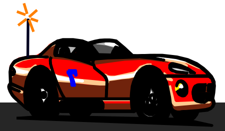
```
void reflexion(car& moving
```

- if moving_car.is_blocked( ), the … e,
  just turn the car around.
- Otherwise, the car has not yet

The recursive call will solve the smaller problem.

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





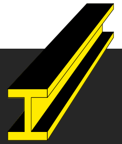
Pseudocode for reflexion

```
void reflexion(car& moving
```

- if moving_car.is_blocked( ), the [...] e, just turn the car around.
- Otherwise, the car has not yet

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

The recursive call will solve the smaller problem.



Pseudocode for reflexion

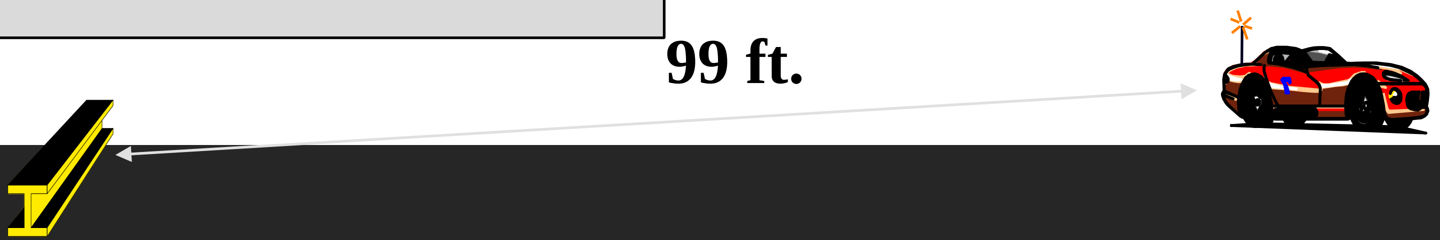
```
void reflexion(car& moving
```

- if moving_car.is_blocked( ), the
  just turn the car around.
- Otherwise, the car has not yet

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

The recursive call will solve the smaller problem.

99 ft.



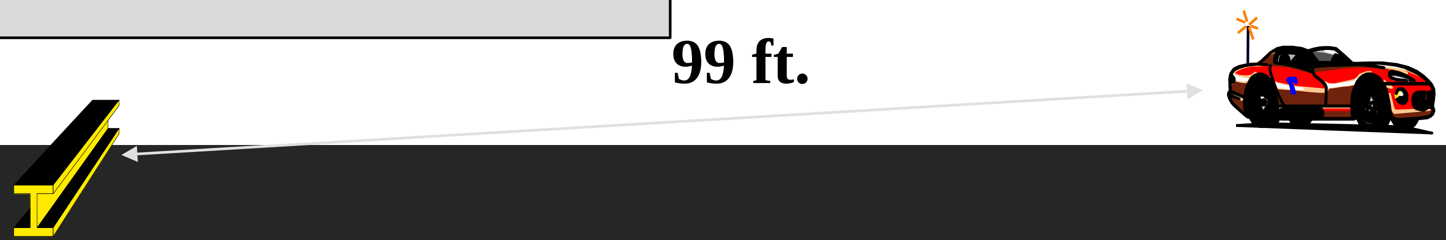
Pseudocode for reflexion

```
void reflexion(car& moving
```

- if moving_car.is_blocked( ), the e, just turn the car around.
- Otherwise, the car has not yet

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

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



Pseudocode for reflexion

```
void reflexion(car& moving
```

- if moving_car.is_blocked( ), the e, just turn the car around.
- Otherwise, the car has not yet

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

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

Pseudocode for reflexion

```
void reflexion(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( );
reflexion(moving_car);
moving_car.move( );
```

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

Pseudocode for reflexion

```
void reflexion(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( );
reflexion(moving_car);
moving_car.move( );
```

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

Pseudocode for reflexion

```
void reflexion(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( );
reflexion(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 reflexion

```
void reflexion(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( );
reflexion(moving_car);
moving_car.move( );
```

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

Pseudocode for reflexion

```
void reflexion(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( );
reflexion(moving_car);
moving_car.move( );
```

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

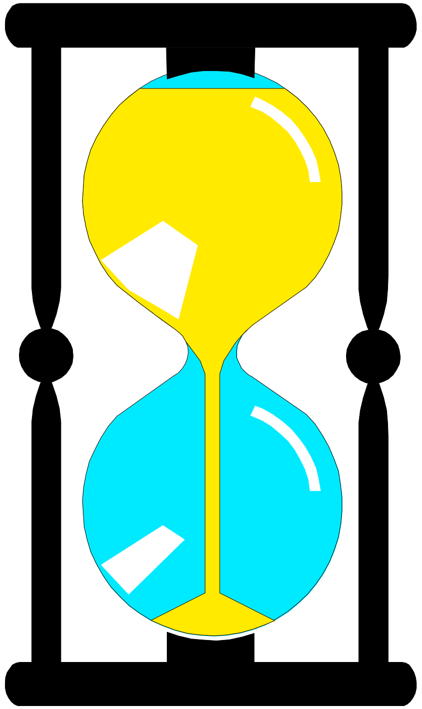
Implementation of reflexion

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

An Exercise

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

```
void car::reflexion( )
{
   . . .
```



You have 2 minutes to write the implementation.

An Exercise

One solution:

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

Credits and Acknowledgements

- Lectures by Prof. Yung Yi, KAIST, South Korea.
- Addison Wesley Longman, Lectures with Data Structures and Other Objects Using C++ by Michael Main and Walter Savitch.