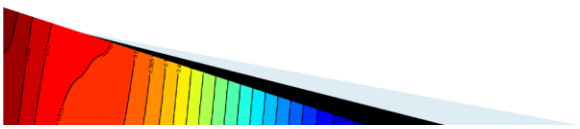
GNG 1106

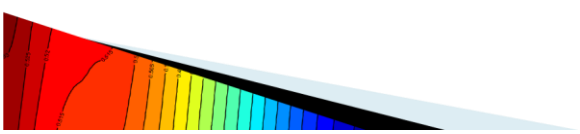
Fundamentals of Engineering Computation

Module 3 - Structures and Arrays

Fall 2016

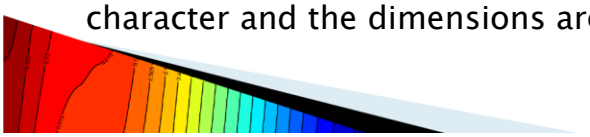


Topic 1: Structures



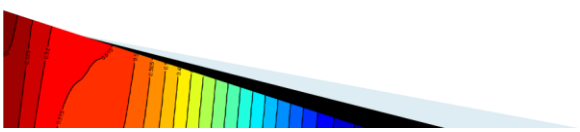
Data Structures

- A structure is used to group variables under one name.
- A structure may contain variables of different types.
- The structure can be viewed as a user-defined type or a complex type.
- A structure is useful when we wish to group data items that are related.
  - For example, we may want to group the dimensions of an object (say a cube), that is, the color, height, width and length; a structure can contain all of these data items.
  - In this example, the data items to be stored are of different types: the color can be represented by an character and the dimensions are real numbers.



The Structure Definition and Structure Variable

- A structure **type** must first be defined BEFORE using it to declare a structure variable
  - Structure variables are like variables, just with content that is more complex.
- The **Members** of the structure variables can be used like any other variable of the same type.
- Members of structures can be
  - Variables of basic types (`char`, `int`, `float`, etc.)
  - Other structure variables; this means that the structures can be composed of other structures.

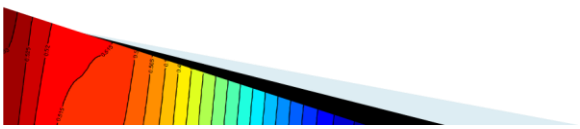


Definition of Structure Type in C

- A structure type must first be defined before a structure variable can be declared.
- The syntax for defining a structure type is:

```
struct structTypeName
{
    type name1; /* similar to declarations */
    type name2; /* but does NOT reserve memory */
    type name3; /* initialization is NOT valid */
  .
  .
  .
};
```

Careful: there is a ; here.



Definition of Structure Types in C – cont'd

- The keyword `struct` introduces the structure definition and gives it a name of a type (`structTypeName`).
- The variables declared within the `{}` are the structure's members; any number of members can be defined; members can be of any type, other structures.
- Members of the same structure definition must have unique names.
- Defining a structure is defining a new type.
  - We can declare variables of type `struct structTypeName`.
  - Defining a structure does not reserve any memory.

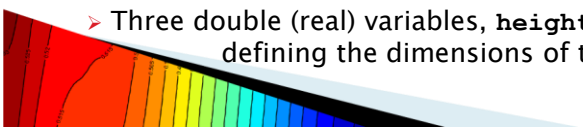
- Consider for example the structure `cube` which contains four members:

```
struct cube
{
    char color;
    double height;
    double width;
   double length;
};
```

  - This structure definition contains :
    - A char variable, **`color`**, for defining the color of the cube,
      - Note that symbolic constants can be used to give names to the character color values:

```
#define RED        'r'
#define GREEN      'g'
```

        Etc.
    - Three double (real) variables, **`height, width, length`**, for defining the dimensions of the cube.



Defining new types – `typedef`

- Instead of using "`struct cube`" all the time to declare a structure, a name can be associate to "`struct cube`" using `typedef`; that is to define a new type as follows:

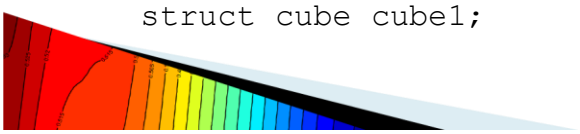
```
typedef struct cube CUBE;
```

  - **`CUBE`** is called a user type;
  - Conventional use is that names for structure user types are all in upper case, hence **`CUBE`**
  - The type **`CUBE`** now represents the type "`struct cube`" and can be used to declare structure variables:

```
CUBE cube1;
```

    - is the same declaration as

```
struct cube cube1;
```

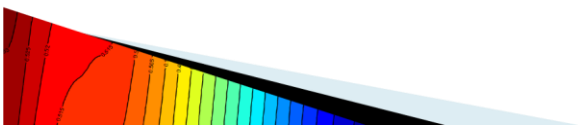


Defining new types – typedef

- It is also possible to combine the definition of a structure with the definition of a user type.

```
typedef struct structTypeName
{
    type name1;
    type name2;
    type name3;.
  .
  .
} NEW_TYPE;
```

This name is optional. If **omitted**, then variables CANNOT be declared with type
struct structTypeName
only with type
NEW_TYPE.



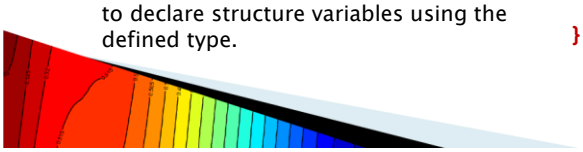
- The following shows three ways for defining a structure type CUBE.

```
struct cubestr
{
   char color;
   double height;
   double width;
   double length;
};
typedef struct cubestr CUBE;
```

```
typedef struct cubestr
{
   char color;
   double height;
   double width;
   double length;
} CUBE;
```

- The method to the right in red is the method adopted in this course.
  - It is the simplest since only the type name (i.e. CUBE) is required.
  - The new type name (CUBE) can be used to declare structure variables using the defined type.

```
typedef struct
{
   char color;
   double height;
   double width;
   double length;
} CUBE;
```

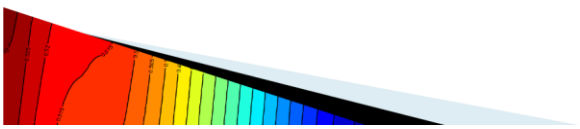


Declaring Structure Variables – Example

- In this course, we shall use a user type to declare structure variables.
- Thus to declare a structure variable defined in the previous slide we shall use:

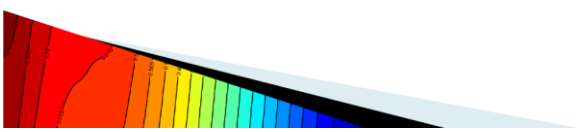
```
CUBE cube1;
```

  - The above declarations creates:
    - the structure variable `cube1` which is of type `CUBE`.
    - This variable contains 4 members: color, height, width, and length.

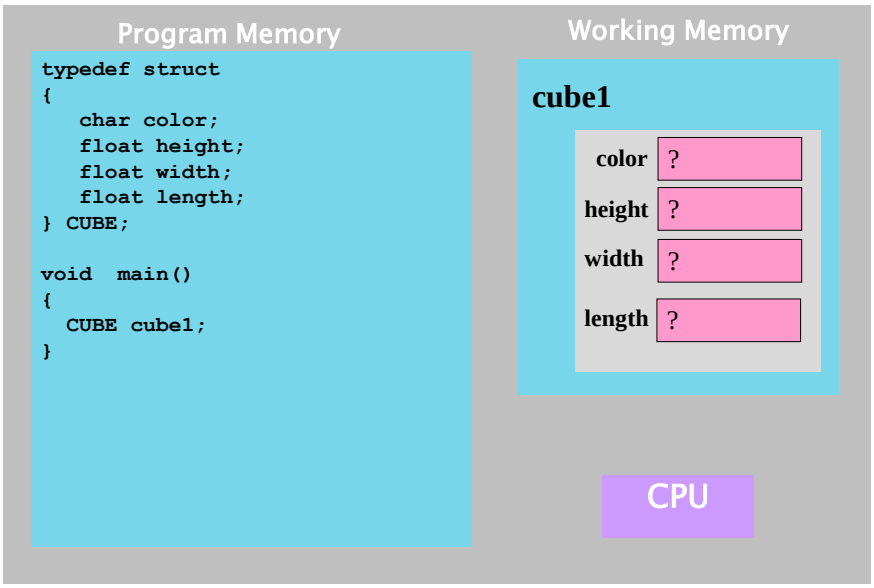


Declaring Structure Variables

- A defined structure type can be used in the same fashion as our basic data types (`int`, `char`, `float`, etc.)
  - To declare structure variables.
    - Such declarations provide "variables" only with data types that are much more complex.
  - Thus variables of structure types are just like variables:
    - Memory is allocated only once a variable is declared
    - Enough memory is allocated to store values for all of the structure's members (see the next slide).



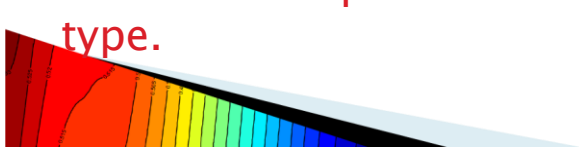
The Structure Variable in Memory



13

Operations with Structure Variables

- The only operations that can be performed on a structure variable are:
  - Assignment of a structure variable to another structure variable of the same type using = (content is copied).
  - Accessing a structure's members.
- Structure variables **cannot** be compared.
- The members of the structure variable can be used in expressions according to their type.



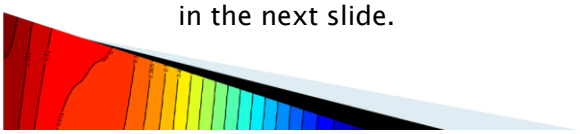
14

Initializing a Structure Variable

- A structure variable can be initialized when it is declared by assigning an initializer list enclosed in `{}` to the variable.
  - Individual initializers are separated by commas, and be of type compatible with the members.
  - Consider the following declaration:

```
CUBE oneCube = {'r', 2.4, 5.6, 3.2};
```

    The above declaration results in the creation of the structure variable `oneCube` of type `CUBE` initialized with:
    - the character 'r' is assigned to the structure member `color`,
    - the value `2.4` is assigned to the structure member `width`,
    - the value `5.6` is assigned to the structure member `height`,
    - the value `3.2` is assigned to the structure member `length`,
- Structure variables can also be initialized after the declaration by accessing individual members within assignments as shown in the in the next slide.



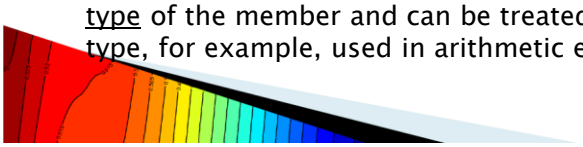
Using the Members of a Structure Variable

- The members of a structure can be accessed using the structure member operator (the period .), also called the dot operator, the structure variable name and the structure member names.
- A member of a structure variable is accessed by placing the dot operator between the name of the structure variable and the name of the member to be accessed.
  - Suppose that the structure variable `oneCube` has been declared previously.

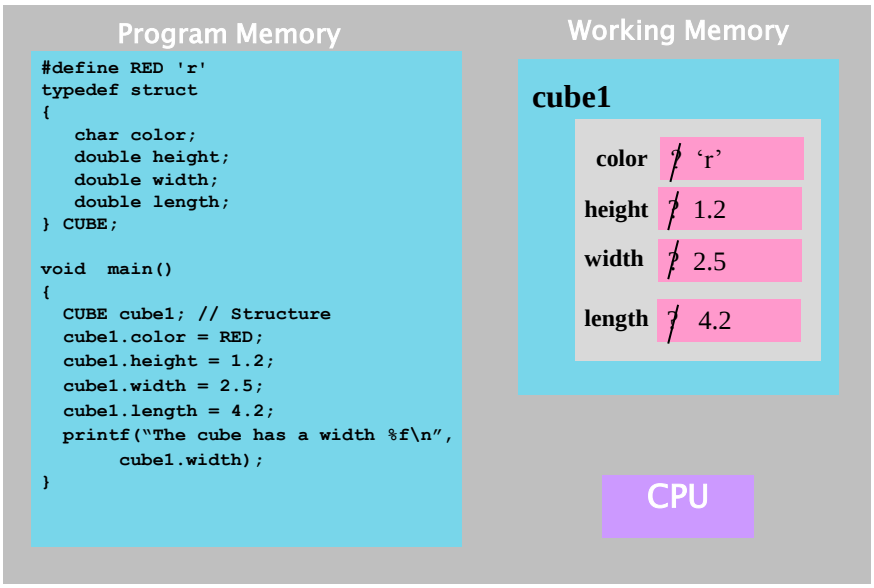
```
cube1.color = RED;
cube1.height = 1.2;
cube1.width = 2.5;
cube1.length = 4.2;
volume = cube1.height*cube1.width*cube1.length;
printf("The cube has a width of %f and volume %f\n",
       cube1.width, volume);
```

Initializes the structure variable `oneCube1`

  - A member of a structure variable is manipulated according to the type of the member and can be treated like a variable of the same type, for example, used in arithmetic expressions.



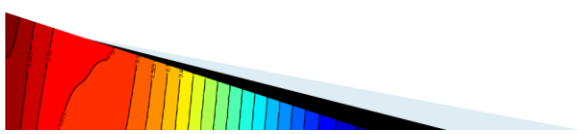
Using the Members of a Structure Variable



17

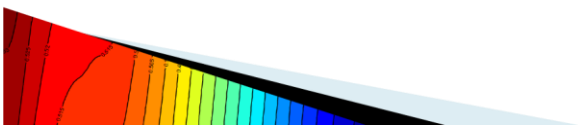
Example cubeType.c

- Shows the following
  - Defining a structure using a user type.
  - Declaring a structure variable.
  - Initializing a structure variable when declaring it.
  - Using structure members
    - Filling with user data
    - Using in an arithmetic expression
  - Assignment operation with structure variables.



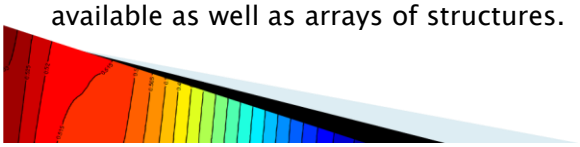
18

Topic 2: Arrays



Arrays

- We often encounter problems in engineering where a large amount of the same type of data must be manipulated (like a hundred temperature measurements, for example).
  - In such cases, it is inconvenient if not impossible to define a distinct variable name for each datum; it is best to use an array.
- An array is a data structure that can conveniently store a large amount of data of the same type.
- An array is a static data structure, which means that its size does not vary during the execution of the program.
- We can define in a computer an array of any of the basic data types available as well as arrays of structures.

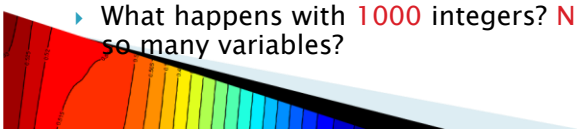


A Problem with Simple Variables…

- Suppose that a program reads 5 integers and displays them in reverse order:

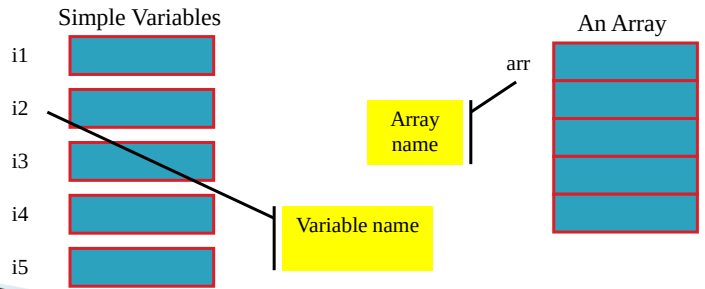
```
scanf("%d", &i1);
scanf("%d", &i2);
scanf("%d", &i3);
scanf("%d", &i4);
scanf("%d", &i5);

printf("%d\n",i5);
printf("%d\n",i4);
printf("%d\n",i3);
printf("%d\n",i2);
printf("%d\n",i1);
```

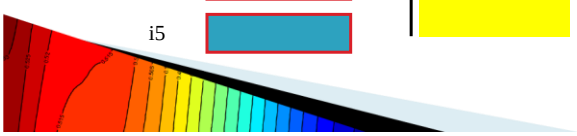
- What happens with 1000 integers? N integers? How to declare so many variables?



Arrays

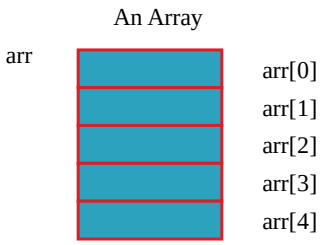
- "Simple" variable contains one value.
- An array has many "variables" inside it, each able to contain a different value.
- An array is a collection of variables of the same type.





Arrays (continued)

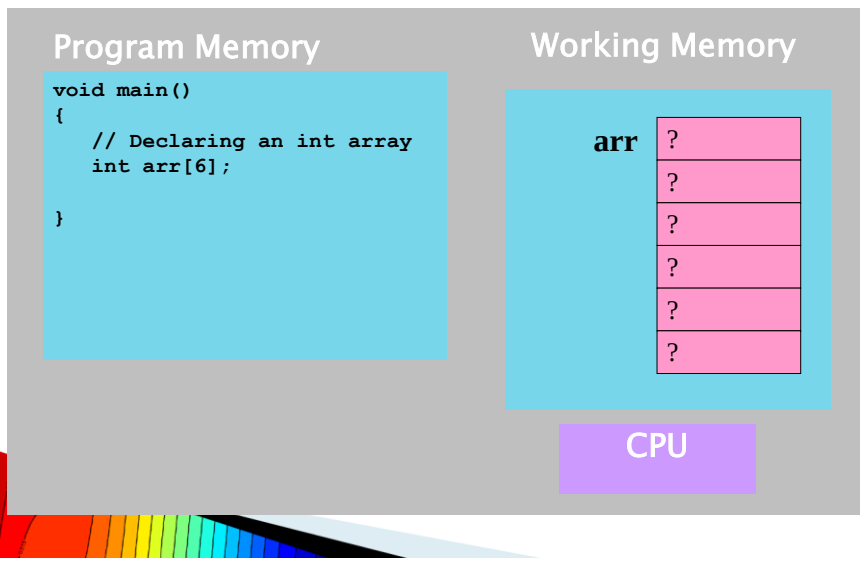
- If array arr has 5 positions, we refer to them using the integers 0–4, called indices or subscripts.
  - e.g. arr[2] is the THIRD position with index 2.
  - Note that arr[2] is equivalent to a variable name and can be used anywhere a variable name is used, e.g. in expressions.



23

A one dimensional array

- An array is a collection of memory locations where multiple values of the same type can be stored.



24

The one dimensional array in C

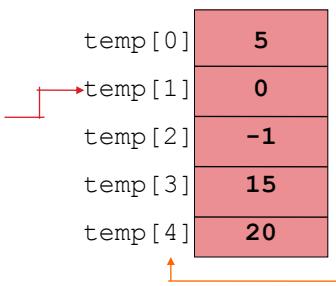
- A one dimensional array consists of an amount of contiguous memory elements that can contain data of the same type.
  - The number of elements are fixed.
- The name of an array must be created in the same way as a variable name and using the same set of characters available and convention (using lower case characters) for a variable name.
  - The name of the array corresponds to the address of the first data element in the array
- An element of an array is accessible by citing the name of the array followed by the element number or index in brackets `[]`.
  - The name of the array with an index is equivalent to a variable name and can be used anywhere a variable name is used (e.g. expressions)
  - Note that expressions can be used as an index
  - Index values start at 0

Important

One Dimensional Arrays in C (continued)

- In C, the array consists of a name and indices are used to access its elements.
- In memory, a 1-D array of dimension 5, called `temp` and containing `int`'s looks like:

An individual element of this array is accessed as `temp[1]`, for example.



The index, which is an integer in `[]` identifies the position in the array, or the element number.

- **In C, the first element in an array is always element 0.**
  - An array of 5 elements is indexed from 0 to 4,
  - an array of N elements is accessed using indices 0 though N-1.
  - Careful: the i^{th} element is at position i-1 in the array; ie: the 4^{th} element is in position 3.
- The position of the element or the index must be an integer or an expression that evaluates to an integer.
- An element of an array is used in the same manner as a variable.
  - It can be used to the left of an assignment operator to store a value in the element.
  - It can be used in an arithmetic or logical expression.

E.g.:

```
temp[1] = 5 + 3;
temp[1] = temp[2-1] + 1;
temp[1] = temp[1] + 1;
temp[0] = temp[1] / 2;
temp[0] = (temp[0] <= temp[1]);
```

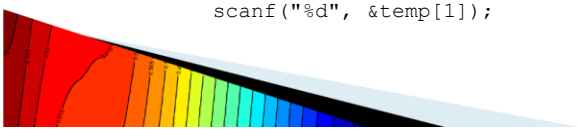
Assume that `temp` is an array of `int`'s

temp[1]	8
temp[1]	9
temp[1]	10
temp[0]	5
temp[0]	1

- The elements of an array can also be used in the arguments or function calls:

E.g.:

```
printf("%d\n", (temp[0]+temp[1]));
scanf("%d", &temp[1]);
```

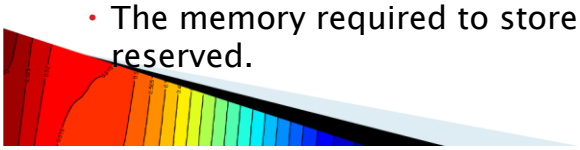


Declaration and Initialization of One Dimensional Arrays

- Like with simple variables an array created in memory with a declaration; the declaration specifies:
  - the name of the array,
  - the type of data stored in the array, and
  - the size of the array (number of elements).

E.g.:
```
int temp[5];
double x[5], y[20];
```

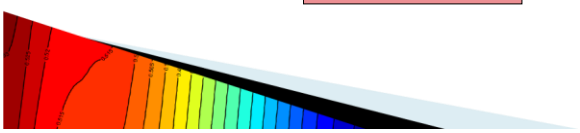
  - The first declaration defines the array `temp` which contains 5 data elements of type `int`.
  - The second declaration defines the arrays `x` and `y`, which contain 5 and 20 elements, respectively, of type `double`.
  - The memory required to store these arrays is reserved.



- An array can be initialized using assignment instructions (next week, we shall see how we can do this with a loop).

  E.g.:
```
double x[5];
x[0] = 0.0;
x[1] = x[0] + 0.5;
x[2] = x[1] + 0.5;
x[3] = x[2] + 0.5;
x[4] = x[3] + 0.5;
```

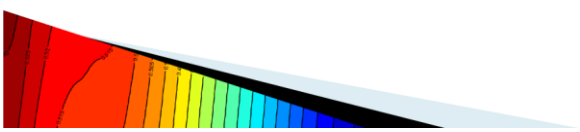
index	x[index]
0	0.0
1	0.5
2	1.0
3	1.5
4	2.0



- Like with structures, we can also initialize an array in its declaration.
  - The initial values are listed using , between {} and assigned to the array definition.
    - E.g.: `double x[5]={0.0, 0.5, 1.0, 1.5, 2.0};`
    - After compilation, the array `x` contains the initial values in contiguous positions:

x[0]	0.0
x[1]	0.5
x[2]	1.0
x[3]	1.5
x[4]	2.0

  - If there are more initial values than elements in the array, a syntax error is generated.



- If there are less initial values than elements, then the last elements are set to zero.

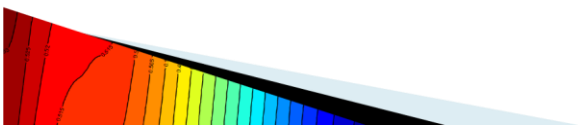
  E.g.: `int temp[5]={0};`

  `double x[5]={0.0, 0.5, 1.0};`

  - Interpreting the declarations gives the arrays `temp` and `x` with the following values:

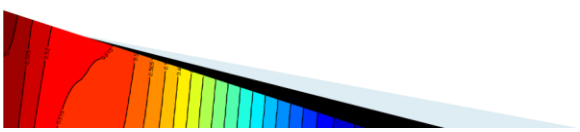
temp[0]	0
temp[1]	0
temp[2]	0
temp[3]	0
temp[4]	0

x[0]	0.0
x[1]	0.5
x[2]	1.0
x[3]	0.0
x[4]	0.0



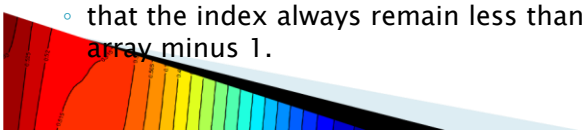
- The use of descriptive symbolic constants in the definition of arrays and in their manipulation is strongly encouraged since they:
  - facilitate modifications to the program if the array size must change, and they
  - eliminate "magic" numbers from the program, thus rendering it more readable.

    E.g.:
    ```
    #define NBR_TEMPS 5
          ⋮
    int temp[NBR_TEMPS], ctr;
    ```
  - In the above example, if the number of temperatures to be stored must be increased from 5 to 10 then only one line of the program would need to be modified:
    - `#define NBR_TEMPS 10`
  - In a large program containing many arrays and code to manipulate them, the use of symbolic constants greatly enhances the maintainability of the code.



Running Over Array Limits

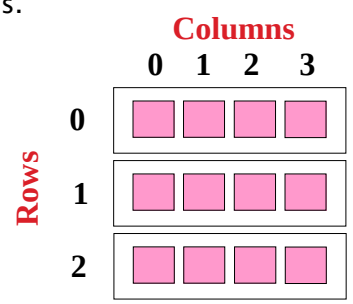
- There is no mechanism in C to prevent a program from running over the upper and lower limits of an array during execution!
  - Reading or accessing an element outside of the array limits will:
    - generally introduce errors into the results obtained from the program, and
    - may not cause the computer to "crash" or to become "hung-up".
  - Assigning a value to an element outside of the array limits will:
    - generally introduce errors into the results obtained from the program, and
    - often will cause the computer to "crash" or to become "hung-up".
- The most common cause for running over the array limits is an error in the looping structure(s) used to traverse the array (see this in the next week). Always verify that:
  - the index never becomes negative in value, and
  - that the index always remain less than or equal to the size of the array minus 1.



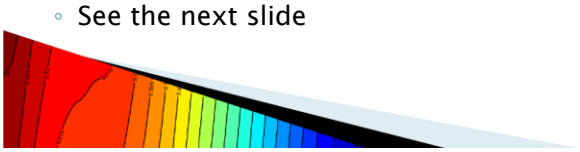
33

The 2 dimensional array – the matrix

- The 2 dimensional array, a matrix, is essentially an array of arrays
  - Each row in the matrix is a 1-dimensional array
  - The elements in the same positions of the row arrays make up the matrix columns.

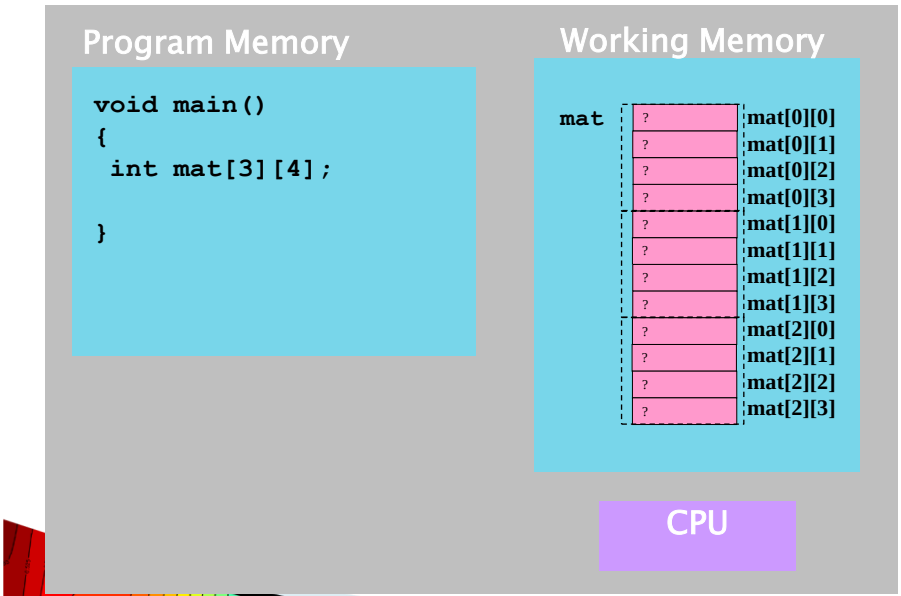


- The matrix, like the 1-D array, occupies a contiguous memory locations
  - See the next slide



34

The matrix in memory



The matrix – an array of arrays

- The two dimensional array is often used to store data arranged in a row-column form, i.e. as matrices
  - Note the organization in memory – it occupies contiguous space just like the array
- A name (like the case of the array) is used to reference the matrix
  - Like the case of the array, the name corresponds to the address of the first element in the matrix
- The elements in a two dimensional array are accessed by citing the name of the array followed by two indices, each inserted between `[]` and `[]`, e.g. `mat[2][5]`.
  - The first index specifies the row and the second index specifies the column.
- The name of a matrix with a single index (e.g. `mat[1]`) is treated like the name of a 1D array, i.e., each row is a 1D array.
  - Thus the matrix is an array of arrays.

- **In C 2D arrays are always indexed from 0 on.**
  - ➔ The first row is row 0 and the first column is column 0.
- In reference to the array `a` of `int`'s on the previous slide:
  - element `a[0][0]` contains `2`,
  - element `a[2][3]` contains `-13`,
  - element `a[1][2]` contains `-5`.
- An element of a two dimensional array is used in the same manner as a variable.
  - It can be used to the left of an assignment operator to store a value in the element.
  - It can be used in an arithmetic or logical expression.

Assume that `a` is an 2D array of `int`'s

E.g.:

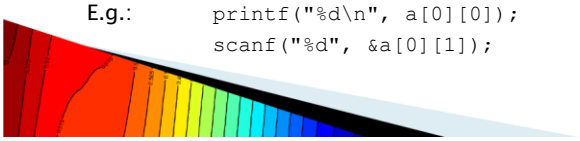
```
a[0][0] = 5 + 3;
a[2][3] = a[0][1-1] - 2;
a[2][3] = a[2][3]+1;
a[0][1] = a[2][3] / 2;
a[0][0] = (a[0][0] <= a[0][1]);
```

a[0][0]	8
a[2][3]	6
a[2][3]	7
a[0][1]	3
a[0][0]	0

- The elements of an array can also be used as an argument in a `printf` or a `scanf`:

E.g.:

```
printf("%d\n", a[0][0]);
scanf("%d", &a[0][1]);
```



37

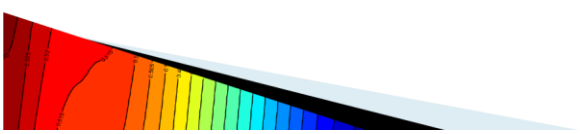
Declaration and Initialization of Two Dimensional Arrays

- A two dimensional array is defined in a declaration; the declaration specifies:
  - the name of the array,
  - the type of data stored in the array, and
  - the size of the array (number of rows and number of columns).

E.g.:

```
int a[3][4];
double y[5][5];
```

  - Thus, `a` is defined as an array of `int`'s having 3 rows and 4 columns and `y` is an array of `double`'s having 5 rows and 5 columns.
- Indexes can also be variables (type `int`) and even expressions.
- In Module 5, we shall see how to initialize a 2D array using nested loops.



38

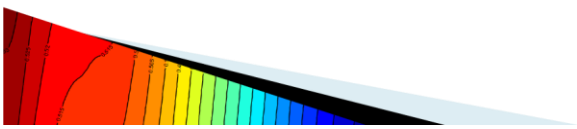
- We can also initialize an array in the declaration in which it is defined.
  - The initial values are listed row-wise, where each row is grouped using {}; the rows are then separated by commas, enclosed in {} and assigned to the array definition.

    E.g.: `int a[3][4]={{2,-1,11,8},{0,4,-5,-9},{3,10,-7,-13}};`

    row 0 row 1 row 2
  - If the {} delimiting the rows are omitted, then the compiler still initializes row-wise starting from the first row to the last.

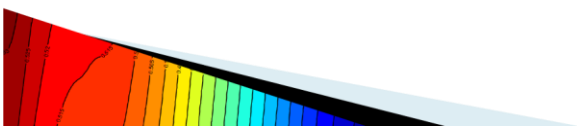
    E.g.: `int a[3][4]={2,-1,11,8,0,4,-5,-9,3,10,-7,-13};`
    - The above stores exactly the same initial values in the array as in the previous example but it is not as readable.
- If values are missing in the initialization list then the missing values will be assumed zero.

  E.g.:
  ```
  int a[3][4]={0};
  int a[3][4]={{2,-1,11},{0,4,-5},{3,10,-7}};
  int a[3][4]={2,-1,11,8,0,4,-5,-9};
  ```
  - In the first case the entire 2D array is set to zero.
  - In the second case, the last column is set to zero.
  - In the third case, the last row is set to zero.



- The use of descriptive symbolic constants in the definition of two dimensional arrays and in their manipulation is strongly encouraged for the same reasons as in the case of the one dimensional arrays.

  E.g.:
  ```
  #define NBR_STUDENTS 3
  #define NBR_TESTS 4
    ⋮
  int a[NBR_STUDENTS][NBR_TESTS];
  ```



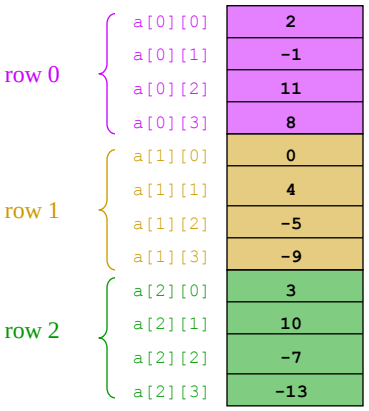
Recall the organization of Two Dimensional Arrays in Memory

- Two dimensional arrays are stored in contiguous memory elements. Storage is organized row-wise, beginning with the first element of the first row and ending with the last element of the last row.

E.g.: `int a[3][4]={{2,-1,11,8},{0,4,-5,-9},{3,10,-7,-13}};`

row 0 row 1 row 2

The array `a` is saved row-wise as 12 integers in contiguous memory locations:



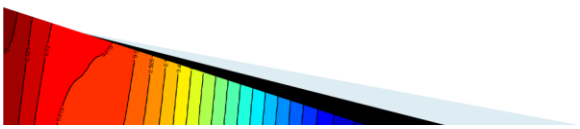
41

Topic 3: More Standard C Functions

42

Standard Functions in C

- There exists in C a standard library of functions which are available for general use.
  - Avoid re-inventing the wheel! Familiarize yourselves with the C standard functions and use them. The standard functions are defined by ANSI (American National Standards Institute) and are available with any C compiler that adheres to the ANSI C standard.
  - We find in the standard library: mathematical functions (studied in Module 2), input/output functions, file manipulation functions and functions to manipulate character strings, among others.
    - `printf` and `scanf` are ANSI C standard functions.



The `getchar` and `putchar` Functions

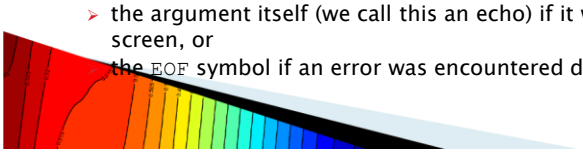
- The function `getchar` can be used to read in a single character from the input stream.
  - Recall that characters stored in the input stream have been typed on the keyboard.
  - The prototype of `getchar` is found in the header file `stdio.h` and reads:

    ```
    int getchar(void);
    ```

  - thus `getchar` returns an integer to the caller and expects no arguments.
  - The integer returned by `getchar` is the ASCII code that corresponds to the next character to be read form the input stream.
  - `getchar` reads characters from the input stream. The characters typed at a keyboard are added to the input stream buffer only after the carriage return key has been typed!
- The function `putchar` can be used to print out a single character on the screen.
  - The prototype of `putchar` is found in the header file `stdio.h` and reads:

    ```
    int putchar(int);
    ```

  - thus `putchar` expects an integer as an argument and returns an integer to the caller.
  - The integer sent to `putchar` as an argument is the ASCII code that corresponds to the character to be printed out on the screen.
  - The integer returned by `putchar` is either:
    - the argument itself (we call this an echo) if it was correctly printed on the screen, or
    - the `EOF` symbol if an error was encountered during the execution of `putchar`.

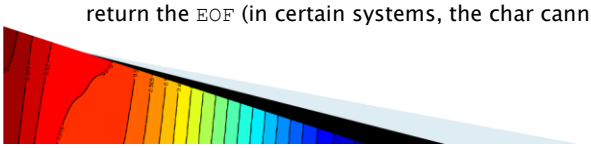


The EOF symbol

- `EOF` stands for End Of File; this symbol is not a standard character and thus does not have an ASCII code.
  - The numerical value associated with the `EOF` symbol obviously cannot be an integer within the range 0 through 127 since these numbers are used to represent standard characters according to the ASCII code.
  - `EOF` is a C symbolic constant and is defined in the header file `stdio.h`
  - The value associated with `EOF` can easily be determined by printing it out to the screen:

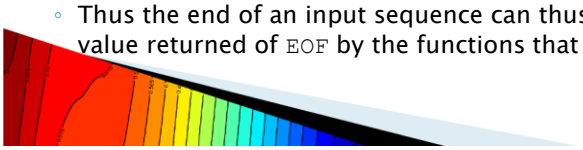
    ```
    printf("%d\n", EOF);
    ```

  - The integer used to represent the `EOF` symbol is system dependent; it is usually corresponds to the value -1 (which is the case on Windows systems).
- When "control z" is typed on a PC (Windows):
  - All characters that follow the Cntrl-Z all the way to the end of the line, are lost (including the newline character). The content before the Cntrl-Z is passed to the program.
  - If the Cntrl-Z is typed at the start of a line, the function such as getchar() returns and EOF (-1).
  - The Cntrl-Z is not translated to a value stored in the buffer of the input stream.
  - The functions `getchar` and `putchar` must therefore be of type `int` in order to return the `EOF` (in certain systems, the char cannot contain a negative value)



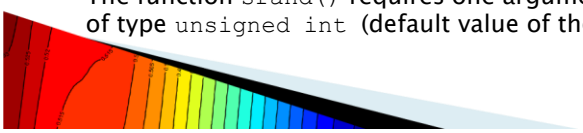
Processing Input Sequences

- A line of text can be entered through the keyboard by typing in the text followed by the carriage return key which enters the new line symbol into the input stream.
  - ➔ Lines of text entered into the input stream are thus delimited by the new line symbol.
  - ➔ The end of a line of text can be found by testing the character read against the new line symbol.
- An input sequence is comprised of one or more lines of text.
  - We need a key other than the carriage return key to indicate the end of an input sequence since the carriage return is already being used to indicate the end of a line of text.
  - The end of an input sequence is indicated by entering "`cntrl-Z`" (the keys `ctrl` and `z` at the same time) on the keyboard. This key sequence can also be represented by "`ctrl z`" or "`^z`".
  - This sequence of keys is interpreted as the end of an input sequence and is translated to EOF (-1), i.e., the value returned by functions that reads standard input.
  - Thus the end of an input sequence can thus be determined by testing the value returned of `EOF` by the functions that read the standard input.



Generation of Random Numbers

- Random numbers are often used to simulate a random process, such as a coin toss or a dice roll.
- The standard functions `rand()` and `srand()` can be used to generate a pseudo-random sequence of numbers. These sequences are called pseudo-random since the numbers eventually start repeating themselves.
- The function `rand()` is of type `int`, thus it returns an integer, and it does not require an argument.
  - The integer returned by `rand()` is contained between 0 and `RAND_MAX` where `RAND_MAX` is a symbolic constant.
  - The value associated with `RAND_MAX`, defined in `stdlib.h` (`#include <stdlib.h>`), is usually 32767.
- From one execution to another, `rand()` generates the same sequence of pseudo-random numbers. This sequence can be changed via the function `srand()` which changes the root or the seed of the random number generator.
  - The function `srand()` does not return a value.
  - The function `srand()` requires one argument which is a positive integer of type `unsigned int` (default value of the seed is 1).



Sequences of Random Integers

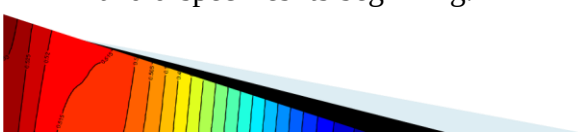
- We often wish to generate a sequence of random integers that are within a prescribed range. The modulus operator `%` is quite useful for this task.

E.g.:
```
int integer;
integer = rand() % 8;          → 0 ≤ integer ≤ 7
integer = 1 + rand() % 6;      → 1 ≤ integer ≤ 6
integer = rand() % 51 - 25;    → -25 ≤ integer ≤ 25
```

- The following general formula can be used to generate random integers within the range a ≤ integer ≤ b:

```
int a, b, integer;
integer = a + rand() % (b - a + 1);
```

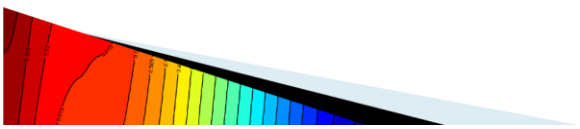
  – The expression `(b - a + 1)` specifies the width of the range and `a` specifies its beginning.



Sequences of Random Real Numbers

- We often wish to generate a sequence of random real numbers that are within a prescribed range.
  - The following bit of code will produce a random real number within the range: $x \le real\_no \le y$.

```
double x, y, real_no;
    ⋮
/* Random real number in the range 0.0 to 1.0. */
real_no = (double) rand() / RAND_MAX;
/* Scale to reduce the range to 0.0 to y - x. */
real_no = real_no * (y-x);
/* Add x to displace the range to x to y. */
real_no = real_no + x;
```



Next Module

- Topic 1: More on arrays
- Topic 2: Character/String Library
- Topic 3: Pointers