C PROGRAMMING
Lecture 4

1st semester 2023-2024

Functions

- Functions
  - Basic elements in C for modular programming
  - Collection of statements using a unique name
  - Programs combine user-defined functions with library functions
- Function calls
  - When invoking function
    - Provide function name and arguments
    - Function performs operations or manipulations
    - Function returns results

Functions

- General structure:
  - ```
    type func_name(formal_param_list){
      local variables definitions
      statements
    }
    ```
- Function exit:
  - When invoking return
  - When statements inside function end
    ```

Function call example

- Math library functions
  - perform common mathematical calculations
  - **`#include <math.h>`**
- Format for calling functions
  - **`FunctionName(`** *argument* **`);`**
    - If multiple arguments, use comma-separated list
  - **`printf( "%.2f", sqrt( 900.0 ) );`**
    - Calls function **`sqrt`**, which returns the square root of its argument
    - All math functions return data type **`double`**
  - Arguments may be constants, variables, or expressions

Functions

- Return only one value (a scalar).
- Can receive formal parameters.
- Should have one entrance and one exit.
- Parameters are **always passed by value**.
- Passing a pointer looks like passing by reference.
- Pointer is passed by values.

Functions - main

- The main function is a function like any other.
- It defines the entry point to your program.

Function definitions

- Functions are defined in three parts.
  - Return data type.
  - Name of the function (subject to variable naming rules)
  - Parameters to be passed into the function.

`int main(void)` // (void) can be replaced by ()

`int` - indicates that the function return an `int` value.

`main` – name of the function.

`void)` or `()` – indicates that no parameters are passed into the function.

Function prototypes

- A function prototype is a forward reference to a function.
- They provide a way to define functions prior to their real definition.
- Used to validate functions

```
int sum( int, int);
```

- Note that formal parameters are not necessary only data types.

Function Definition

- To define a function start with the function header.
- Define the return data type.
- Define the function name.
- Define the formal parameters.

```
int sum( int nr1, int nr2)
{
  return nr1+nr2;
}
```

Function Definition

- No semicolon at the end of the function definition.
- Formal parameters `nr1` and `nr2` are declared.
- Returns the value of `nr1+nr2`.
- Example
  - in main function.

    ```
    int i = sum(1,9);
    ```
  - returns the sum of 1 and 9 (=10).
  - stores this value in the variable `i`.

Header Files

- Header files
  - Contain function prototypes for library functions
  - **`<stdlib.h>`** , **`<math.h>`** , etc
  - Load with **`#include <filename>`**

    `#include <math.h>`
- Custom header files
  - Create file with functions
  - Save as **`filename.h`**
  - Load in other files with **`#include "filename.h"`**
  - Reuse functions

Example - random

- **`rand`** function
  - Load **`<stdlib.h>`**
  - Returns "random" number between **`0`** and **`RAND_MAX`** (at least **`32767`**)

```
i = rand();
```

- Scaling
  - To get a random number between **`1`** and **`n`**

```
1 + ( rand() % n )
```

    - **`rand() % n`** returns a number between **`0`** and **`n - 1`**

Example - random

- **`srand`** function
  - **`<stdlib.h>`**
  - Takes an integer seed and jumps to that location in its "random" sequence

    `srand(` *seed* **`);`**
  - **`srand( time( NULL ) );//load <time.h>`**
    - **`time( NULL )`**
      - Returns the time at which the program was compiled in seconds
      - “Randomizes" the seed

Example - factorial

```
#include<stdio.h>
int main(){
        int factorial(int n), i;
        printf("Value of n, and n!\n\n");
        for(i=0;i<=5;i++)
             printf(" %2d   %4d\n",i, factorial(i));
        return 0;
}
int factorial(int x){
        int y;
        if(x<0) return 0;
        for(y=1;x>0;x--)
                y=y*x;
        return y;
}
```

Single-Dimensional Arrays

- Generic declaration:

```
typename variablename[size]
```

- `typename` is any type
- `variablename` is any legal variable name
- `size` is a number the compiler can figure out
- For example

```
int  a[10];
```

- Defines an array of ints with subscripts ranging from 0 to 9
- There are 10*sizeof(int) bytes of memory reserved for this array.
- You can use `a[0]=10; x=a[2]; a[3]=a[2];`
- You can use `scanf("%d",&a[3]);`

Array-Bounds Checking

- In general, array bounds subscripts can be checked during:
  - Compilation (some C compilers will check literals)
  - Runtime (bounds are never checked)
- When accessing off bounds of any array, it will calculate the address and then attempts to use it
- May get “a value” (usually garbage)
- May get a memory exception (segmentation fault, core dump error)
- Programmer has to take care not to under/over flow

Arrays as Function Parameters

- In C, the rule is: “parameters are passed by value”.
- The array addresses (meaning the values of the array names), are passed to the function `f_array()`.

```
void f_array(int arr[ ],
int size)
{
   int i;
   for(i=0;i<size;i++)
   {
      arr[i]++;
   }
}
```

```
main()
{
   int arr1[3]={1,2,3};
   int arr2[4]={1,2,3,4};
   int i;
   f_array(arr1,3);
   for(i=0;i<3;i++)
      printf("%d\n",arr1[i]);
   f_array(arr2,4);
   for(i=0;i<4;i++)
      printf("%d\n",arr2[i]);
   return 0;
}
```

Example – Array Sorting

```
#include <stdio.h>
void sort(int arr[ ],int size)
{
  int i,j,k;
  for(i=0;i<size;i++)
      for(j=i;j>0;j--)
   if(arr[j]<arr[j-1])
   {
      k=arr[j];
      arr[j]=arr[j-1];
     arr[j-1]=k;
   }
}
```

Example – Array Sorting

```
int main()
{
   int i;
   int arr[10] = {1,5,4,8,7,2,4,5,9,0};
   printf(“Initial array: ”);
   for(i=0;i<10;i++)
      printf("%d ",arr[i]);

   printf("\n Sorted array: ");
   sort(arr,10);
   for(i=0;i<10;i++)
      printf("%d ",arr[i]);

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

Returning Array from Function

- C programming language doesn’t allow you to return an entire array from a function.
- You can return a pointer to an array by specifying the array's name without an index. Pointers are described in the following slides

Multidimensional Arrays

- C programming language allows arrays with several dimensions. General form is:

```
type name[size1]...[sizeN];
```

- Maximum number of subscripts (i.e. dimensions) is 12 .

Two Dimensional Arrays

- Two dimensional arrays are declared the same way one dimensional array is. Given a matrix (rows and columns)

```
int matrix[20][10];
```

- The first subscript gives the row number, and the second subscript specifies column number.

Declaration and initialization

- We can also initialize a two-dimensional array in a declaration statement; E.g.,

```
int m[2][3]={{1,2,3},{4,5,6}};
```

- which specifies the elements in each row of the array (the interior braces around each row of values could be omitted). The matrix for this example is:

$$m = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Declaration and initialization

- Specification of the number of rows (first subscript) can be omitted. But other subscripts can't be omitted. So, the following represents a valid initialization:

```
int m[][3]={{1,2,3},{4,5,6}};
```

Initializing 2D Arrays

- The following initializes arr[4][3] :

```
int arr[4][3]={{1,2,3},{4,5,6},
{7,8,9},{10,11,12}};
```

- Also can be done by:

```
int arr[4]
[3]={1,2,3,4,5,6,7,8,9, 10,11,12};
```

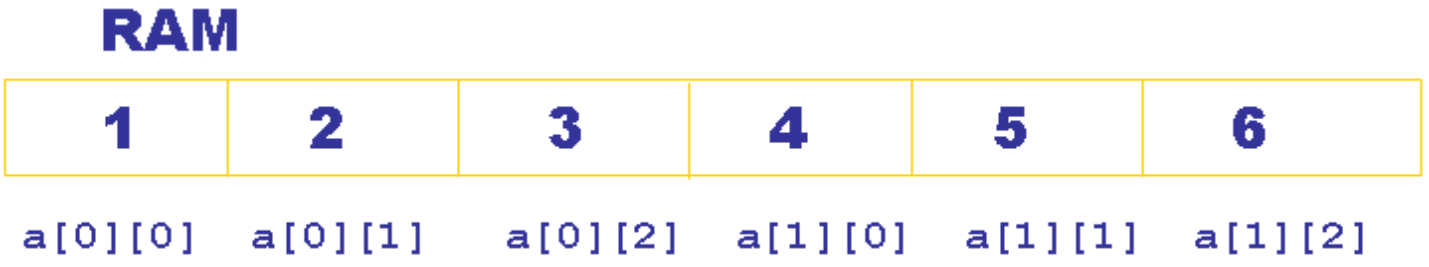
- is equivalent to

```
arr[0][0] = 1;
arr[0][1] = 2;
arr[0][2] = 3;
arr[1][0] = 4;
...
arr[3][2] = 12;
```

Memory Storage for a 2D Array

`int a[2][3]= {1,2,3,4,5,6};` specifies 6 integer locations.

- Storage for array elements are in contiguous locations in memory in row major order referenced by subscripts (index) values starting at 0 both for rows and columns.



Example

- Being given a matrix consisting of 10 rows of 10 integers per row (100 in total), write a program which reads 100 integers and stores the numbers in a two dimensional array named “matrix” and then computes the largest of all of these numbers and prints the largest value to the screen.

```
#include <stdio.h>
int main()
{
  int  row, col, maximum;
  int mat[10][10];

  for(row=0;row<10;++row)
    for(col=0;col<10;++col)
      scanf("%i",&mat[row][col]);

  maximum = mat[0][0];
  for(row=0;row<10;++row)
      for(col=0;col<10;++col)
   if (mat[row][col] > maximum)
      maximum = mat[row][col];

  printf(" max value in the array = %i\n",maximum);
  return 0;
}
```

Another Example

Write a program that computes the product of 2 matrices.
Take care to have valid values for dimensions, both with respect to a maximum size defines and to relation between column size value of the first and the row size value of the second matrix.

```
#include<stdio.h>
#define MAX 10

int main()
{
        int mat1[MAX][MAX],mat2[MAX][MAX], rest[MAX][MAX];
        int m1_nr1, m1_nr2, m2_nr1, m2_nr2, ind_col1,
ind_row1, ind_col2, ind_row2;
        int sum=0;
        do{
             printf("Matrix1 dims (between 2 and %d): ",MAX);
             scanf("%d %d",&m1_nr1, &m1_nr2);
        }while(m1_nr1>MAX||m1_nr2>MAX||m1_nr1<=1||m1_nr2<=1);
        do{
             printf("Matrix2 dims (between 2 and %d): ",MAX);
                scanf("%d %d",&m2_nr1, &m2_nr2);
        }while(m2_nr1>MAX||m2_nr2>MAX||m2_nr1<=1||m2_nr2<=1);
        if(m1_nr2!=m2_nr1)
        {
                printf("Multiplication not possible!\n");
                return -1;
        }
```

```
   printf("\n\nElements of the first matrix\n");
     for(ind_row1=0;ind_row1<m1_nr1;ind_row1++)
       for(ind_col1=0;ind_col1<m1_nr2;ind_col1++)
       {
            printf("Enter value for element mat1[%d]
[%d]: ",ind_row1, ind_col1);
           scanf("%d",&mat1[ind_row1][ind_col1]);
            printf("\n");
       }
       printf("\n\nEnter elements of the 2nd matrix\n");
       for(ind_row2=0;ind_row2<m2_nr1;ind_row2++)
         for(ind_col2=0;ind_col2<m2_nr2;ind_col2++)
         {
            printf("Enter value for element mat2[%d]
[%d]: ",ind_row2, ind_col2);
            scanf("%d",&mat2[ind_row2][ind_col2]);
            printf("\n");
         }
```

```
        for(ind_row1=0;ind_row1<m1_nr1;ind_row1++)
        {
           for(ind_col1=0;ind_col1<m2_nr2;ind_col1++)
           {
              for(ind_row2=0;ind_row2< m1_nr2;ind_row2++)
                  sum=sum+mat1[ind_row1][ind_row2]*
                 mat2[ind_row2][ind_col1];
              res[ind_row1][ind_col1] = sum;
              sum = 0;
           }
         }
         printf("\n\nElements of the product matrix:\n\n");
         for(ind_row1=0;ind_row1<m1_nr1;ind_row1++)
         {
           for(ind_col1=0;ind_col1<m2_nr2;ind_col1++)
               printf("%d ",res[ind_row1][ind_col1]);
           printf("\n");
         }
         return 0;
}
```