C PROGRAMMING
Lecture 3

1st semester 2022-2023

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();
    ```
  - Pseudorandom
    - Same sequence for every function call
- 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

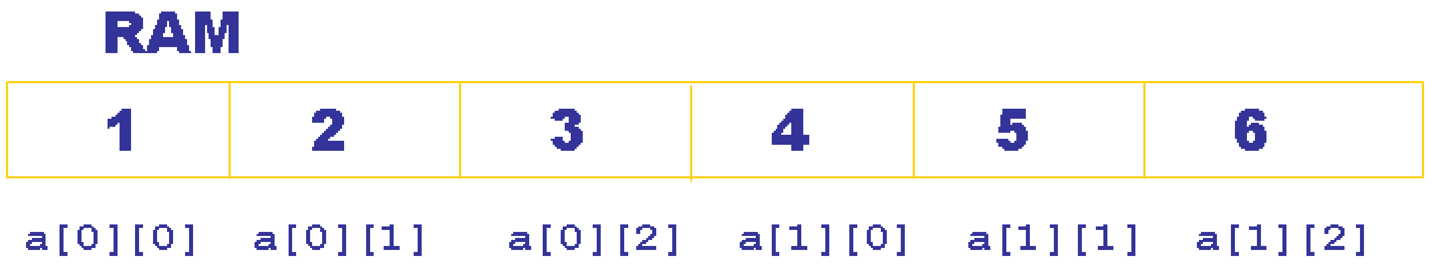
```
    arr[0][0] = 1;
    arr[0][1] = 2;
    arr[0][2] = 3;
    arr[1][0] = 4;
    ...
    arr[3][2] = 12;
```

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;
}
```

Pointers

- Review of variables
- Parameter passing
- Pointer declaration
- Pointer types
- Pointer arithmetics
- Examples

Remember variables

- Just to remember, what is a variable?
- A variable in a program is something with a name, the value of which can vary (it is also called object by R&K).
- It is assigned a specific block of memory to hold the value of that variable (address)
- The size of that block depends on the range over which the variable is allowed to vary (type)
- Imagine two "values" associated with the object. One is the value of the integer stored there and the other the "value" of the memory location, (address). Some refer these two values rvalue (right value) and lvalue (left value).
- Rvalues cannot be used on the left side of the assignment statement.
  - `10 = i;  //  is illegal`

Remember variables

```
 int nr1, nr2;
nr1 = 1;
nr2 = 2;   // observe this line
nr1 = nr2; // then observe this line
```

•the compiler interprets nr2 in first assignment as the address of the variable (its lvalue) and copies the value 2 to that address. Next line, the variable nr2 is interpreted as its rvalue (on the right hand side of the assignment operator '='). There nr2 refers to the value stored at the memory location of nr2, in this case 2. So, the value is copied to the address (the lvalue) of nr1.

Parameter passing

```
#include <stdio.h>
int main()
{
      void func(int nr);
      int val=1;
      printf("%d\n", val);    // first call for printf
      func(val);
      printf("%d\n", val);    // second call for printf
      func(val);
}
void func(int val)
{
      val++;
      printf("%d\n", val);    // third call for printf
}
```

Parameter passing

- The displayed result will be:

```
1
2
1
2
```

- Value of val is stored in a temporary location, as a copy

Parameter passing: arrays

```
#include <stdio.h>
int main()
{
      void func(int nr[], int max);
      int cnt, num[3];
      for(cnt=0;cnt<3;cnt++)
            num[cnt]=cnt;
      func(num,3);
      for(cnt=0;cnt<3;cnt++)
            printf("%d  ", num[cnt]);
}
void func(int num[], int max)
{
      int i;
      for(i=0;i<max;i++)
            num[i]=num[i]+1;
}
```

Parameter passing: arrays

- The result displayed is:

```
        1  2  3  4  5  6
```

- When array name is used, the address of the first element is passed
- The value of an array name is an address
- Copy of the address is passed to the function

Pointers - Introduction

- A pointer is a variable that can have as value a storage address
- Suppose we have an integer val. The unary operator & gives the address of its operand.
- The value after applying & operator (i.e. address) can be assigned to a pointer variable:

```
ptr = &val;
```

- By this we say “ptr points to val”
- the & operator retrieves the lvalue (address) of val, even though val is on the right hand side of the assignment operator, and copies that to the contents of ptr

Pointers - Declaration

- Pointers, being variables, must be declared
- In C, a pointer variable can point to values of one type only (except special case, see later)
- Declaration of a pointer variable is done by preceding its name with an asterisk. The declaration of pointer variable must specify the type:

```
    int *ptr;
```

Pointers - Declaration

- ptr is the name of the variable .
- * tells the compiler that ptr is a pointer variable, i.e. reserve however many bytes is required to store an address in memory.
- int says that the pointer variable stores the address of an integer.
- Similar to “regular”variables, ptr has no value after declaration. If the declaration is outside of any function, it is initialized to a value guaranteed in such a way that it doesn’t point to any C object or function. A pointer initialized in this manner is called a "null" pointer.

Pointers - Declaration

```
#include <stdio.h>
int nr1, nr2;
int *ptr;
int main()
{
    nr1 = 1;
    nr2 = 2;
    ptr = &nr1;
    printf("\n");
    printf("nr1 value %d and is at %p\n", nr1, &nr1);
    printf("nr2 value %d and is at %p\n", nr2, &nr2);
    printf("ptr value %p and is at %p\n", ptr, &ptr);
    printf("The value of the integer pointed to by ptr 
is %d\n", *ptr);
    return 0;
}
```

Pointers - type

•When we declare a variable, the compiler is given
- the name of the variable
- the type of the variable.

•For example, we declare a variable of type integer with the name val:

```
      int val;
```

•For the "int" part of this statement the compiler reserves (depending on the platform) 4 bytes of memory to hold the value of the integer.

•It also sets up a symbol table. In that table it adds the symbol val and the relative address in memory where those 4 bytes were reserved.

Pointers - Operators

- `int nr1;`
- `int *ptr;`
- `ptr = &val;`
- the & operator retrieves the address of nr1
- the dereferencing operator is * and it is used as follows:

```
  *ptr = 2;
```

- puts 2 to the address pointed to by ptr. When we use * we refer to the value pointed by ptr, not the value of the pointer itself.

Pointers and Arrays

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int vec[3];
  vec[0] = 1;
  vec[1] = 2;
  vec[2] = 3;
  printf("vec[2]=%d\n", vec[2]); //  same as
                    // printf("vec[2]=%d\n",
*(vec+2));
  return 0;
}
```

Pointers and Arrays

•When declaring an array a block of memory is allocated large enough to hold the intended number of that type (eg: integers) the variable is a pointer that points to the first element in the array. Indexing with the expression printf("vec[2]=%d\n", vec[2]);
•C is using pointer arithmetic to step into the array the appropriate number of times, and reading the value in the memory location it ends up in.
•When accessing 3rd element of the array using vec[2] then C is looking at the address pointed to by vec (the first element of the array), and steps 2 integers forward reading the value it finds there.

Pointers and Arrays

- As we said, first element in array is the same with the array name:

  `ptr = &vec[0];` is equivalent to

  ```
  ptr = vec;
  ```
- But, vec is not a pointer, is the address of the first element of vec !!!

  ```
  vec = ptr;
  ```
- is illegal, vec is a constant (unmodifiable lvalue)

Pointers - Operations

A pointer, constructed either as a variable or function parameter, contains a value: an address

Pointers Arithmetic

- In C there are some arithmetic operations allowed when using pointers:

Consider ptr a pointer to a type t and i an integer. The expressions ptr + i and ptr - i have as result a pointer with value greater or less by i*sizeof(type) then ptr. For example:

```
int *ptr;
ptr + 3
```
means address of p + 12 bytes

- Consider the case:

```
int vec[3];
int *ptr;
ptr = vec;
```

- We know that vec[i] is the same as *(ptr+i). Wherever one writes a[i] it can be replaced with *(a + i). This is NOT saying that pointers and arrays are the same thing, they are not. We only say that to identify a given element of an array we have the choice of two syntaxes, one using array indexing and the other using pointer arithmetic, with identical results.

Pointers Arithmetic

- Looking at(a + i), the rules of C state that addition is commutative. That is (a + i) is identical to (i + a). Thus we could write *(i + a) just as easily as *(a + i).
- But *(i + a) could have come from i[a] ! What about:

```
#include <stdio.h>
int main(void)
{
    char a[10]="A string";
    a[3]='x';
    2[a]='y';
    puts(a);
    return 0;
}
```