Lecture 10

A Magic Square is a square filled with numbers where every row
and every column and each of the two diagonals all add up to the same value.
Here is an example of a 3X3 magic square

4	9	2
3	5	7
8	1	6

To build a 3X3 magic square we can use the following method.

Start in the middle of the top row and put your first number in there

0 0 0

0 0 0

0 1 0

Then, until you have inserted all 9 numbers do the following steps

(1). move on to the next number to insert move down one row and move right one column

(2). If you are on a square which just contains a zero then write in the next number

(3). If you are on a square which already contains a number you have inserted then move up a row.

At any stage, if you have moved off the top of the square just wrap around to the bottom of that column or if you have moved off the right of the square just wrap around to the start of that row.

0 0 2

0 0 0

0 1 0

0 0 2

3 0 0

0 1 0

```
void main()
{
     int magic3 [3] [3] ;
     int i ;
     int j ;
     int c ;
     int n ;

// initialize magic3

     i= 0 ;
    while ( i != 3)
{
      j =0 ;
     while ( j != 3)
     {
           magic3 [ i ] [ j ] = 0 ;
           j = j+1;
      }
     i = i+1;
  }
```

```
printf(“please enter starting number for 3X3 magic square : “) ;
scanf(“%d”, n) ;
```

```
// populate magic3
   i= 2 ;
   j= 1 ;
  c= n ;
  while ( c != n+9)
  {
       magic [ i ] [ j ] = c ;
       if ( magic [ (i+1) % 3 ] [ (j+1) % 3 ] != 0 )
           {
                i = (i-1) % 3 ;
        }
     else
        {
                i = (i+1) % 3 ;
                j = (j+1) % 3 ;
        }
     c = c+1 ;
}
```

Structures

So far we have looked at simple variables
which record values. However, we often want to
group a few different items of data together.

When we were playing with date problems we wanted to record a day,
a month and a year. At that time we used 3 int variables. It might be useful
to link a particular day and a particular month and a particular year together.

e.g. someone's date of birth, someone's graduation date etc…

C provides us with a way to group values together. It provides us with
Structures.

A structure to represent a date

```
struct Date {
                int day ;
                int month ;
                int year ;
              } ;
```

We use the “keyword” **struct**
to indicate that what follows
will be the definition of a structure.

In this case Date is the name of the structure.

within { } we declare the parts of the structure called Date

Finally we finish with a “;”

Definitions such as these should happen
outside of any functions in the program.

You should put them before the start of the main () function

```
#include <stdio.h>

struct Date
        {
        int day ;
        int month ;
        int year ;
        } ;

int main()
{
// your program
}
```

Now, within the main program
we can declare a variable to be a Date.

To declare a Date called today and to give it values we do this

```
int main()
{
        struct Date today ;
        today.day = 7 ;
        today.month = 5 ;
        today.year = 2018 ;

        printf("%d – %d – %d", today.day, today.month, today.year )

}
```

```
struct Date today ;
```

Once again we use the “keyword” struct

We give the name of the particular type of structure, in this case it is a Date structure.

Then we name the variable which we are declaring.

To access the parts of today and give them values

we use the name of the variable followed by a “.”
followed by the name of the particular part.

```
today.day = 7 ;
today.month = 5 ;
today.year = 2018 ;
```

Suppose we wanted to record some facts
about a small group of students. We are interested
in storing a student's age and their height.

We could declare a structure called Person as follows.

```
struct Person
                {
                int age ;
                int height ;
                } ;
```

Within our program we could then declare an array of this type and read values into it.

```
int main()
{
        struct Person student [5] ;
        int i ;

        i = 0 ;
        while (i != 5)
        {
                printf("Please enter age for student %d : ", i);
                scanf("%d", &student[i].age);
                printf("Please enter height for student %d : ", i);
                scanf("%d", &student[i].height);
                i = i+1;
        }
}
```

Now, suppose I wanted to know what age the oldest student was.

I propose that we write a function to compute this

```
int oldest (struct Person f[], int size)
{
        int max ;
        int j ;
        max = f[0].age ;
        while (j != size)
        {
                if (f[j].age > max)
                        max = f[j].age ;
                j = j+1 ;
        }
        return max ;
}
```

In my main program I can now use this

```
printf("oldest is %d : " , oldest (student, 5));
```

Because student is an array we pass it across to the function like we saw in the last lecture.

Of course this is really passing a pointer to the location containing the first entry in the array.

We also pass across the size of the array so we know when to stop accessing values in it.

Suppose we wanted to determine
the average height of the students.

```
float average_height (struct Person f[], int size)
{
        int total ;
        int j ;
        float average ;
        total = 0 ;
        j = 0 ;
        while (j != size)
        {
                total = total + f[j].height ;
                j = j+1 ;
        }
        average = total / size ;
        return average ;
}
```

Suppose we have declared the following structure

```
struct Date
        {
        int day ;
        int month ;
        int year ;
        } ;
```

We now want to record details about employees in our company.

We want to record their name, their staff number, their salary
and the date they joined the company.

We are going to declare a structure to group these together

```
struct Employee
                {
                char name [20] ;
                int staff_id ;
                float salary ;
                struct Date start_date ;
                } ;
```

Here the start_date is a variable which is itself a structured type.

Suppose in our main program we now declare a variable

```
struct Employee employee1 ;
```

The different parts can be accessed as before

employee1.name
employee1.staff_id
employee1.salary

So how do we access the start_date?

employee1.start_date.day

employee1.start_date.month

employee1.start_date.year

We can declare and initialise variables with values

e.g.

int x = 10 ;

We can do similar things with structure variables

```
struct Employee employee1 = {"Henry", 101, 34500.50, {22, 5, 1985}};
```

We use functions to structure our programs so that

different programming tasks are kept together.

this allows us to reuse functions in other places.

we can write programs that are easy to read and understand.

Structures play a similar role but they structure the data rather than the program instructions.

Structuring data is very important when you want to represent things in the world.

Suppose we have declared the following structure

```
struct Date
        {
        int day ;
        int month ;
        int year ;
        } ;
```

In our main program we can declare an array of such dates

```
int main()
{

        struct Date dates[5] = {{1,1,2018},
                                {1,3,2018},
                                {12,4,2018},
                                {23,11,2018},
                                {3,4,2019}} ;
```

```
void display ( struct Date d[], int size )
{
        int i ;

        i = 0 ;
        while (i != size)
        {
                printf("date %d : %d – %d – %d \n”,
                                                i ,
                                                d[i].day,
                                                d[i].month,
                                                d[i].year ) ;
                i = i+1 ;
        }
}
```

This function displays the contents of the array of Date. Remember that what will be passed to the function is a pointer to the start of the array.

Suppose we wanted to change one of the dates.

We could write a function like this

```
void alter ( struct Date d[], int size )
{
        int i ;
        printf("which date do you want to change? : \n ") ;
        scanf("%d", &i);

        while ((i < 0) || (i >= size))
        {
                printf("date out of range, please enter again \n") ;
                scanf("%d", &i) ;
        }

        printf("enter new day : \n") ;
        scanf("%d", &d[i].day );
        printf("enter new month : \n") ;
        scanf("%d", &d[i].month );
        printf("enter new year : \n") ;
        scanf("%d", &d[i].year );

}
```

When we pass an array as an argument to a function we are passing a pointer to the first element in the array. Because we are dealing with the actual array and not a copy of it we must be careful as any changes we make to it will change the array in the main program.

If we pass an element of an array to a function we are passing a copy of that data and changes made within the function will not change the array element in the main program.

Here is a function which takes an element of
our array of dates and displays it.

```
void display1 (struct Date d )
{
    printf("The date you called the function with is %d – %d – %d \n",
                        d.day, d.month, d.year) ;
}
```

In our main program we would call this function like this

```
        display1( dates[2]) ;
```

The contents of dates[2] will be copied and passed in to the function.

We could have passed a pointer to that element in the array.
But now we must refer to the parts of the structure in a different way.

```
void display1 ( struct Date *d )
{
        printf("The date is %d - %d - %d \n",
                        d->day, d->month, d->year) ;
}
```

Here we pass in a pointer to a structure of type Date.
The elements of the structure are accessed using this arrow notation “->” and not the dot “.” used previously.

In the main program we call the function like this

```
        display1( &dates[2] ) ;
```

Because we pass in a pointer to the array element we must be careful as any changes to the structure will change the array in the main program.

If we wanted to have a function that changed
one of the dates in the array we could do so like this.

```
void alter1 (struct Date *d)
{
        printf("enter new day : \n") ;
        scanf("%d", &d->day );
        printf("enter new month : \n") ;
        scanf("%d", &d->month );
        printf("enter new year : \n") ;
        scanf("%d", &d->year );
}
```

We would call this as follows in the main program

```
        alter1 (&dates[0]) ;
```

Data validation

- So far we have always assumed that if we ask the user to enter some data they will enter some valid data.

- However, when dealing with users it is best to assume that if they can do something wrong they will do so.

- Garbage in, garbage out. If you give the program rubbish then expect it to produce rubbish.

Data validation

- A lot of effort in programming must be spent writing code to validate the user's input.

- If you write functions to do data validation, try to make them general so you can reuse them.

Suppose we have an array int f[10] and we want to
ask the user to select one of the locations in the array.

The user should input a value between 0 and 9.

If the input a value outside that range then we would not
want to try to access that array location.

But, suppose the array was like this int g[100]
The size has changed and so has the range of valid values,
but the core of the problem is really the same.

Is the value that they input somewhere between a lower bound
and an upper bound.

```
int in_range (int value, int lower, int upper)
{
   if ((value < lower) || (value > upper))
      { return 0 ; }
   else
      { return 1 ; }
}
```

Suppose we wanted to input a date as 3 values
d, m and y. Let us write a function to check if
the values represent a valid date

```
int valid_date (int d, int m, int y )
{
   If ((d < 1) || (d > maxdays (m, y))
      { return 0 ; }
   else if ((m < 1) || ( m > 12 ))
      { return 0 ; }
   else
      { return 1 ; }
}
```

Typically we would perform data validation like this.
Suppose you wanted to get a valid date

```
printf(“Please enter a valid date as d m y : ) ;
scanf(“%d”, &d ) ;
scanf(“%d”, &m ) ;
scanf(“%d”, &y ) ;
date_ok = valid_date(d, m, y) ;
While (! date_ok)
{
   printf(“You entered an invalid date \n) ;
   printf(“Please enter a valid date as d m y : ) ;
   scanf(“%d”, &d ) ;
   scanf(“%d”, &m ) ;
   scanf(“%d”, &y ) ;
   date_ok = valid_date(d, m, y) ;
   }
// d, m, y contain a valid date
```

Similarly, if we want an index in f[1]

```
printf(“Please enter a value in the range 0 through 9 : “ ) ;
scanf(“%d”, &i );
is_valid = in_range ( i, 0, 9) ;
while ( !is_valid )
{
   printf(“You entered a value which is out of the range \n” ) ;
   printf(“Please enter a value in the range 0 through 9 : “ ) ;
   scanf(“%d”, &i );
   is_valid = in_range ( i, 0, 9) ;
}
```

The general shape is like this

“get data “
“check if it is valid “

While (“data invalid “)
{
  “tell user data is invalid”
  “get data”
  “check if it is valid”
}

Of course, if the user continues to enter invalid data then
this while loop will keep going.

Often we want to give the user a limited number of chances
to enter data and if they exceed the limit we want to do something else

```
int attempts = 0 ;
int too_many ;

“get data “
“check if it is valid “

While ( (“data invalid “) && (attempts != too_many) )
{
   “tell user data is invalid”
   attempts = attempts + 1 ;
   “get data”
   “check if it is valid”
}

// “data is valid” || attempts == too_many
```

Date problems again

- We have done some basic date problems..
- Determine if a year is leap
- Compute tomorrow’s date
- Compute yesterday’s date
- Let us look at some more

Date problems again

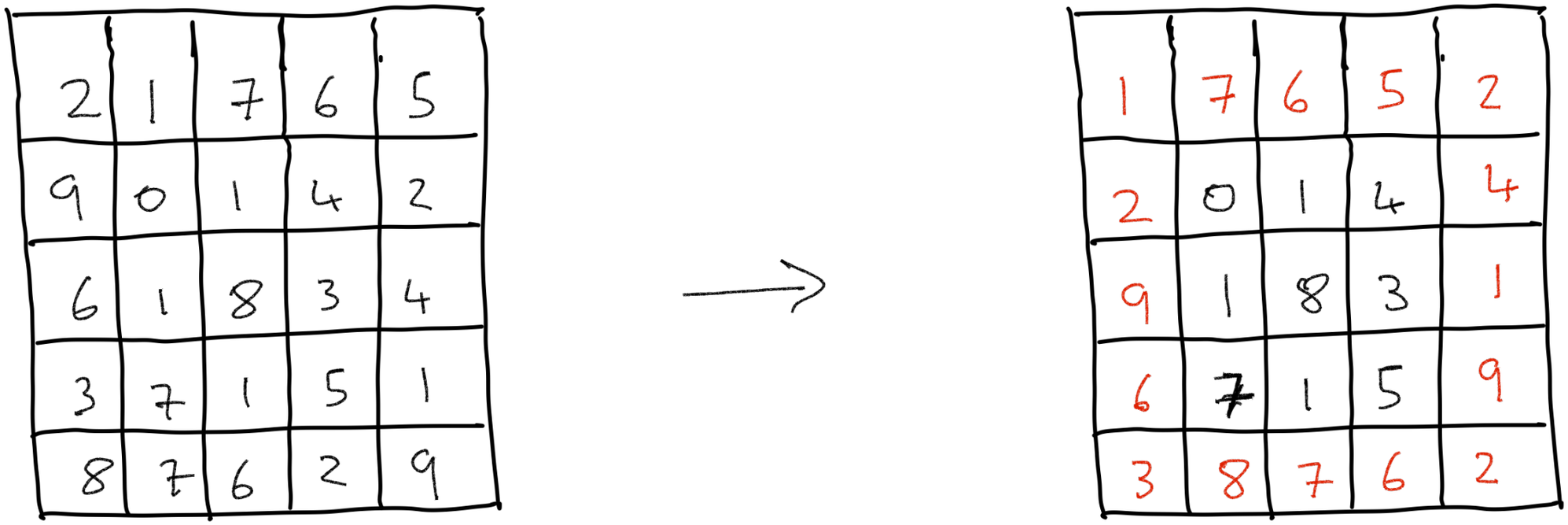
- Given 2 dates d1/m1/y1 and d2/m2/y2 determine if the second date is after the first one.

- How would you represent the dates? Perhaps using a structure?

- Given todays date, what date will it be in 2 weeks from now? Or in 5 weeks from now?

Assignment 3

- Write a program to read values into an array int f [5] [5]
- The program should then “rotate” the values on the outside of the array by 1 place anti-clockwise
- The program should display the array both before and after doing the rotation.



2	1	7	6	5
9	0	1	4	2
6	1	8	3	4
3	7	1	5	1
8	7	6	2	9

1	7	6	5	2
2	0	1	4	4
9	1	8	3	1
6	~~7~~	1	5	9
3	8	7	6	2

A Challenge

- Write numbers in the following sentence so the sentence becomes true. This sentence has __ zeros, __ ones, __ twos, __ threes, __ fours, __ fives, __ sixes, __ sevens, __ eights, and __ nines.