UESTC1005 - Introductory Programming

Multiple Dimensions Arrays, Preprocessors, and Strings

Week 10 | Lecture 08

Dr Ahmed Zoha

Lecturer in Communication Systems

School of Engineering

University of Glasgow

Email**: **ahmed.zoha@glasgow.ac.uk

Multiple Dimensions Arrays

Array size in the 1st dimension

- C programming language allows multidimensional arrays

Array name

Array size in the Nth dimension

- Here is the general form of a multidimensional array declaration:

Array type

```
arrayType arrayName[size1][size2] ... [sizeN]
```

- The simplest form of the multidimensional array is the two-dimensional array. To declare a two-dimensional array of size x, y you would write something as follows:

Array size in the 2nd dimension

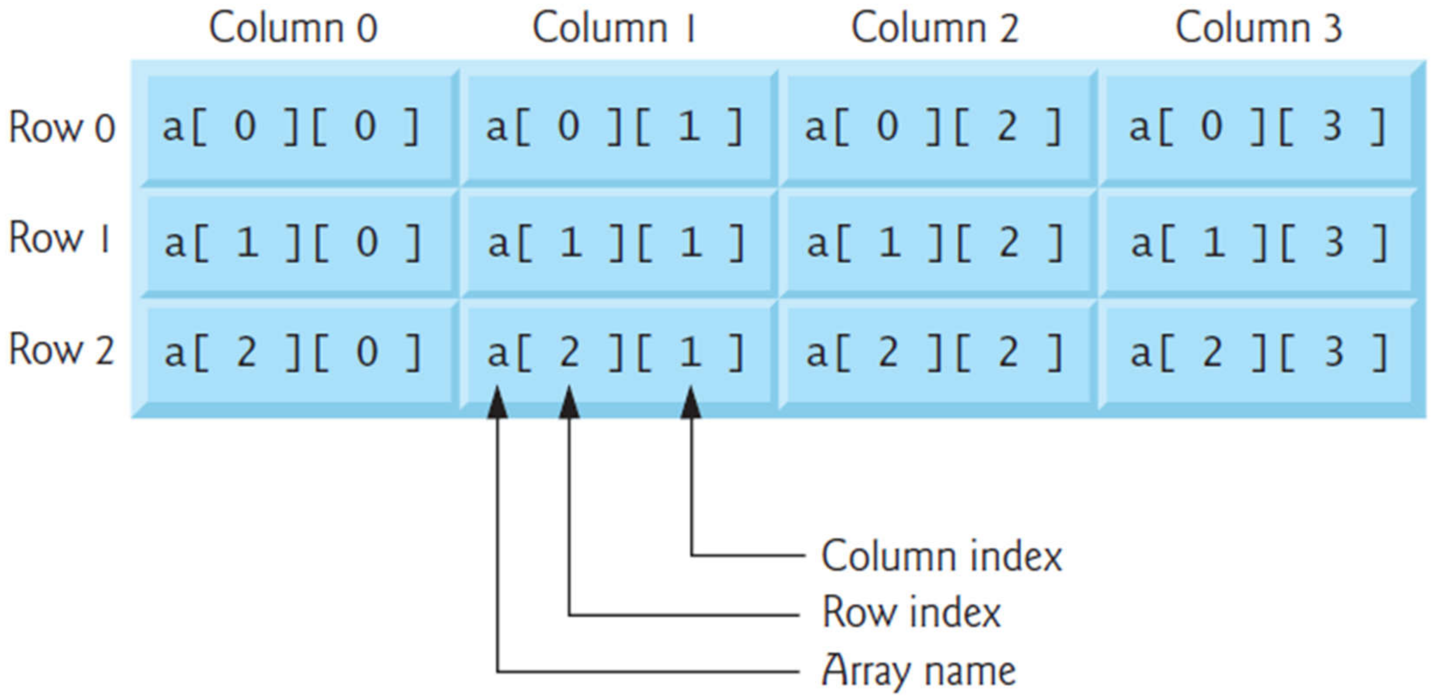
```
arrayType arrayName[x][y]
```

  – We call it a *x-by-y* array

- Where `arrayType` can be any valid C data type and `arrayName` will be a valid C identifier -- the same as 1D array and scalar variables
- A two-dimensional array can be think as a table which will have x number of rows and y number of columns

Multiple Dimensions

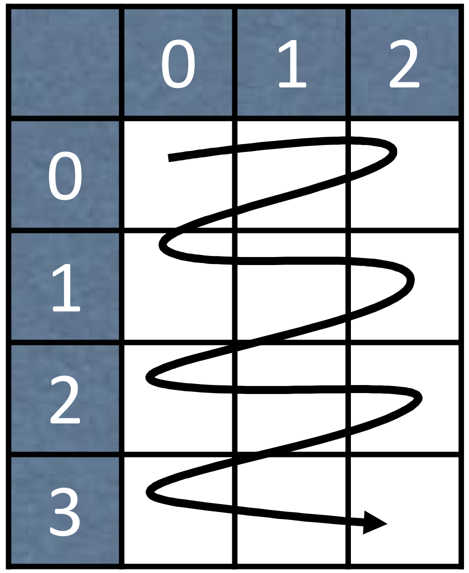
3x4 Array,
e.g., int a[3][4]; → in total 12 elements



Two Dimensions

- Arrays with more than one dimension can be defined in the following way :

```
int myArray[4][3]    // declare 4 rows and 3 columns

x = myArray[i][j]    // element from ith row, jth col
```



Remember to count from 0

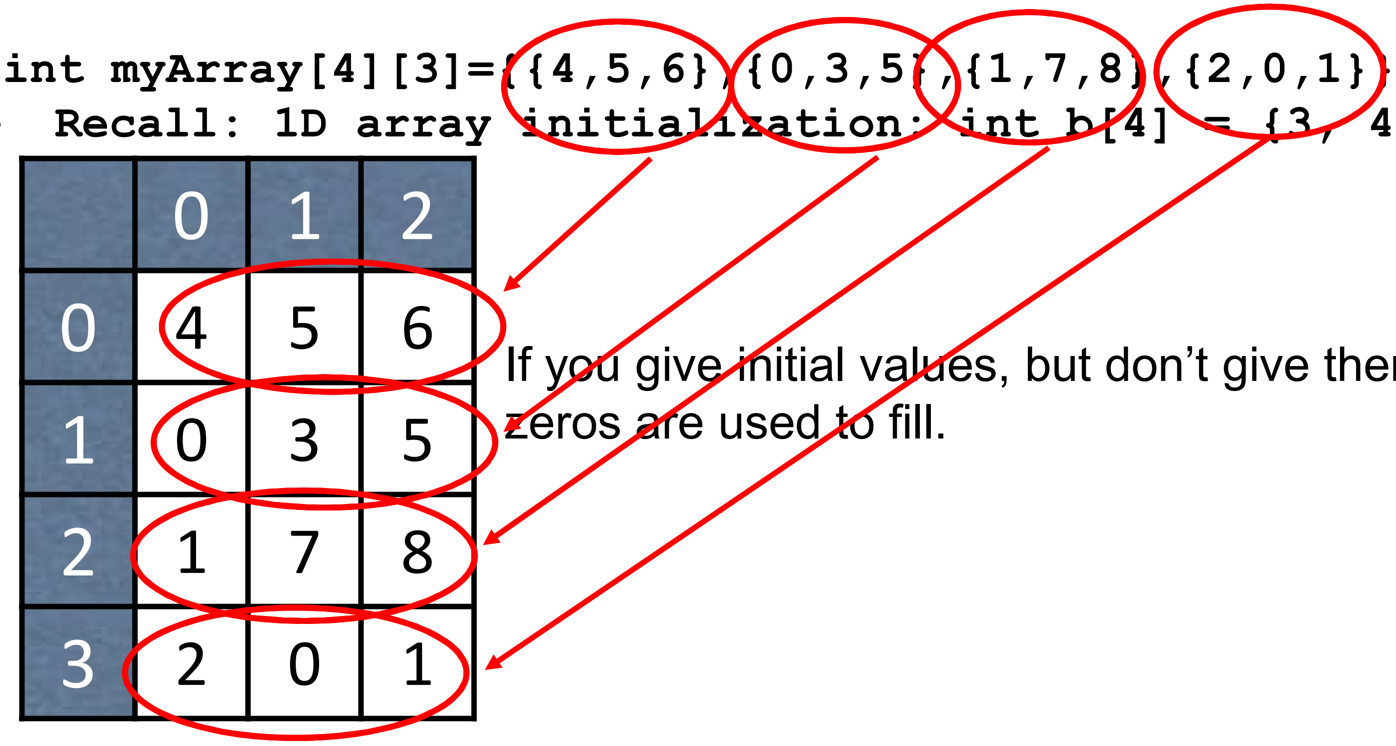
Storage in C is row-major ordering; store row 0, then row 1, etc.

Two Dimensions

- Arrays can be declared and given values :

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

– Recall: 1D array initialization: `int b[4] = {3, 4, 1, 4};`



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

If you give initial values, but don't give them all, zeros are used to fill.

This is how they sit in memory… (each int occupies 4 bytes)

&myArray	&myArray+4	+8	12	16	20	24	28	32	36	40	44
4	5	6	0	3	5	1	7	8	2	0	1

Two Dimensions

- Initialization
  - `int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };`
  - Initializers grouped by row in braces

1	2
3	4

  - If not enough, unspecified elements set to zero

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

1	0
3	4

- Referencing elements
  - Specify row, then column

```
printf( "%d", b[ 0 ][ 1 ] );
```

Designated Initializers

- Subscripts can be used in the initialization list, similar to 1-D arrays.

```
int matrix[4][3] = {[0][0]=1,[1][1]=5, [2][2]=9};
```

All the unspecified elements are set to 0

Each set of values that initializes the elements are between curly braces

Multiple Dimensions

- Arrays with more than one dimension can be defined in the following way :

```
int myArray[4][3]
```

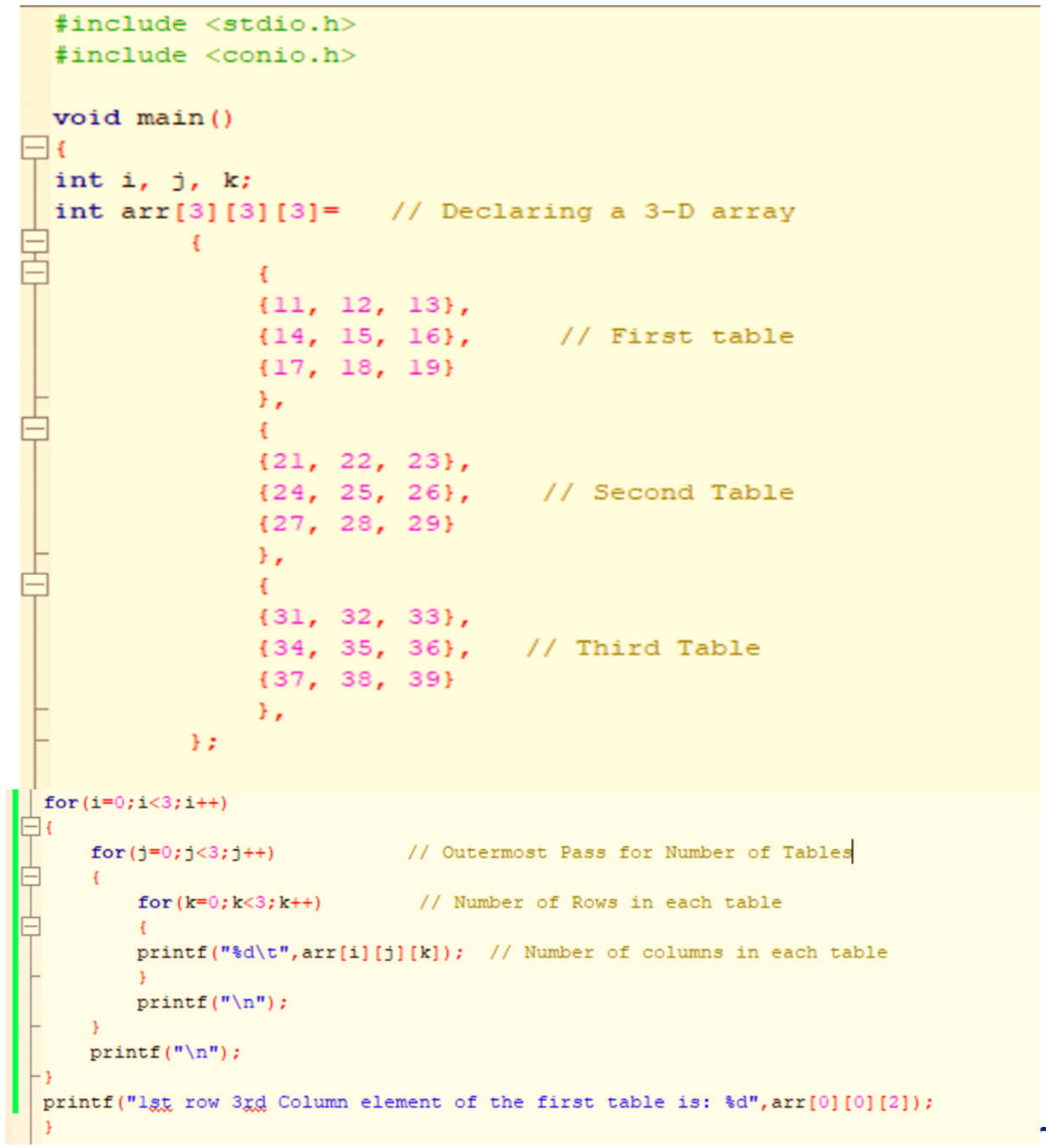
	0	1	2
0			
1			
2			
3			

```
int myArray[2][4][3]
```

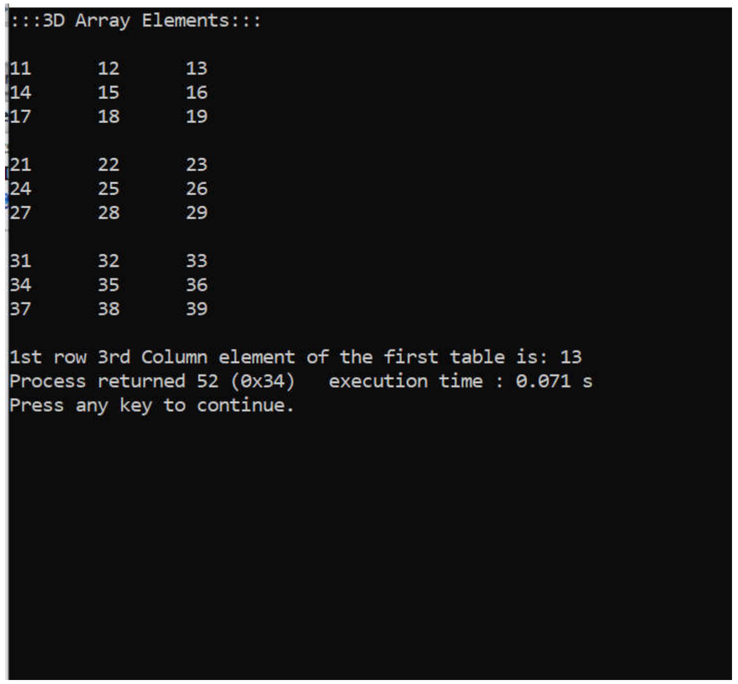
0	0	1	2
0			
1			
2			
3			

1	0	1	2
0			
1			
2			
3			

Ex 1: Initializing and Declaring a 3-D Array



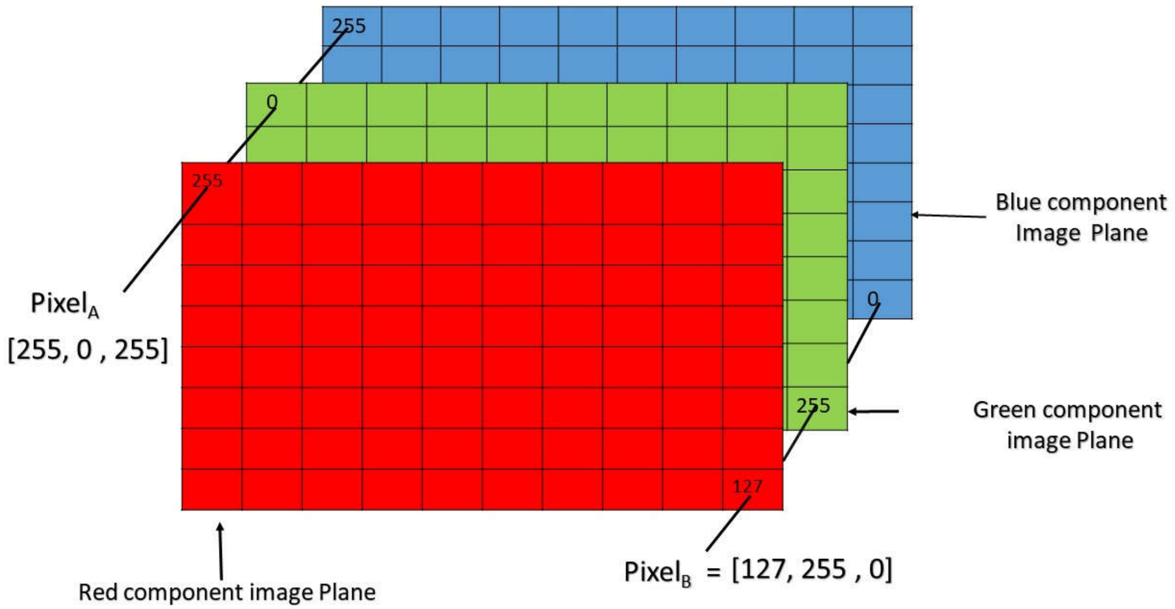
```c
#include <stdio.h>
#include <conio.h>

void main()
{
int i, j, k;
int arr[3][3][3]=    // Declaring a 3-D array
        {
            {
            {11, 12, 13},
            {14, 15, 16},       // First table
            {17, 18, 19}
            },
            {
            {21, 22, 23},
            {24, 25, 26},      // Second Table
            {27, 28, 29}
            },
            {
            {31, 32, 33},
            {34, 35, 36},     // Third Table
            {37, 38, 39}
            },
        };
for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)            // Outermost Pass for Number of Tables
    {
        for(k=0;k<3;k++)          // Number of Rows in each table
        {
        printf("%d\t",arr[i][j][k]);  // Number of columns in each table
        }
        printf("\n");
    }
    printf("\n");
}
printf("1st row 3rd Column element of the first table is: %d",arr[0][0][2]);
}
```



```
:::3D Array Elements:::

11      12      13
14      15      16
17      18      19

21      22      23
24      25      26
27      28      29

31      32      33
34      35      36
37      38      39

1st row 3rd Column element of the first table is: 13
Process returned 52 (0x34)   execution time : 0.071 s
Press any key to continue.
```

Multi dimensional arrays in Real World

- Multi dimensional arrays real-world applications: Audio and Image


255
0
255
Pixel$_A$
[255, 0 , 255]
Blue component Image Plane
0
255
Green component image Plane
127
Pixel$_B$ = [127, 255 , 0]
Red component image Plane

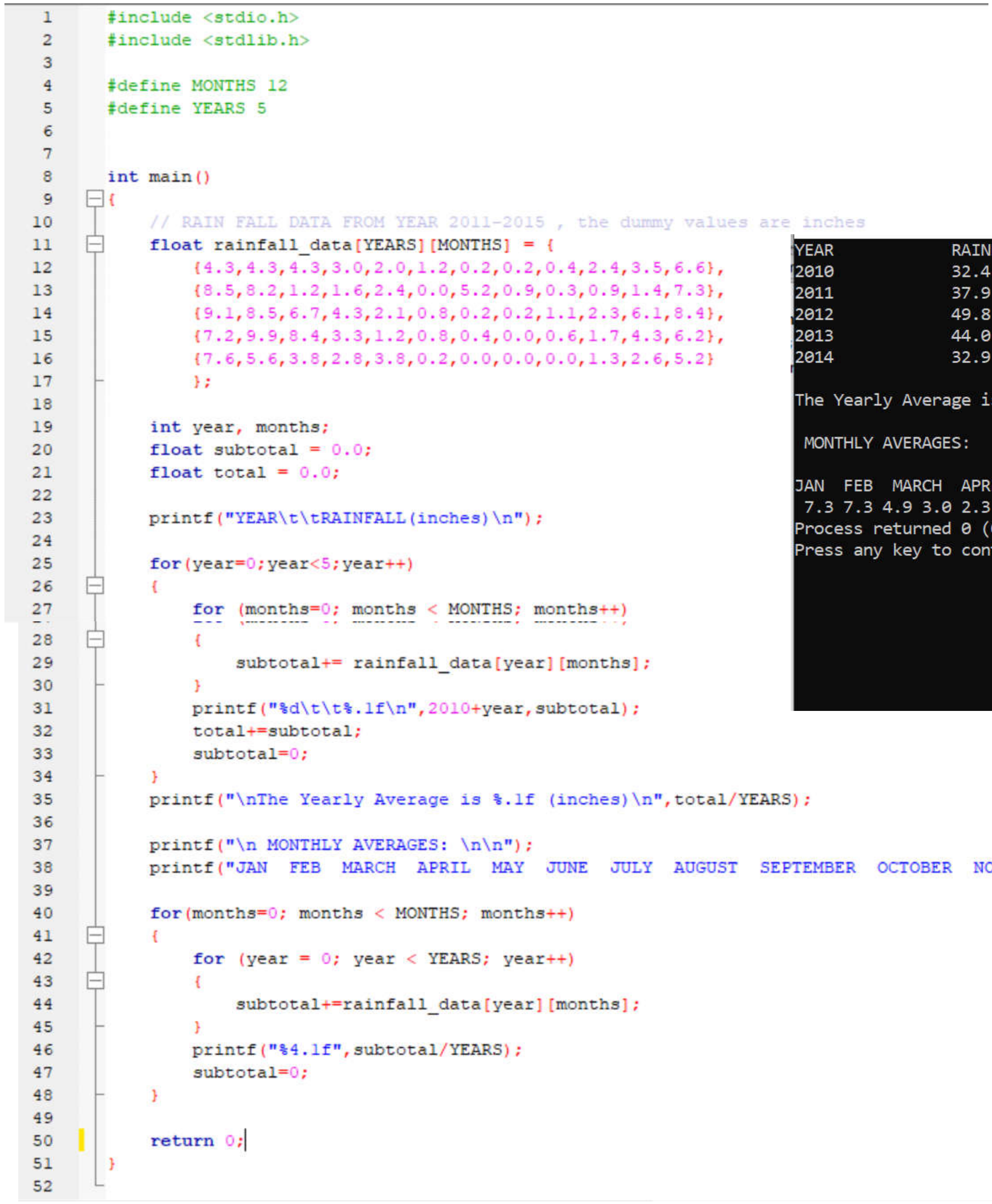
Pixel of an RGB image are formed from the corresponding pixel of the three component images

Each pixel has a position(0,0,0)….(0,183,154), stored in RGB

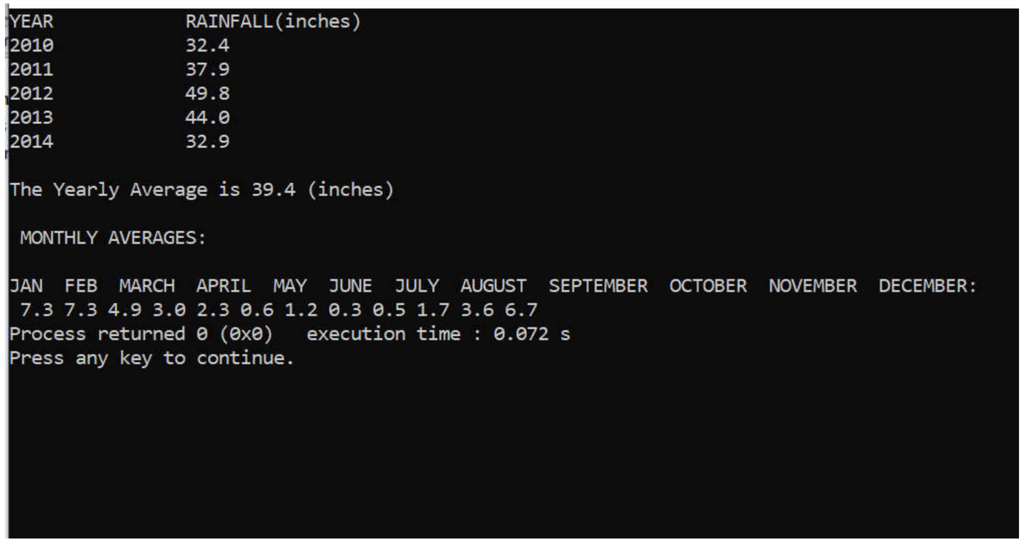
Ex 2: Creating a Weather Program (1/2)

- This program will find the total rain fall each year, the average yearly rainfall, and the average rainfall for each month.
- Input will be a 2D array with hard-coded values for rain falls amounts to last 5 years
  - The arrays should have 5 rows and 12 columns
  - Rainfall amounts can be floating point numbers
- Output for the program is
  - Display the year and rainfall for all of the months
  - You should display the yearly average value
  - Also you should display the monthly averages

Weather Program (2/2)



```c
#include <stdio.h>
#include <stdlib.h>

#define MONTHS 12
#define YEARS 5


int main()
{
    // RAIN FALL DATA FROM YEAR 2011-2015 , the dummy values are inches
    float rainfall_data[YEARS][MONTHS] = {
        {4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6},
        {8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3},
        {9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4},
        {7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2},
        {7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2}
        };

    int year, months;
    float subtotal = 0.0;
    float total = 0.0;

    printf("YEAR\t\tRAINFALL(inches)\n");

    for(year=0;year<5;year++)
    {
        for (months=0; months < MONTHS; months++)
        {
            subtotal+= rainfall_data[year][months];
        }
        printf("%d\t\t%.1f\n",2010+year,subtotal);
        total+=subtotal;
        subtotal=0;
    }
    printf("\nThe Yearly Average is %.1f (inches)\n",total/YEARS);

    printf("\n MONTHLY AVERAGES: \n\n");
    printf("JAN  FEB  MARCH  APRIL  MAY  JUNE  JULY  AUGUST  SEPTEMBER  OCTOBER  NOVEMBER  DECEMBER: \n");

    for(months=0; months < MONTHS; months++)
    {
        for (year = 0; year < YEARS; year++)
        {
            subtotal+=rainfall_data[year][months];
        }
        printf("%4.1f",subtotal/YEARS);
        subtotal=0;
    }

    return 0;
}

```



```
YEAR            RAINFALL(inches)
2010            32.4
2011            37.9
2012            49.8
2013            44.0
2014            32.9

The Yearly Average is 39.4 (inches)

 MONTHLY AVERAGES:

JAN  FEB  MARCH  APRIL  MAY  JUNE  JULY  AUGUST  SEPTEMBER  OCTOBER  NOVEMBER  DECEMBER:
 7.3 7.3 4.9 3.0 2.3 0.6 1.2 0.3 0.5 1.7 3.6 6.7
Process returned 0 (0x0)   execution time : 0.072 s
Press any key to continue.
```

University of Glasgow

Preprocessor and Macros

电子科技大学 University of Electronic Science and Technology of China

Preprocessor

- Preprocessing
  - Occurs before a program is compiled
  - Inclusion of other files
  - Definition of symbolic constants and macros
  - Conditional compilation of program code
  - Conditional execution of preprocessor directives
- Format of preprocessor directives
  - Lines begin with **#**
  - Only whitespace characters before directives on a line

The `#include` Preprocessor Directive

- **`#include`**
  - Copy of a specified file included in place of the directive
  - **`#include <filename>`**
    - Searches standard library for file
    - Use for standard library files, for example #include<stdio.h>
  - **`#include "filename"`**
    - Searches current directory, then standard library
    - Use for user-defined files, for example #include “my_file.h”

The `#define` Preprocessor Directive: Symbolic Constants

- **`#define`**
  - Preprocessor directive used to create symbolic constants and macros
  - Symbolic constants
    - When program compiled, all occurrences of symbolic constant replaced with replacement text
  - Format

    `#define` *identifier replacement-text*
  - Example:

```
#define PI 3.14159
```

  - Everything to right of identifier replaces text

```
#define PI = 3.14159
```

    - Replaces "**`PI`**" with "**`= 3.14159`**"
  - Cannot redefine symbolic constants once they have been created

The `#define` Preprocessor Directive: Macros

- Macro
  - Operation defined in **`#define`**
  - A macro without arguments is treated like a symbolic constant
  - A macro with arguments has its arguments substituted for replacement text, when the macro is expanded
  - Performs a text substitution – no data type checking
  - The macro

```
    #define CIRCLE_AREA( x ) ( PI * ( x ) * ( x ) )
```

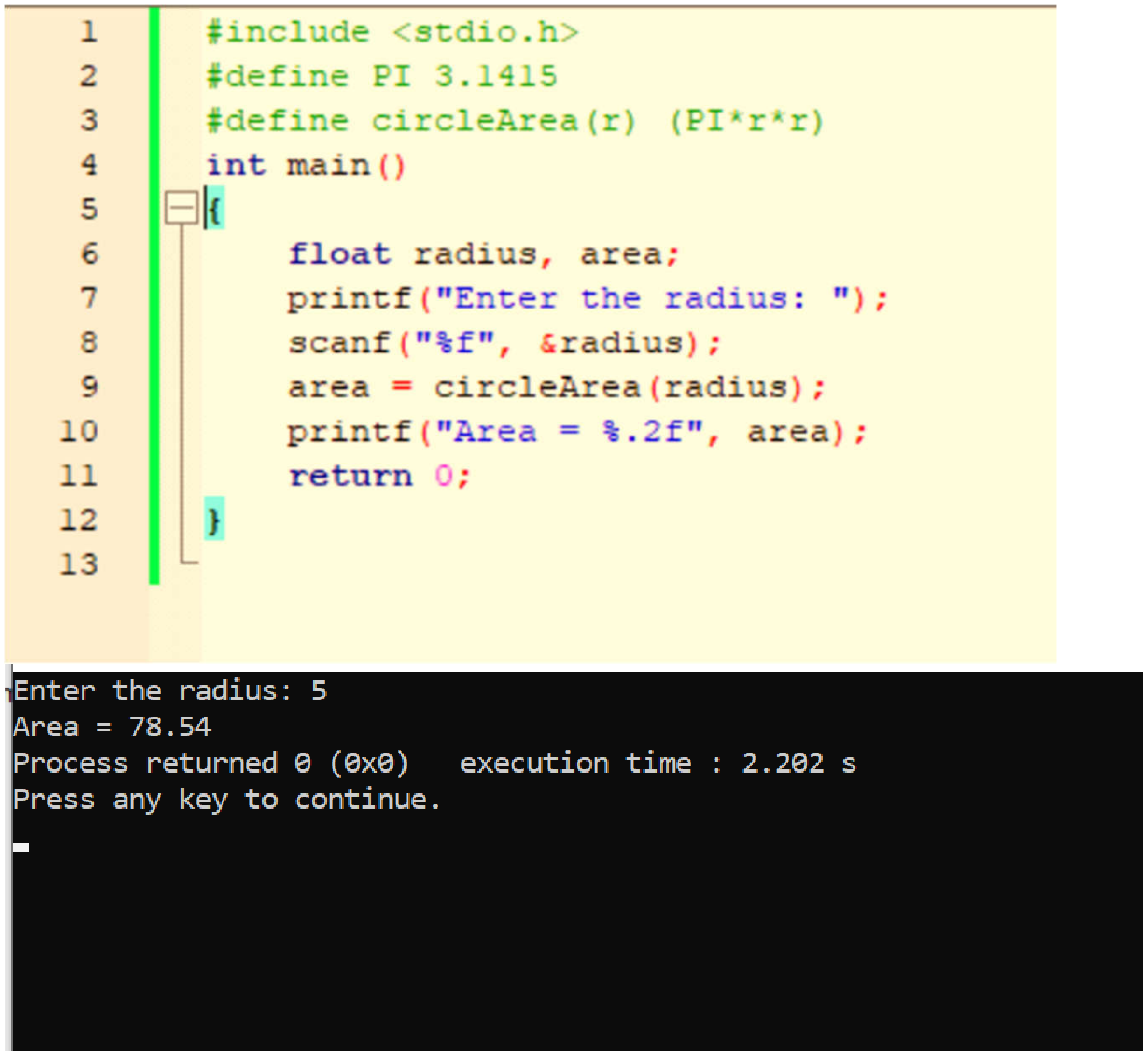
would cause

```
    area = CIRCLE_AREA( 4 );
```

to become

```
    area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

Ex 3: Compute Area of a Circle using Macros



```c
#include <stdio.h>
#define PI 3.1415
#define circleArea(r) (PI*r*r)
int main()
{
    float radius, area;
    printf("Enter the radius: ");
    scanf("%f", &radius);
    area = circleArea(radius);
    printf("Area = %.2f", area);
    return 0;
}

```

```
Enter the radius: 5
Area = 78.54
Process returned 0 (0x0)   execution time : 2.202 s
Press any key to continue.
```

Conditional Compilation

- To instruct preprocessor whether to include a block of code or not. To do so, conditional directives can be used
- if statement is checked during the execution time to check for conditions **but conditionals are used to include or exclude a block of code before the execution of program**

- To use conditionals #ifdef #if #defined #else and #elseif conditionals are used

```
#ifdef MACRO
  // conditional codes
#endif
```

The special operator #defined is used to test whether a certain macro is defined or not. It's often used with #if directive.

```
#if defined BUFFER_SIZE && BUFFER_SIZE >= 2048

// codes
```

Strings

Strings

- A string is basically a block of text like some ones name
- Since people use computer programs you need to be able to store data that is meaningful to people. So quite often you will come across the need to store and/or manipulate a sequence of characters.
- Earlier in the course we came across the variable type *char* that is used to store a single character.
- A string is really a collection of characters
- Therefore it should be no surprise to find that a string is an array of characters:

```
char stringName[length];
```

- The following line of C sets up a string capable of holding a 20 character name

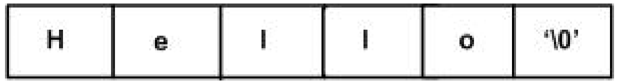
```
char yourName[20];
```

Strings

- This generates a new problem, we have 20 slots capable of storing character, what if we only use 4 of them?
- Imagine we want to store the name ”Adam” as a string, how does the computer know where the end of the name is?
- There is a special convention that stipulates when the end of a string has been reached. This is that a value of `\0` is used as the terminator character, it is also referred to as a null character
- The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word `“Hello”`

```
char greeting[6] = ”Hello”;
```

- the memory presentation of above defined string



Strings

- Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the `'\0'` at the end of the string when it initializes the array.
- You must allow space for it when declaring the string variable
- If you want to store a 10 character name you need space for 11 when you include the null. Do not assume that if you are going to fill all the space made available that you don't need the null, you always need the null at the end of the string!
- To clarify this further, do not confuse the character 'o' with the value 0

Characters and Strings

Difference:
' '
" "

- Characters
  - Building blocks of programs
    - Every program is a sequence of meaningfully grouped characters
  - Character constant
    - An **`int`** value represented as a character in single quotes
    - **`'z'`** represents the integer value of **`z`**
- Strings
  - Series of characters treated as a single unit
    - Can include letters, digits and special characters (**`*`**, **`/`**, **`$`**)
  - String literal (string constant) - written in double quotes
    - **`"Hello"`**
  - Strings are arrays of characters
    - String a pointer to first character
    - Value of string is the address of first character

Strings Declarations

- String declarations
  - Declare as a character array or a variable of type **`char`**

    ```
    char color[] = "blue";
    ```

  - Remember that strings represented as character arrays end with **`'\0'`**
    - **`color`** has **`5`** elements
- Inputting strings
  - Use **`scanf`**

    ```
    scanf("%s", word);
    ```

    - Copies input into **`word[]`**
    - Do not need **`&`** (because a string is a pointer)
    - *word* :: means the address of the first element of the array
  - Remember to leave room in the array for **`'\0'`**

Example

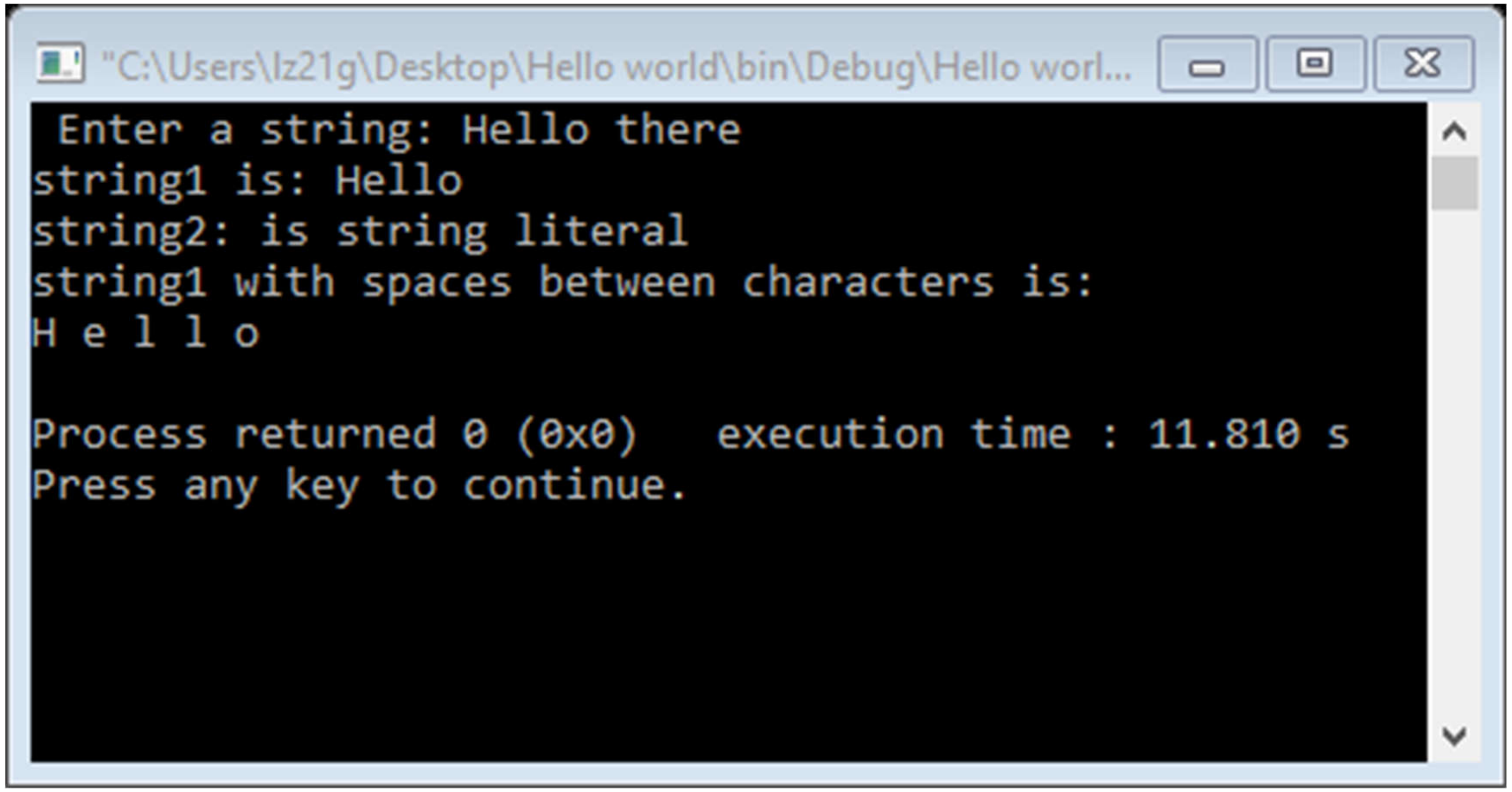
- Let’s look at the ‘Get name’ example program:

```
int main()
{
   char string1[ 20 ], string2[] = "string literal";
   int i;

   printf(" Enter a string: ");
   scanf( "%s", string1 );
   printf( "string1 is: %s\nstring2: is %s\n"
           "string1 with spaces between characters is:\n",
           string1, string2 );

   for ( i = 0; string1[ i ] != '\0'; i++ )
      printf( "%c ", string1[ i ] );

   printf( "\n" );
   return 0;
}
```

- Here we can see that the `scanf` function stores the users input into a string variable called string1
- The format specifier `%s` means that you can read a string from the user
- Notice that the `&` symbol is not needed in this case as it was when we used `scanf` earlier
- The reasons for this will be come clear when pointers are covered

Example

- When the above code is compiled and executed, it produces the following result:



```
 Enter a string: Hello there
string1 is: Hello
string2: is string literal
string1 with spaces between characters is:
H e l l o

Process returned 0 (0x0)   execution time : 11.810 s
Press any key to continue.
```

More on Strings

- How do we take in a name that includes a space?
- A function in stdio.h called *gets* (short for get string) will allow you to enter characters until you hit return (new line). Therefore you can read data containing spaces as follows:

```
char buffer[120];
gets(buffer);
```

- Note that no check is made by the function *gets* to see if there is enough space to store all that you are entering, it is assumed rightly or wrongly.
- It is a good idea to over allocate space to try and ensure that this is not a problem

More on Strings

- There is another function called `getchar` (short for get character)

```
        char ch;
        ch = getchar();
```

- The brackets after the `getchar` tell the compiler that it is a function and therefore are important

More on Strings

- C supports a wide range of functions that manipulate null-terminated strings and are part of the `'string.h'` library:

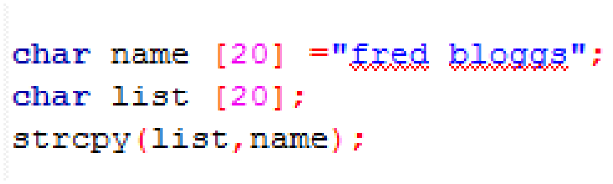
S.N.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

More on Strings (Live Demo)

- **strcpy**

```
strcpy(destination, source);
```

- String copy – this has two parameters which are basically the source and destination, and the source is copied to the destination.

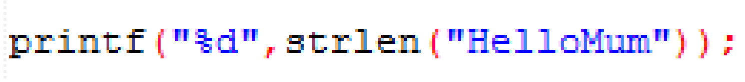


```
char name [20] ="fred bloggs";
char list [20];
strcpy(list,name);
```

- **strlen**

```
strlen(string);
```

- String length – this has one parameter, the string it is to count the length of. It returns the length of the string.



```
printf("%d",strlen("HelloMum"));
```

- This would print out 8 since the terminator character is not counted.

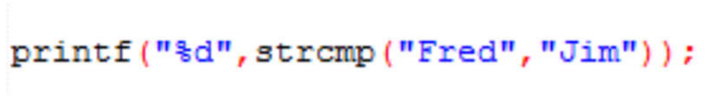
More on Strings

- **strcmp**

```
strcmp (s1, s2);
```

- String compare – this has two parameters which are the two strings to compare. This function returns a value as follows:
  - 1 if the first string is greater than the second
  - 0 if the two strings are the same
  - -1 if the first string is less than the second

- When deciding which string is greater or less than the character codes of the strings are used. Therefore the normal rules of alphabetic ordering apply: i.e. a<b, A<a, 1<A.



```
printf("%d",strcmp("Fred","Jim"));
```

- This would print -1 because Fred is less than Jim alphabetically.