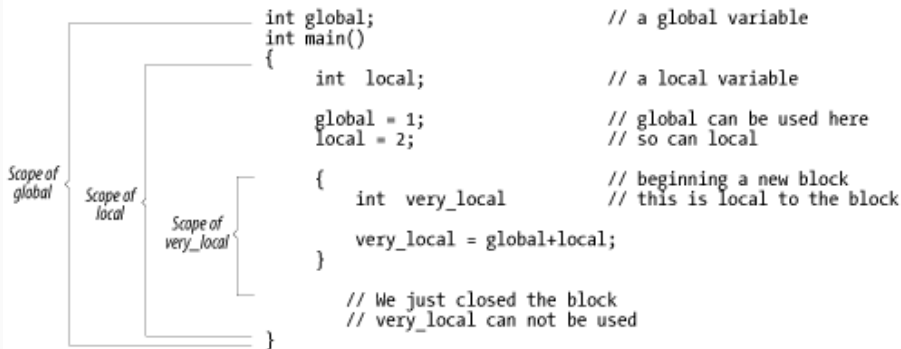


# Blocks and scope

**Block:** statements and declarations enclosed in braces.

**Scope:** the region where a declaration's name is visible.



```
int global;           // a global variable
int main()
{
    int local;        // a local variable
    global = 1;        // global can be used here
    local = 2;         // so can local
    {
        int very_local // beginning a new block
                       // this is local to the block
        very_local = global+local;
    }
    // We just closed the block
    // very_local can not be used
}
```

# Blocks and scope: example

```
#include <stdio.h>
int FindSum(int, int);
int var1 = 10;
int main()
{
    int var2 = 20;
    {
        int var3;
        var3 = FindSum(var1,var2);
        printf("%d\n", var3);
    }
    float var3=100;
    printf("%f\n", var3);
    return 0;
}
```

```
int FindSum(int a, int b)
{
    int c=a+b;
    printf("%d\n",var1);
    return c;
}
```

Treat these two listings as consecutive parts of one source file.

# Blocks and scope: example

```
#include <stdio.h>
int FindSum(int, int);
int var1 = 10;
int main()
{
    int var2 = 20;
    {
        int var3;
        var3 = FindSum(var1,var2);
        printf("%d\n", var3);
    }
    float var3=100;
    printf("%f\n", var3);
    return 0;
}
```

```
int FindSum(int a, int b)
{
    int c=a+b;
    printf("%d\n",var1);
    return c;
}
```

Treat these two listings as consecutive parts of one source file.

## Output

```
10
30
100.000000
```

# Scope of the variables

- Global `var1`: from its declaration to the end of the source file.
- `var2`: from its declaration to the end of `main`'s block.
- Integer `var3`: from its declaration to the end of the inner block.
- Floating-point `var3`: from its later declaration to the end of `main`'s block.
- Parameters `a`, `b` and local `c`: within `FindSum`.

The two declarations of `var3` introduce different variables.

What exactly is printed?

```
int main(void) {  
    int x = 10;  
    {  
        int x = 3;  
        x++;  
        printf("%d ", x);  
    }  
    printf("%d", x);  
    return 0;  
}
```

Assume `#include <stdio.h>`.

**Poll response:** Type the printed values in order.

Discuss in your group and submit one answer.

1 mark

Which one line uses a variable outside its scope?

```
int main(void) {  
    int total = 0;  
    {  
        int value = 7;  
        total += value;           // A  
    }  
    printf("%d", total);         // B  
    printf("%d", value);         // C  
    return 0;  
}
```

Assume `#include <stdio.h>`.

**Poll response:** Type A, B or C, followed by the variable name.

Discuss in your group and submit one answer.

1 mark

**S1:** 4 10

The inner `x` hides the outer `x` within the inner block. Incrementing it leaves the outer value unchanged.

**S2:** `C` value

The scope of `value` ends at the closing brace of the inner block. `total` remains in scope at `B`.

One mark each. For S1, award half a mark per correct value.

# Takeaways

- Insertion sort grows a sorted prefix. Shifting avoids repeated swaps.
- Functions support modular programming and reuse.
- A declaration describes the interface. A definition supplies the body. A call executes the function.
- Scope determines where a variable name is visible.

## Coming up

- More programs using functions.
- The role of `main` and its return value.
- Functions calling other functions, and recursion.

## CS1100 – Introduction to Programming

- Data Types in C, Operators. Input and the Output.
- Modifying the control flow in Programs if-else, switch, loops : while, do-while, for.
- Arrays and Strings in C.
- Basics of functions.

} Week 1-8

## CS1100 – Introduction to Programming

- Data Types in C, Operators. Input and the Output.
- Modifying the control flow in Programs if-else, switch, loops : while, do-while, for.
- Arrays and Strings in C.
- Basics of functions.

Week 1-8

- 
- Using functions to do "modular" programming - Examples.
  - recursive invocation of functions.
  - pass by reference in functions – later.

Week 9

## Blocks and Scope: Recap

**Block** : A program segment written within curly brackets.

**Scope** : The program segment where a particular declaration of a variable is applicable.

## Blocks and Scope: Recap

**Block** : A program segment written within curly brackets.

**Scope** : The program segment where a particular declaration of a variable is applicable.

```
int global;           // a global variable
int main()
{
    int local;        // a local variable

    global = 1;        // global can be used here
    local = 2;         // so can local

    {
        int very_local // beginning a new block
                        // this is local to the block

        very_local = global+local;

    }

    // We just closed the block
    // very_local can not be used
}
```

The diagram illustrates the scope of variables in the provided code. On the left, three vertical curly brackets indicate the scope of each variable:

- Scope of global:** This bracket spans the entire code block, from the first line to the final closing brace, indicating that the `global` variable is accessible throughout.
- Scope of local:** This bracket starts at the opening brace of the `main` function and ends at its closing brace, indicating that the `local` variable is only accessible within the `main` function.
- Scope of very\_local:** This bracket starts at the opening brace of the innermost block and ends at its closing brace, indicating that the `very_local` variable is only accessible within that specific block.

## Practicing the Concept : Blocks and Scope

```
int FindSum(int, int);
```

```
int var1 = 10;
```

```
int main()
```

```
{
```

```
    int sum,var2 = 20;
```

```
    {
```

```
        int var3;
```

```
        var3 = FindSum(var1,var2);
```

```
        printf("%d\n", var3);
```

```
    }
```

```
    float var3=100;
```

```
    printf("%d %f\n", sum,var3);
```

```
    return 0;
```

```
}
```

```
int FindSum(int a, int b)
```

```
{
```

```
    int sum=a+b;
```

```
    printf("%d\n",var1);
```

```
    return sum;
```

```
}
```

## Practicing the Concept : Blocks and Scope

```
int FindSum(int, int);
```

```
int var1 = 10;
```

```
int main()
```

```
{
```

```
    int sum, var2 = 20;
```

```
    {
```

```
        int var3;
```

```
        var3 = FindSum(var1, var2);
```

```
        printf("%d\n", var3);
```

```
    }
```

```
    float var3=100;
```

```
    printf("%d %f\n", sum, var3);
```

```
    return 0;
```

```
}
```

```
int FindSum(int a, int b)
```

```
{
```

```
    int sum=a+b;
```

```
    printf("%d\n", var1);
```

```
    return sum;
```

```
}
```

- Scope of var2 is the whole of main

## Practicing the Concept : Blocks and Scope

```
int FindSum(int, int);
int var1 = 10;

int main()
{
    int sum,var2 = 20;
    {
        int var3;
        var3 = FindSum(var1,var2);
        printf("%d\n", var3);
    }
    float var3=100;
    printf("%d %f\n", sum,var3);
    return 0;
}
```

```
int FindSum(int a, int b)
{
    int sum=a+b;
    printf("%d\n",var1);
    return sum;
}
```

- Scope of var2 is the whole of main
- Scope of int var3 is only the inner block.

## Practicing the Concept : Blocks and Scope

```
int FindSum(int, int);
int var1 = 10;

int main()
{
    int sum,var2 = 20;
    {
        int var3;
        var3 = FindSum(var1,var2);
        printf("%d\n", var3);
    }
    float var3=100;
    printf("%d %f\n", sum,var3);
    return 0;
}
```

```
int FindSum(int a, int b)
{
    int sum=a+b;
    printf("%d\n",var1);
    return sum;
}
```

- Scope of var2 is the whole of main
- Scope of int var3 is only the inner block.
- Scope of float var3 is only the outer block.

# Practicing the Concept : Blocks and Scope

```
int FindSum(int, int);
int var1 = 10;

int main()
{
    int sum,var2 = 20;
    {
        int var3;
        var3 = FindSum(var1,var2);
        printf("%d\n", var3);
    }
    float var3=100;
    printf("%d %f\n", sum,var3);
    return 0;
}
```

```
int FindSum(int a, int b)
{
    int sum=a+b;
    printf("%d\n",var1);
    return sum;
}
```

- Scope of var2 is the whole of main
- Scope of int var3 is only the inner block.
- Scope of float var3 is only the outer block.
- Scope of int var1 is the whole program.

# Practicing the Concept : Blocks and Scope

```
int FindSum(int, int);
```

```
int var1 = 10;
```

```
int main()
```

```
{
```

```
    int sum,var2 = 20;
```

```
    {
```

```
        int var3;
```

```
        var3 = FindSum(var1,var2);
```

```
        printf("%d\n", var3);
```

```
    }
```

```
    float var3=100;
```

```
    printf("%d %f\n", sum,var3);
```

```
    return 0;
```

```
}
```

```
int FindSum(int a, int b)
```

```
{
```

```
    int sum=a+b;
```

```
    printf("%d\n",var1);
```

```
    return sum;
```

```
}
```

- Scope of var2 is the whole of main
- Scope of int var3 is only the inner block.
- Scope of float var3 is only the outer block.
- Scope of int var1 is the whole program.

**Local vs Global** variables : var1 is global but var2 is local for main function.

# What does static do?

```
#include "stdio.h"
void DoSomething() {
    static int x=5;
    {
        static int y=6;
        x++;
        y++;
        printf ("x = %d y = %d\n", x, y);
    }
}
int main () {
    int i;
    for (i = 1; i < 10; i++)
        DoSomething();
}
```

- x and y are initialized only once, when execution first reaches their declarations.

# What does static do?

```
#include "stdio.h"
void DoSomething() {
    static int x=5;
    {
        static int y=6;
        x++;
        y++;
        printf ("x = %d y = %d\n", x, y);
    }
}
int main () {
    int i;
    for (i = 1; i < 10; i++)
        DoSomething();
}
```

- x and y are initialized only once, when execution first reaches their declarations.
- Their values remain stored after DoSomething() returns. The next call continues from the previous values.

# What does static do?

```
#include "stdio.h"
void DoSomething() {
    static int x=5;
    {
        static int y=6;
        x++;
        y++;
        printf ("x = %d y = %d\n", x, y);
    }
}
int main () {
    int i;
    for (i = 1; i < 10; i++)
        DoSomething();
}
```

- x and y are initialized only once, when execution first reaches their declarations.
- Their values remain stored after DoSomething() returns. The next call continues from the previous values.
- static changes the variable's lifetime, but not its scope: x remains local to the function and y to the inner block.

# What does static do?

```
#include "stdio.h"
void DoSomething() {
    static int x=5;
    {
        static int y=6;
        x++;
        y++;
        printf ("x = %d y = %d\n", x, y);
    }
}
int main () {
    int i;
    for (i = 1; i < 10; i++)
        DoSomething();
}
```

- x and y are initialized only once, when execution first reaches their declarations.
- Their values remain stored after DoSomething() returns. The next call continues from the previous values.
- static changes the variable's lifetime, but not its scope: x remains local to the function and y to the inner block.
- The function runs 9 times. The first output is x = 6 y = 7; the last is x = 14 y = 15.

# 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.
- Static local variables retain their values between function calls.

## Coming up :

- We will do hands-on examples of using functions.
- Is `main` program a function?  
Why are we ending with `"return 0;"` Who is it returning to?

## 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.
- Static local variables retain their values between function calls.

### Coming up :

- We will do hands-on examples of using functions.
- Is `main` program a function?  
Why are we ending with `"return 0;"` Who is it returning to?
- 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.
- Static local variables retain their values between function calls.

## Coming up :

- We will do hands-on examples of using functions.
- Is `main` program a function?  
Why are we ending with `"return 0;"` Who is it returning to?
- 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.
- Static local variables retain their values between function calls.

## Coming up :

- We will do hands-on examples of using functions.
- Is `main` program a function?  
Why are we ending with `"return 0;"` Who is it returning to?
- 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.
- Static local variables retain their values between function calls.

## Coming up :

- We will do hands-on examples of using functions.
- Is `main` program a function?  
Why are we ending with `"return 0;"` Who is it returning to?
- Can a function invoke other functions? **Yes !**
- Can a function invoke itself? **Yes! Recursion.**

## Hands-on Example 1 (Recap): Selection Sort

**Selection Sort:** Sort  $n$  numbers in descending order

**Pseudo-code :**

for  $i$  ranging from 1 to  $n$

- $\text{maxindex} =$  the index of the max element in the part of the array indexed from  $i$  to  $n$ . Find  $\text{maxindex}$ .
- swap elements  $\text{array}[i]$  and  $\text{array}[\text{maxindex}]$ ;

## Hands-on Example 1 (Recap): Selection Sort

**Selection Sort:** Sort  $n$  numbers in descending order

**Pseudo-code :**

for  $i$  ranging from 1 to  $n$

- $\text{maxindex}$  = the index of the max element in the part of the array indexed from  $i$  to  $n$ . Find  $\text{maxindex}$ .
- swap elements  $\text{array}[i]$  and  $\text{array}[\text{maxindex}]$ ;

**Subtasks identified:**

$\text{findmax}(i,n)$  : find the index of maxelement in the subarray from  $i$  to  $n$ .

$\text{swap}(i,j)$  : swap  $i^{\text{th}}$  and  $j^{\text{th}}$  elements of  $A$ .

# Hands-on Example 1 (Recap): Selection Sort

**Selection Sort:** Sort  $n$  numbers in descending order

**Pseudo-code :**

for  $i$  ranging from 1 to  $n$

- $\text{maxindex}$  = the index of the max element in the part of the array indexed from  $i$  to  $n$ . Find  $\text{maxindex}$ .
- swap elements  $\text{array}[i]$  and  $\text{array}[\text{maxindex}]$ ;

**Subtasks identified:**

$\text{findmax}(i,n)$  : find the index of maxelement in the subarray from  $i$  to  $n$ .

$\text{swap}(i,j)$  : swap  $i^{\text{th}}$  and  $j^{\text{th}}$  elements of  $A$ .

Once this is done (and solved), here is the remaining code.

```
for i = 1 to n
{
    max = findmax(i,n);
    swap(i,max);
}
```

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

|   |   |   |
|---|---|---|
| X | X | O |
|   |   | O |
| O | X |   |

- Two Player Game (X-player & O-player).

## 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.

## 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.

## 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.

|   |   |   |
|---|---|---|
| X | X | O |
|   | X | O |
| O | X |   |

## 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  $3 \times 3$  character array. Stores, 'X' or 'O' or Blank in each cell.

## Programming the Referee: functions

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

## Programming the Referee: functions

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

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

## Programming the Referee: functions

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

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

- Show the board to both players.

## Programming the Referee: functions

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

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  $3 \times 3$  character array. Stores, 'X' or 'O' or Blank in each cell.
- Define it as a global character array `board[3][3]` of order  $3 \times 3$ .

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  $3 \times 3$  character array. Stores, 'X' or 'O' or Blank in each cell.
- Define it as a global character array `board[3][3]` of order  $3 \times 3$ .

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  $3 \times 3$  character array. Stores, 'X' or 'O' or Blank in each cell.
- Define it as a global character array `board[3][3]` of order  $3 \times 3$ .

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");
    }
}
```