CS1100 – Introduction to Programming

Trimester 3, April – June 2021

Instructor:

Shweta Agrawal (shweta.a@cse.iitm.ac.in)

Lecture 21

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

X	X	O
		O
O	X	

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

Hands-on Example : 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 : 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: Four Functions:

We will do this using four functions:

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

Programming the Referee: 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.

Programming the Referee: 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.

Programming the Referee: 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 prorgam 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>

char board[1000][1000]; int N=3;
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 the `board[2][2]` be the character grid.
- Do the character search with `s = XXX` to determine if X-player wins.
- Do the character search with `s = OOO` to determine if O-player wins.

So we can reuse that code.

Implementing checkwin()

```
int checkwin()
{
  int i,j; int n=3;
  // 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-1) {
      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);
}
```

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]`.

Reversing an Array: Using Auxiliary Array

```
#include <stdio.h>
 void print(int arr[], int n)
{
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
}
 void reverse(int arr[], int n)
{
    int aux[n];

    for (int i = 0; i < n; i++) {
        aux[n - 1 - i] = arr[i];
    }

    for (int i = 0; i < n; i++) {
        arr[i] = aux[i];
    }
}

int main(void)
{
    int arr[] = { 1, 2, 3, 4, 5 };
    int n = sizeof(arr)/sizeof(arr[0]);

    reverse(arr, n);
    print(arr, n);

    return 0;
}
```

Reversing an Array: In Place

```c
#include <stdio.h>

void print(int arr[], int n)
{
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
}
 void reverse(int arr[], int n)
{
    for (int low = 0, high = n - 1; low < high; low++, high--)
    {
        int temp = arr[low];
        arr[low] = arr[high];
        arr[high] = temp;
    }
}

int main(void)
{
    int arr[] = { 1, 2, 3, 4, 5 };
    int n = sizeof(arr)/sizeof(arr[0]);

    reverse(arr, n);
    print(arr, n);

    return 0;
}
```

Macros in C

- A macro is a fragment of code that is given a name. You can define a macro in C using the #define preprocessor directive. Example # define c 299792458 (speed of light)

Macros in C

- A macro is a fragment of code that is given a name. You can define a macro in C using the #define preprocessor directive. Example # define c 299792458 (speed of light)
- By default, of type integer. Can change datatype by adding suffixes: 123456789*L* is a long constant, 123456789ul is an unsigned long constant etc.

Macros in C

- A macro is a fragment of code that is given a name. You can define a macro in C using the #define preprocessor directive. Example # define c 299792458 (speed of light)
- By default, of type integer. Can change datatype by adding suffixes: 123456789*L* is a long constant, 123456789ul is an unsigned long constant etc.

```
#include <stdio.h>
#define PI 3.1415

int main()
{
    float radius, area;
    printf("Enter the radius: ");
    scanf("%f", &radius);

    // Notice, the use of PI
    area = PI*radius*radius;

    printf("Area=%.2f",area);
    return 0;
}
```

Enumerated Constants

- Macros let us define a single constant at a time. What if we want to define many?

Enumerated Constants

- Macros let us define a single constant at a time. What if we want to define many?
- Declaration: `enum boolean {No, Yes};` defines two constants No = 0, and Yes = 1.

Enumerated Constants

- Macros let us define a single constant at a time. What if we want to define many?
- Declaration: enum boolean {No, Yes}; defines two constants No = 0, and Yes = 1.
- enum months {jan = 1, feb, march, april, may, jun, jul, aug, sep, oct, nov, dec};

Enumerated Constants

- Macros let us define a single constant at a time. What if we want to define many?
- Declaration: enum boolean {No, Yes}; defines two constants No = 0, and Yes = 1.
- enum months {jan = 1, feb, march, april, may, jun, jul, aug, sep, oct, nov, dec};
- When a value is explicitly specified (jan=1) then it starts counting from there

Enumerated Constants

- Macros let us define a single constant at a time. What if we want to define many?
- Declaration: enum boolean {No, Yes}; defines two constants No = 0, and Yes = 1.
- enum months {jan = 1, feb, march, april, may, jun, jul, aug, sep, oct, nov, dec};
- When a value is explicitly specified (jan=1) then it starts counting from there
- Values start from 0 unless specified otherwise.

Enumerated Constants

- Macros let us define a single constant at a time. What if we want to define many?
- Declaration: enum boolean {No, Yes}; defines two constants No = 0, and Yes = 1.
- enum months {jan = 1, feb, march, april, may, jun, jul, aug, sep, oct, nov, dec};
- When a value is explicitly specified (jan=1) then it starts counting from there
- Values start from 0 unless specified otherwise.
- Not all values need to be specified. If some values are not specified, they are obtained by increments from the last specified value.

Enumerated Constants

- Macros let us define a single constant at a time. What if we want to define many?
- Declaration: enum boolean {No, Yes}; defines two constants No = 0, and Yes = 1.
- enum months {jan = 1, feb, march, april, may, jun, jul, aug, sep, oct, nov, dec};
- When a value is explicitly specified (jan=1) then it starts counting from there
- Values start from 0 unless specified otherwise.
- Not all values need to be specified. If some values are not specified, they are obtained by increments from the last specified value.
- Better than #define, as the constant values are generated for us.

Enumerated Constants

```
#include <stdio.h>

enum week {Sun, Mon, Tue, Wed, Thur, Fri, Sat};

int main()
{
    // creating today variable of enum week type
    enum week today;
    today = Wed;
    printf("Day %d",today+1);
    return 0;
}
```

Output is: Day 4.

- Note that the variable values are treated as integers though they look like strings!
- In the program, can use $Wed > 0$ etc. Wed will be treated as an (unisgned) integer.

Enumerated Constants

```
#include <stdio.h>

enum escapes {BELL = ’\a’, BACKSPACE = ’\b’, TAB = ’\t’, NEWLINE =’\n’}

int main()
{
    // creating today variable of enum week type
    enum escapes element;
    element = BELL;
    printf("We have %d",element);
    return 0;
}
```

Output is: We have 7.

Declaring Constants

- The qualifier const applied to a declaration specifies that the value will not be changed.

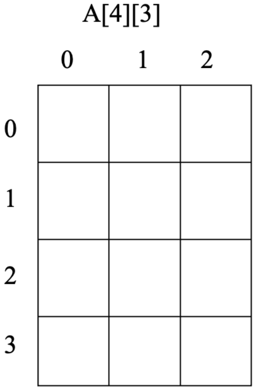
Declaring Constants

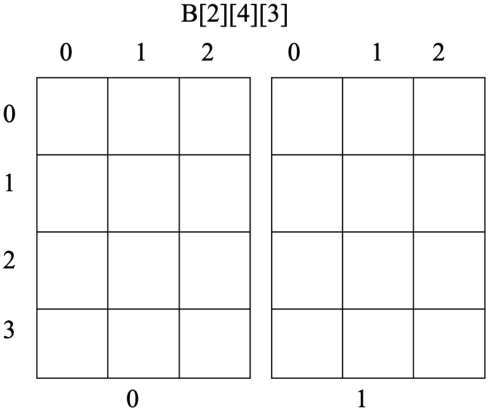
- The qualifier const applied to a declaration specifies that the value will not be changed.
- If I declare const int J = 25; , this means that J is a constant throughout the program.

Declaring Constants

- The qualifier const applied to a declaration specifies that the value will not be changed.
- If I declare const int J = 25; , this means that J is a constant throughout the program.
- Response to modifying J depends on the system. Typically, a warning message is issued while compilation.

Multi-Dimensional Arrays





Storage and Initialization are row by row

Multi-Dimensional Arrays

- double array3d[100][50][75];

Multi-Dimensional Arrays

- double array3d[100][50][75];
- double array4d[60][100][50][75];
  Requires 60*100*50*75*8 = 171.66 MB!

Multi-Dimensional Arrays

- double array3d[100][50][75];
- double array4d[60][100][50][75];
  Requires 60*100*50*75*8 = 171.66 MB!
- Find out how many dimensions your system/compiler can handle.

Initializing 2D Arrays

- int a[3][2] $= \{\{1,4\},\{5,2\},\{6,5\}\}$;
  Recommended that each value is initialized explicitly.

Initializing 2D Arrays

- int a[3][2] $= \{\{1, 4\}, \{5, 2\}, \{6, 5\}\}$;
  Recommended that each value is initialized explicitly.
- int a[3][2] $= \{1, 4, 5, 2, 6, 5\}$;
  Stored in row major order (better not to assume).

Initializing 2D Arrays

- int a[3][2] = {{1, 4}, {5, 2}, {6, 5}};
  Recommended that each value is initialized explicitly.
- int a[3][2] = {1, 4, 5, 2, 6, 5};
  Stored in row major order (better not to assume).
- int a[3][2] = {{1}, {5, 2}, {6}}; Some elements are not initialized explicitly – they are initialized to 0.

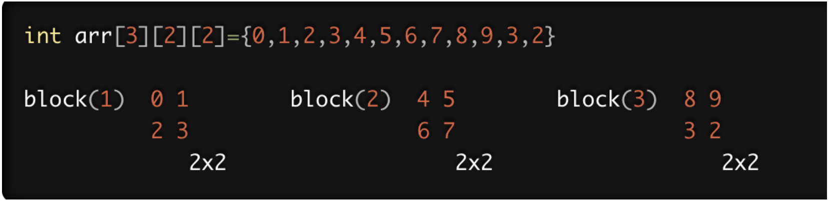
Initializing 2D Arrays

- int a[3][2] $= \{\{1, 4\}, \{5, 2\}, \{6, 5\}\}$;
  Recommended that each value is initialized explicitly.
- int a[3][2] $= \{1, 4, 5, 2, 6, 5\}$;
  Stored in row major order (better not to assume).
- int a[3][2] $= \{\{1\}, \{5, 2\}, \{6\}\}$; Some elements are not initialized explicitly – they are initialized to 0.
- a[0][1] $= 0; a[2][1] = 0$;

Initializing 2D Arrays

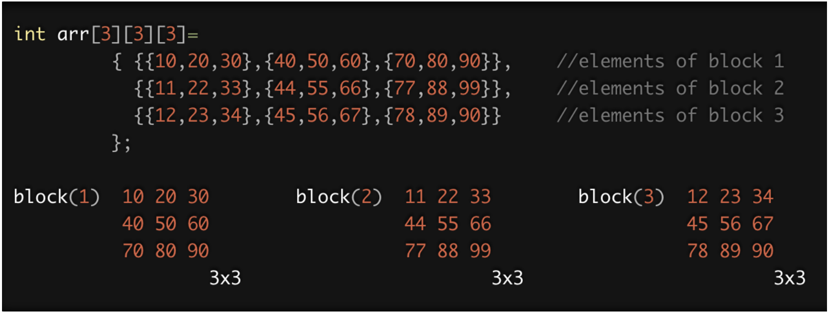
- int a[3][2] = {{1, 4}, {5, 2}, {6, 5}};
  Recommended that each value is initialized explicitly.
- int a[3][2] = {1, 4, 5, 2, 6, 5};
  Stored in row major order (better not to assume).
- int a[3][2] = {{1}, {5, 2}, {6}}; Some elements are not initialized explicitly – they are initialized to 0.
- a[0][1] = 0; a[2][1] = 0;
- Better not to assume!

Initializing 3D Arrays: Block by Block!



```
int arr[3][2][2]={0,1,2,3,4,5,6,7,8,9,3,2}

block(1)  0 1          block(2)  4 5          block(3)  8 9
          2 3                    6 7                    3 2
             2x2                    2x2                    2x2
```



```
int arr[3][3][3]=
        { {{10,20,30},{40,50,60},{70,80,90}},     //elements of block 1
          {{11,22,33},{44,55,66},{77,88,99}},     //elements of block 2
          {{12,23,34},{45,56,67},{78,89,90}}      //elements of block 3
        };

block(1)  10 20 30         block(2)  11 22 33         block(3)  12 23 34
          40 50 60                   44 55 66                   45 56 67
          70 80 90                   77 88 99                   78 89 90
                  3x3                        3x3                        3x3
```