

Hands-on Example 2: Referee of Tic-Tac-Toe

X	X	O
		O
O	X	

- Two Player Game (X-player & O-player).
- The game proceeds when each player places 'X' or 'O' in a blank space in the matrix in alterante turns.

- Initial configuration : the board is empty.
- Winning : if there is a sequence of three consecutive cells (vertical, horizontal, forward diagonal or reverse diagonal) where the player's symbol appears.
- Draw : if the board is full, but neither of the players has reached a winning configuration yet.

X	X	O
	X	O
O	X	
X	X	O
O	O	X
X	X	O

Programming the Referee: functions

- Representing the board: A 3×3 character array. Stores, 'X' or 'O' or Blank in each cell.

Programming the Referee: functions

- Representing the board: A 3×3 character array. Stores, 'X' or 'O' or Blank in each cell.
- Define it as a global character array `board[3][3]` of order 3×3 .

Programming the Referee: functions

- Representing the board: A 3×3 character array. Stores, 'X' or 'O' or Blank in each cell.
- Define it as a global character array `board[3][3]` of order 3×3 .

Think modular : Tasks involved for a referee - *the board keeper*.

Programming the Referee: functions

- Representing the board: A 3×3 character array. Stores, 'X' or 'O' or Blank in each cell.
- Define it as a global character array `board[3][3]` of order 3×3 .

Think modular : Tasks involved for a referee - *the board keeper*.

- Show the board to both players.

Programming the Referee: functions

- Representing the board: A 3×3 character array. Stores, 'X' or 'O' or Blank in each cell.
- Define it as a global character array `board[3][3]` of order 3×3 .

Think modular : Tasks involved for a referee - *the board keeper*.

- Show the board to both players.
- Check if any of them won, if so, declare won.

Programming the Referee: functions

- Representing the board: A 3×3 character array. Stores, 'X' or 'O' or Blank in each cell.
- Define it as a global character array `board[3][3]` of order 3×3 .

Think modular : Tasks involved for a referee - *the board keeper*.

- Show the board to both players.
- Check if any of them won, if so, declare won.
- If not, ask for a move from the correct player.

Programming the Referee: functions

- Representing the board: A 3×3 character array. Stores, 'X' or 'O' or Blank in each cell.
- Define it as a global character array `board[3][3]` of order 3×3 .

Think modular : Tasks involved for a referee - *the board keeper*.

- Show the board to both players.
- Check if any of them won, if so, declare won.
- If not, ask for a move from the correct player.
- Check if the move is legal, if so, update the board.

Programming the Referee: functions

- Representing the board: A 3×3 character array. Stores, 'X' or 'O' or Blank in each cell.
- Define it as a global character array `board[3][3]` of order 3×3 .

Think modular : Tasks involved for a referee - *the board keeper*.

- Show the board to both players.
- Check if any of them won, if so, declare won.
- If not, ask for a move from the correct player.
- Check if the move is legal, if so, update the board.
- Keep doing this until board is full or somebody wins.

Four Functions:

We will do this using four functions:

- `showconfig()` : to print the current configuration of the board.

Four Functions:

We will do this using four functions:

- `showconfig()` : to print the current configuration of the board.
- `checkwin()` : to check if the current configuration of the board (available in the global array `board`) is a winning configuration for any of the players, if yes, print the appropriate message. If it is a draw, then also it can print an appropriate message.

Four Functions:

We will do this using four functions:

- `showconfig()` : to print the current configuration of the board.
- `checkwin()` : to check if the current configuration of the board (available in the global array `board`) is a winning configuration for any of the players, if yes, print the appropriate message. If it is a draw, then also it can print an appropriate message.
- `checklegal(i,j)` : to check if putting a symbol in the `i,j` the location of the board is legal or not. That is, is a symbol already there? Then the move is illegal.

Four Functions:

We will do this using four functions:

- `showconfig()` : to print the current configuration of the board.
- `checkwin()` : to check if the current configuration of the board (available in the global array `board`) is a winning configuration for any of the players, if yes, print the appropriate message. If it is a draw, then also it can print an appropriate message.
- `checklegal(i,j)` : to check if putting a symbol in the `i,j` the location of the board is legal or not. That is, is a symbol already there? Then the move is illegal.
- `putsymbol(i,j,c)` : Assuming we checked the legality of the move by the player, put down the symbol `c` (which is either 'X' or 'O') at the entry `board[i][j]`.

Pseudo-code of the main program

Now the main program is compact and intuitive.

```
// Assume 1 and 2 are used for X and O.  
p = 0  
while (checkwin() returns false)  
{  
    showconfig();  
    read the next move (i,j) of player no:(p+1)  
    // note that p+1 is either 1 or 2.  
  
    if (checklegal(i,j) == false) continue;  
    putsymbol(i,j,(p+1));  
    p = (p+1) % 2.  
}  
Print "Game Over"
```

The prototype declarations

```
#include <stdio.h>

#define N 3
char board[N][N];
char player[2] = {'X','O'};

void init();
void showconfig(void);
int checkwin(void);
int checklegal(int, int);
int putsymbol(int,int,char);

int main()
{
    init();
    ....
}
```

Implementing `showconfig()`

Implementing `showconfig()`

Exercise on printing a 2-dimensional array in matrix form.

Implementing showconfig()

Exercise on printing a 2-dimensional array in matrix form.

```
void showconfig()
{
    printf("\n-----\n");
    for (int i=0; i<N; i++)
    {
        for (int j=0; j<N; j++)
            printf("| %c ",board[i][j]);
        printf("| \n-----\n");
    }
}
```

Implementing `checkwin()` : The naive way

Idea 1 : `checkwin` : is a close cousin of the *character grid question*.

Implementing `checkwin()` : The naive way

Idea 1 : `checkwin` : is a close cousin of the *character grid question*.

Recall character grid question : *Given a character grid, and a string s , check if the rows, columns or diagonals of the grid that contain s .*

Implementing `checkwin()` : The naive way

Idea 1 : `checkwin` : is a close cousin of the *character grid question*.

Recall character grid question : *Given a character grid, and a string s , check if the rows, columns or diagonals of the grid that contain s .*

- Let `board[N][N]`, with $N = 3$, be the character grid.
- Do the character search with $s = \text{XXX}$ to determine if X-player wins.
- Do the character search with $s = \text{000}$ to determine if O-player wins.

So we can reuse that code.

Implementing checkwin()

```
int checkwin(void)
{
    int i,j;
    // checking if X won because of a row of Xs
    for(i = 0; i < N; i++) {
        for(j = 0; j < N; j++)
            if (board[i][j] != 'X') break;
        if(j == N) {
            printf("X won");
            return 1;
        }
    }
    // do similar for columns and diagonals.
    // do similar for O-symbol
    return 0;
}
```

A better "modular" design for `checkwin()`

Idea 2 : Think Modular !

A better "modular" design for `checkwin()`

Idea 2 : Think Modular !

New function `checkwindir(int dir, char player)` : checks the winning configuration for player ('X'/'O') in the direction (1/2/3/4 - representing horiz/vert/diag/revdiag).

A better "modular" design for checkwin()

Idea 2 : Think Modular !

New function `checkwindir(int dir, char player)` : checks the winning configuration for player ('X'/'O') in the direction (1/2/3/4 - representing horiz/vert/diag/revdiag).

Pseudocode for `checkwindir(dir,player)`

- for $i=1$ to N
- for $j=1$ to N
 - If $dir = 1$ all checks should be `board[i][j] != 'X'`.
 - If $dir = 2$ all checks should be `board[j][i] != 'X'`.
 - If $dir = 3$ all checks should be `board[j][j] != 'X'`.
 - If $dir = 4$ all checks should be `board[j][N-j-1] != 'X'`.
- If any check fails, then try next i . If all succeeds for the full run of the j -loop, then declare WINNING.

A better "modular" design for checkwin()

```
int checkwindir(int dir, char player)
{
    int s,t,i,j;
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            switch (dir) {
                case 1: s=i; t=j; break;
                case 2: s=j; t=i; break;
                case 3: s=j; t=j; break;
                case 4: s=j; t=N-j-1; break;
            }
            if (board[s][t] != player) break;
        }
        if (j == N) return (1);
    }
    return(0);
}
```

```
int checkwin(void)
{
    for (int dir=1; dir<5; dir++)
        for (int p=0; p<2; p++)
            if (checkwindir(dir,player[p]) == 1)
                return (1);

    return (0);
}
```

Quiz 1: Array Indices

For a 3×3 board, the reverse diagonal uses

`board[j][N-j-1]`.

List the three array indices visited as j runs from 0 to 2.

Quiz 1: Array Indices

For a 3×3 board, the reverse diagonal uses

`board[j][N-j-1]`.

List the three array indices visited as j runs from 0 to 2.

Answer:

$(0, 2), (1, 1), (2, 0)$.

These are the top-right, centre, and bottom-left cells.

Two more functions to define

- `checklegal(i,j)` : to check if putting a symbol in the `i,j` the location of the board is legal or not. That is, is a symbol already there? Then the move is illegal.

Two more functions to define

- `checklegal(i,j)` : to check if putting a symbol in the `i,j` the location of the board is legal or not. That is, is a symbol already there? Then the move is illegal.
- `putsymbol(i,j,c)` : Assuming we checked the legality of the move by the player, put down the symbol `c` (which is either 'X' or 'O') at the entry `board[i][j]`.

Take Aways

- Functions : Modular Programming. Build programs brick by brick. Reusing built and tested part.
- Declaration, Definition and Invocation of functions.
- Block and Scope. Local and Global Variables.
- Hands-on examples of using functions.

Questions :

- Can a function invoke other functions?

Take Aways

- Functions : Modular Programming. Build programs brick by brick. Reusing built and tested part.
- Declaration, Definition and Invocation of functions.
- Block and Scope. Local and Global Variables.
- Hands-on examples of using functions.

Questions :

- Can a function invoke other functions? **Yes !.**

Take Aways

- Functions : Modular Programming. Build programs brick by brick. Reusing built and tested part.
- Declaration, Definition and Invocation of functions.
- Block and Scope. Local and Global Variables.
- Hands-on examples of using functions.

Questions :

- Can a function invoke other functions? **Yes !.**
- Can a function invoke itself?

Take Aways

- Functions : Modular Programming. Build programs brick by brick. Reusing built and tested part.
- Declaration, Definition and Invocation of functions.
- Block and Scope. Local and Global Variables.
- Hands-on examples of using functions.

Questions :

- Can a function invoke other functions? **Yes !.**
- Can a function invoke itself? **Yes! Recursion.**

New Idea - Recursive Function Calls

New Idea - Recursive Function Calls

Can a function invoke itself?

New Idea - Recursive Function Calls

Can a function invoke itself? **Yes !** - but why would it want to?

New Idea - Recursive Function Calls

Can a function invoke itself? **Yes !** - but why would it want to?

Here is a situation :

- We wish to define the function `int fact(int n)` : to return the factorial of a number n .

New Idea - Recursive Function Calls

Can a function invoke itself? **Yes !** - but why would it want to?

Here is a situation :

- We wish to define the function `int fact(int n)` : to return the factorial of a number n .
- We have not written `fact` function yet, but we want to write it using itself.

New Idea - Recursive Function Calls

Can a function invoke itself? **Yes !** - but why would it want to?

Here is a situation :

- We wish to define the function `int fact(int n)` : to return the factorial of a number n .
- We have not written `fact` function yet, but we want to write it using itself.

Here is the idea :

Write the function `fact` in such a way that :

New Idea - Recursive Function Calls

Can a function invoke itself? **Yes !** - but why would it want to?

Here is a situation :

- We wish to define the function `int fact(int n)` : to return the factorial of a number n .
- We have not written `fact` function yet, but we want to write it using itself.

Here is the idea :

Write the function `fact` in such a way that :

- it returns 1, if the argument is 1.

New Idea - Recursive Function Calls

Can a function invoke itself? **Yes !** - but why would it want to?

Here is a situation :

- We wish to define the function `int fact(int n)` : to return the factorial of a number n .
- We have not written `fact` function yet, but we want to write it using itself.

Here is the idea :

Write the function `fact` in such a way that :

- it returns 1, if the argument is 1.
- else it returns n times the result of itself when called with argument $n-1$.

fact function : Iterative vs Recursive

```
int fact(int n){  
    int i;  
    int result;  
    result = 1;  
    for (i = 1; i <= n; i++)  
        result = result * i;  
    return result;  
}
```

fact function : Iterative vs Recursive

```
int fact(int n){  
    int i;  
    int result;  
    result = 1;  
    for (i = 1; i <= n; i++)  
        result = result * i;  
    return result;  
}
```

invocation : `f = fact(4);`

fact function : Iterative vs Recursive

```
int fact(int n){  
    int i;  
    int result;  
    result = 1;  
    for (i = 1; i <= n; i++)  
        result = result * i;  
    return result;  
}
```

invocation : `f = fact(4);`

```
int fact(int n){  
    if (n == 1) return(1);  
    return (n*fact(n-1));  
}
```

fact function : Iterative vs Recursive

```
int fact(int n){  
    int i;  
    int result;  
    result = 1;  
    for (i = 1; i <= n; i++)  
        result = result * i;  
    return result;  
}
```

invocation : `f = fact(4);`

```
int fact(int n){  
    if (n == 1) return(1);  
    return (n*fact(n-1));  
}
```

- `(n == 1)` case is called the **base case**. If it not provided, it will turn out to be an infinite recursion.

..... to be continued.

Quiz 2: Tracing Recursion

Suppose `fact(4)` uses the recursive definition on the previous slide.

1. What sequence of function calls occurs?
2. How many calls to `fact` occur, including the base case?
3. What value does the original call return?

Quiz 2: Tracing Recursion

Suppose `fact(4)` uses the recursive definition on the previous slide.

1. What sequence of function calls occurs?
2. How many calls to `fact` occur, including the base case?
3. What value does the original call return?

Answer: The calls are

`fact(4)`, `fact(3)`, `fact(2)`, `fact(1)`.

There are 4 calls, and `fact(4)` returns $4 \times 3 \times 2 \times 1 = 24$.

CS1100 – Introduction to Programming

Prashanth L.A.

Department of Computer Science and Engineering, IIT Madras

- What happens when a function calls itself?
- Factorial, Pascal's triangle, and Virahanka numbers
- Recursive GCD and the largest element of an array

Adapted from teaching material shared by CSE, IIT Madras colleagues.

A smaller instance of the same problem

A recursive function calls itself with a smaller instance.

$$4! = 4 \times 3! \quad 3! = 3 \times 2! \quad 2! = 2 \times 1!$$

Factorial for a nonnegative integer n

$$n! = \begin{cases} 1, & n = 0, \\ n(n-1)!, & n > 0. \end{cases}$$

- **Base case:** an answer we can return immediately.
- **Recursive case:** solve a smaller instance, then use its answer.

Factorial: a loop and a recursive function

Iteration

```
int fact_iter(int n)
{
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}
```

Recursion

```
int fact(int n)
{
    if (n == 0) return 1;
    int smaller = fact(n - 1);
    return n * smaller;
}
```

Both return 24 for $n = 4$ and 1 for $n = 0$.

Use small nonnegative inputs so the result fits in `int`.

Recursive control flow: calls going down

main calls fact(4). Each call has its own value of n.

Active call	What it does next
-------------	-------------------

fact(4)	Calls fact(3) and waits for its answer
---------	--

fact(3)	Calls fact(2) and waits for its answer
---------	--

fact(2)	Calls fact(1) and waits for its answer
---------	--

fact(1)	Calls fact(0) and waits for its answer
---------	--

fact(0)	Reaches the base case and returns 1
---------	-------------------------------------

A caller pauses at the call. It resumes when that call returns.

What a waiting call remembers

At the instant `fact(0)` is executing:

Call	Local <code>n</code>	Work still to finish
<code>fact(0)</code>	0	Return 1 now
<code>fact(1)</code>	1	Store the answer in <code>smaller</code> , return $1 \times \text{answer}$
<code>fact(2)</code>	2	Store the answer in <code>smaller</code> , return $2 \times \text{answer}$
<code>fact(3)</code>	3	Store the answer in <code>smaller</code> , return $3 \times \text{answer}$
<code>fact(4)</code>	4	Store the answer in <code>smaller</code> , return $4 \times \text{answer}$

- Parameters and ordinary local variables belong to each call.
- The call stack remembers these values and where to resume.

Recursive control flow: returns coming back

The most recent unfinished call completes first.

Call that resumes	Computation	Returns to
fact(1)	<code>smaller = 1; $1 \times 1 = 1$</code>	fact(2)
fact(2)	<code>smaller = 1; $2 \times 1 = 2$</code>	fact(3)
fact(3)	<code>smaller = 2; $3 \times 2 = 6$</code>	fact(4)
fact(4)	<code>smaller = 6; $4 \times 6 = 24$</code>	main

Where does execution continue?

In `int smaller = fact(n - 1);`, the returned value initializes `smaller`. The next statement is `return n * smaller;`.

Quiz 1: code before and after a recursive call

```
void trace(int n)
{
    if (n == 0) return;
    printf("enter %d\n", n);
    trace(n - 1);
    printf("leave %d\n", n);
}
```

Write the exact output of trace(3).

Does leave 3 print before or after leave 1?

Quiz 1: code before and after a recursive call

```
void trace(int n)
{
    if (n == 0) return;
    printf("enter %d\n", n);
    trace(n - 1);
    printf("leave %d\n", n);
}
```

Write the exact output of trace(3).

Does leave 3 print before or after leave 1?

Answer

enter 3, enter 2, enter 1, leave 1, leave 2, leave 3, each on a new line.