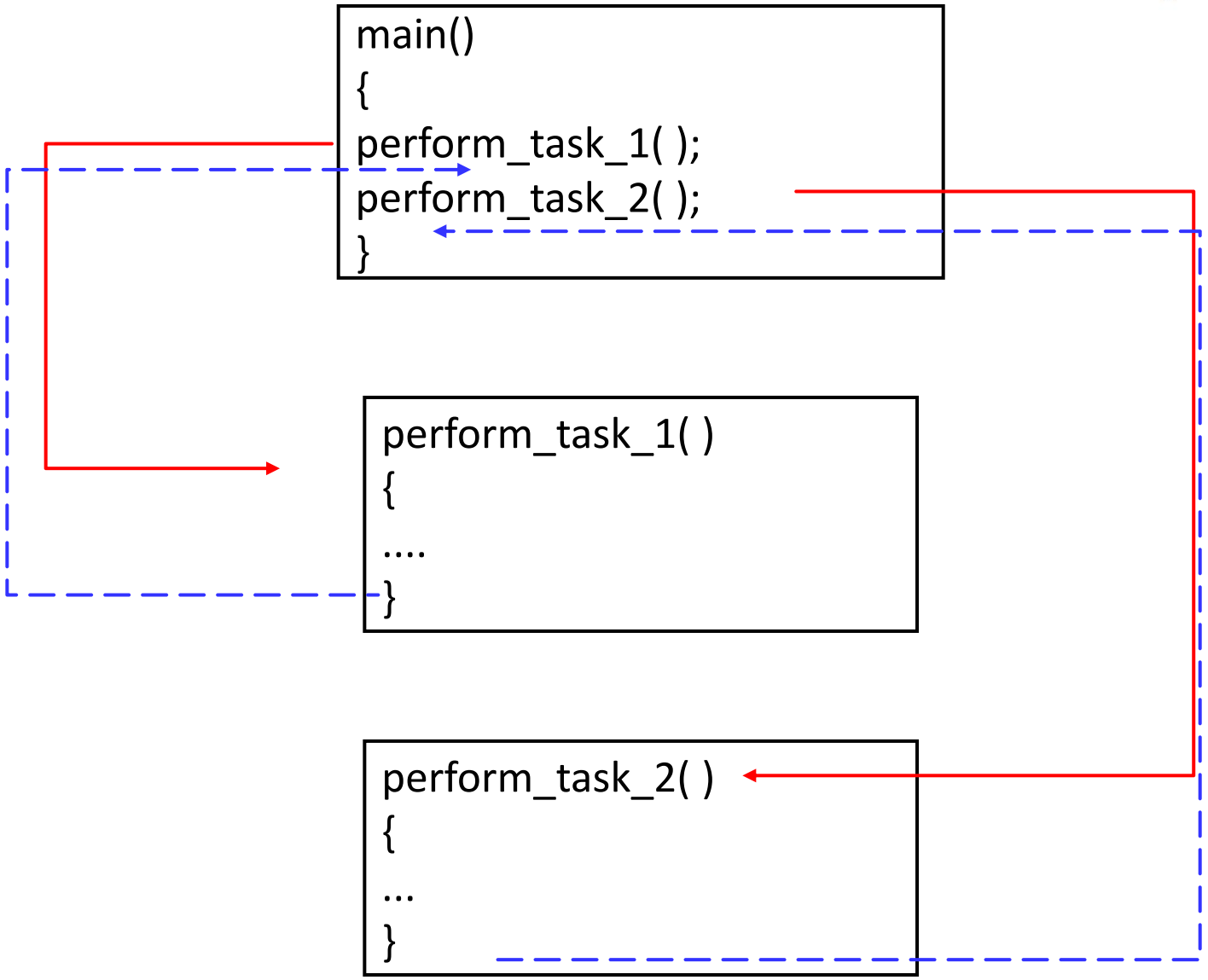
אוניברסיטת
אריאל
בשומרון

Functions and Scopes

אוניברסיטת
אריאל
בשומרון

Functions

Function Call - Diagram



Function definition

- The first line of a function is called the function header.
- General syntax of the function header

```
Output_type Output_parameters Function_name(Input_type Input_parameters)
```

- **Output**
  - The type of the value that the function returns.
  - If no return type is specified, **int** is the default.
  - If the function only performs some task and has no value to return - void is the return type.

- **Input**
  - Input parameters are variables that are initialized with values passed to the function.
  - () – empty parentheses, in case that the function has no parameters.
  - (void) – same as empty parentheses.

Function Body

- The function body contains:
  - Definitions of local variables are optional, but are usually present.
  - Executable statements, which may be any statement in C.

- If the function returns a value, the *return* statement is obligatory. It may occur in a function more than once.
  - General syntax - *return (expression);*
  - The parentheses are optional.

- If the function returns no value, it can still be exited before its' end,
  - General syntax - *return;*

Function Parameters

אוניברסיטת אריאל בשומרון

- Let's rewrite our function **typerow( ),** so that it will print out a varying number of whichever char requested by the calling function. For example, **typerow(26, '$');** will result in 26 dollar signs in a row .

- The prototype - declaration of a function - **typerow(int number, char ch);** informs the compiler that **typerow()** receives an argument of type **int** and an argument of type char. An attempt to invoke the function with other parameters will cause a compilation error.

Function Parameters

אוניברסיטת אריאל בשומרון

```
void typerow(int number, char ch)
{
        int counter;
        for(counter = 1; counter <= number ; ++counter)
                putchar(ch);
        putchar('\n');
}
```

Exiting a Function

אוניברסיטת אריאל בשומרון

- *return* - one way to leave a function
- The control returns to the calling function
- *exit* - another way to leave a function (and the whole program).
  - General syntax - void exit(int status);
  - The function exit() (stdlib.h) terminates the program.
  - The return value may be 0 (usually to indicate success), or a non-zero value (usually to indicate failure).
  - If written in main(), return and exit() have the same effect.

Returning a Value From a Function

אוניברסיטת אריאל בשומרון

- abs is a function that returns the absolute value of an integer.
- Intuitively, we want to use the following:

x = abs(a);

- Where a is the argument passed to the function abs(), which returns the number x - stripped of any negative sign.

Returning a Value From a Function- example


```
1   #include <stdio.h>

2   int abs(int num); // function declaration

3   void main(void){
4           int a=10, b=0, c=-22;
5           int x, y, z;
6           x = abs(a);
7           y = abs(b);
8           z = abs(c);
9           printf("%d %d %d\n", x,  y,  z);
10  }
11
12  int abs(int num){
13          int absnum;
14          absnum = (num < 0) ?  -num : num;
15          return(absnum);
16  }
```

Call by Value

אוניברסיטת אריאל בשומרון

- 'actual parameter' - the parameter that is sent from the calling function.
- 'formal parameter' or 'argument' - the parameter received by the function
- When sending a variable to a function we ALWAYS send it by value.
- The parameter that the function receives is a completely new variable.
- The only connection between the formal parameter and the actual parameter is the value - when the formal parameter is created, it is automatically initialized with the value of the actual parameter.
- Changes to the formal parameter can't effect the actual parameter, since they are absolutely not connected, each of them having its own address in the computer's memory.

Call by Reference

- Sometimes we want the parameter that was sent to a function to be changed by that function.
- When passing a variable by reference it's address is passed, rather than it's value, and so the called function can change the variable's value.
- Default pass –
  - By value – for a scalar argument, simple variable. We shall later see how call by reference is implemented with an argument which is a scalar variable.
  - By reference - if the argument is an array

Call by Value vs. Call by Ref.

אוניברסיטת אריאל בשומרון

By Value	By Reference
The value of the actual parameter is not changed (even if the formal parameter is changed).	Any change to the formal parameter is reflected in the actual parameter.
Argument may be any expression : **-3, i, 2 * a + b, -123, 8.23, sqrt(abs(n)) .**	Argument must be an address of a variable.

אוניברסיטת
אריאל
בשומרון

Variable Scope

Scope and Life Time of Variables

אוניברסיטת אריאל בשומרון

- Scope of a variable - the part of the program in which the variable is visible (i.e. may be used).
- Life time of a variable - the time between the 'birth' of the variable on the computers memory and the time it 'dies'.
- There are two broad classifications of scope:
  - Global scope
  - Local scope - automatic local (e.g. stack), static local.

- When referencing a variable, the compiler searches for such a variable name within the immediate local scope. If not found, then the next "higher" global scope is searched.

Local Variables

אוניברסיטת אריאל בשומרון

- Variables that are declared inside a function or block are called local variables.
- They can be used only by statements that are inside that function or block of code.
- Local variables are not known to functions outside their own.

```c
#include <stdio.h>

int main () {

  /* local variable declaration */
  int a, b;
  int c;

  /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;

  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

  return 0;
}
```

Global Variables

- Global variables are defined outside a function, usually on top of the program.
- Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

```c
#include <stdio.h>

/* global variable declaration */
int g;

int main () {

  /* local variable declaration */
  int a, b;

  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;
}
```

Local / Global preference

- A program can have same name for local and global variables but the value of local variable inside a function will take preference.

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main () {

  /* local variable declaration */
  int g = 10;

  printf ("value of g = %d\n",  g);

  return 0;
}
```

Global vs. Automatic Local – cont'd

אוניברסיטת אריאל בשומרון

	Global	Automatic Local
Declaration	outside any function	in the block { }
Initialization	Unless specified otherwise, automatically initialized to 0	No automatic initialization. If specifically initialized - it occurs at each 'birth'
Scope	functions beneath it, in the same module	its block only
'Birth'	once, before **main()**	each time the block is entered
'Death'	once, after **main()** ends	each time the block is exited
Address	on the data area	on the stack area

Local / Global initialization

- When a local variable is defined, it is not initialized by the system, you must initialize it yourself.
- Global variables are initialized automatically by the system according to the table below -

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

Scope Diagram – Global or Local ?


```
int num = 42;
void func(void)
{
   int other_num = 10;
   printf ("%d %d", num, other_num);
}
void main()
{
   int num = 1, other_num = 2;
   printf ("%d %d", num, other_num);
   {
      int other_num = 3;
      printf ("%d %d", num, other_num);
   }
   printf ("%d %d", num, other_num);
   func();
}
```

Example: scope.c

```
1   /* scope.c
2   This program demonstrates the scope of variables */
3
4   #include <stdio.h>
5
6   void func(void); /* function prototype*/
7
8   /* global variables, defined outside any function. All the
9   functions in this file that are written below them can see them */
10  int Tom;
11  int Jerry;
12
13  void main(void)
14  {
15          int main_local = 100; /* local variable in main */
16
17          Tom = Jerry = main_local = 1;  /*  change the two globals and the local */
18
```

Example: scope.c – cont'd


```
20          printf("In main, before calling func(): Tom=%d,"
21      "Jerry=%d, main_local=%d\n",Tom, Jerry, main_local);
22
23      func();    /* call the function func() */
24
25          printf("In main, after calling func():  Tom=%d, "
26      "Jerry=%d, main_local=%d\n",Tom, Jerry, main_local);
27  }
28
29  void func(void)
30  {
31          int i; /* this variable (i) is local in function func(). */
32          int Jerry; /* local to the function func().When we refer to
33          Jerry, the compiler finds it in the function's scope, and
34          doesn't look any further. This function can never see the
35          global Jerry */
```

Example: scope.c – cont'd

אוניברסיטת אריאל בשומרון

```
35          Tom = Jerry = i = 1000; /* this increases the global variable Tom */
36          printf("In func(), before block : Tom=%d, "Jerry=%d,i=%d\n", Tom, Jerry, i);
37          {
38  /* local to the block. This block can't see any Jerry that is out of this block */
39          int Jerry = 2000;
40          Tom = i = 2000;
41          printf("In the block : Tom = %d, Jerry = %d, i = "%d\n", Tom, Jerry, i);
42          } /* The block ends here */
43          printf("In func() after block : Tom=%d, Jerry=%d,"i=%d\n", Tom, Jerry, i);
44  }
```

Example: Exercise


```
1   /* scope2.c - Fill in the blank spaces. If the line compiles, write
    the output.  If it doesn't, write the reason. */
2
3   #include <stdio.h>
4
5   int  global = 100;
6   void func();
7
8   void main(void)
9   {
10          int  main_scope = 10, i = 20;
11
12          printf("%d/n", j);   /* ______ */
13          printf("%d %d %d", main_scope, i, global);  /* _____ */
14          func();
15          printf("%d", j);   /* ______ */
19          printf("%d %d %d\n", main_scope, i, global);  /* _____ */
20          printf("%d\n", other_global);
21  }
```

Example: Exercise – cont'd

אוניברסיטת אריאל בשומרון

```
19  int other_global = 200;
20
21  void func()
22  {
23          int i = 3, j = 4;
24          {
25                  int i = 5;
26                  printf("%d %d %d\n", i, j, other_global);   /* _____ */
27          }
28          printf("%d %d %d %d\n", i, j, global, other_global); /* _____ */
29          printf("%d", main_scope);  /* _____ */
30  }
```

Global Variables – Be careful

אוניברסיטת אריאל בשומרון

- Avoid global variables
  - It is hard to predict the contents of a global variable, since many functions can change it.
  - It captures memory throughout the lifetime of the whole program.
  - The behavior of a function should be predicted, by the arguments that it gets. If the functions behavior also depends on a global variable, it is not 100% independent and it is more confusing to track its output.

Static Local Variables

אוניברסיטת אריאל בשומרון

- Maintains its value from one function call to another.
- In other words, the variable should be born once and die once, regardless of the number of times the function is called.
- The characteristics of a static local variable are the same as those of the global, except from the scope, which is like automatic local.
- General syntax - static var_type var_name

```
void func(void)
{
        static int num = 3;
        .....
 }
```

Local Storage Classes

- There are two local storage classes :
  - Automatic
  - Static
- The automatic local variables - automatic storage class (Stack).
- The static local variables and global variables - static storage class.
- The characteristics of the automatic storage class are those described in the table for the automatic local variables.
- The characteristics of the static storage class are those described in the table for the global variables. Except for the scope

	Global	Automatic Local
Declaration	outside any function	in the block { }
Initialization	Unless specified otherwise, automatically initialized to 0	No automatic initialization. If specifically initialized - it occurs at each 'birth'
Scope	functions beneath it, in the same module	its block only
'Birth'	once, before **main()**	each time the block is entered
'Death'	once, after **main()** ends	each time the block is exited
Address	on the data area	on the stack area

Example: static-vs-local1.c


```
1   /* static-vs-local1.c
2   This program illustrates the difference between: static local and
    automatic local variables */
3
4   #include <stdio.h>
5
6   void func(void);
7
8   void main(void)
9   {
10          int i;
11
12          for (i = 1; i <= 5; i++) /* call func() five times */
13                  func();
14          printf("\n");
15  }
```

Example: static-vs-local1.c – cont'd

אוניברסיטת אריאל בשומרון

```
17  void func(void)
18  {
19          int aut = 0; /* local variable. Automatic storage class by default. Created
            each time this block starts, destroyed each time this block ends */
20
21          static int stat = 0; /* initialized to 0 anyway…, Created before main(),
            destroyed after the program ends */
22
23          aut++;
24          stat++;
25          printf("automatic = %d      stat = %d\n", aut,stat);
26  }
```

Example: static-vs-local1.c – cont'd

aut = 1	stat = 1
aut = 1	stat = 2
aut = 1	stat = 3
aut = 1	stat = 4
aut = 1	stat = 5

Example: static-vs-local2.c

אוניברסיטת אריאל בשומרון

```
1   /* static-vs-local2.c
2   This program illustrates the difference between: static local and
    automatic local variables */
3
4   #include <stdio.h>
5
6   void func(void); /* function prototype*/
7
8   void main(void)
9   {
10          int i;
11
12          for (i = 1; i <= 5; i++) /* call func() five times */
13                  func();
14          printf("\n");
15  }
```

Example: static-vs-local2.c – cont'd


```
17  void func(void)
18  {
19          int  aut = 0;  /* Local variable. Automatic storage class by
default. Created each time this block starts, destroyed each time
this block ends */
17
18          static int stat;   /* initialized to 0 any way….Created
19                      before main(), destroyed after the program ends */
20
21          stat = 0;  /*This line didn't appear in the previous program */
22          aut++;
23          stat++;
24          printf("automatic = %d      stat = %d\n", aut,stat);
25  }
26
```

Example: static-vs-local2.c – cont'd

aut = 1	stat = 1
aut = 1	stat = 1
aut = 1	stat = 1
aut = 1	stat = 1
aut = 1	stat = 1

Summary

אוניברסיטת אריאל בשומרון

- Function - an autonomous code segment that accomplishes a particular task.
- A function may receive parameters and may return a value.
- Actual parameters – sent.
- Formal parameters (arguments) – received.
- Variables may be passed as parameters to a function :
  - By value –
    - ✓ The value of the variable is passed.
    - ✓ Changing the formal parameter does not effect the actual parameter.
    - ✓ Scalar variables are sent this way by default.
  - By reference -
    - ✓ The address of the variable is passed.
    - ✓ Changing the formal parameter does effect the actual parameter.
    - ✓ Arrays can only be sent this way.

Summary – cont'd

- There are 2 storage classes :
  - Static
    - ✓ Static local.
    - ✓ Global (may be called – extern global).
    - ✓ Static global.
  - Automatic
    - Automatic local.
- The arguments received by a function are automatic local variables of that function.
- A program should be split to several modules.
- The keyword **extern** help sharing global variables or functions between different modules of the program.
- The keyword **static** makes a global variable or a function private to a certain module.

Example: Tic-Tac-Toe.c

אוניברסיטת אריאל בשומרון

```
1   /* A program for playing Tic-Tack-Toe, using functions. */
2
3   #include <stdio.h>
4   #define TRUE 1
5   #define FALSE 0
6   #define WIN 1
7   #define NO_WIN 0
8
9   int play(char sign, char mat[3][3]);
10  int get_coordinate(char sign, char direction[]);
11  int check_legal_coords(int row, int col, char mat[3][3]);
12  int check_win(int row, int col, char sign, char mat[3][3]);
13  void print_mat(char mat[3][3]);
14  void declare_winner(char sign);
```

Example: Tic-Tac-Toe.c – cont'd

אוניברסיטת אריאל בשומרון

```
16  void main(void) {
17          char mat[3][3] = {0}; /* the game board */
18          int flag = NO_WIN; /* will turn to WIN when someone wins */
19          char sign = '*'; /*the sign of the first player.'*' starts */
20          printf ("The player who starts will be player '*', the other player will be player 'o'.\n"
21                  "The first player who fills a whole row/column/diagonal with his sign wins.\n");
22
23          while(flag == NO_WIN) { /*loop until someone wins the game */
24                  flag = play(sign,mat); /* sent play() the board and current player*/
25                  if(flag == WIN)      /* check the value that play()returned */
26                          declare_winner(sign);
27                  sign = (sign == '*' ? 'o' : '*');   /* turn to the other player */
28          }
29  }
```

Example: Tic-Tac-Toe.c – cont'd

אוניברסיטת אריאל בשומרון

```
36  int play(char sign, char mat[3][3]){
37          int row, col; /* the coordinates for the user's next move */
38          int flag = NO_WIN; /*will turn into WIN if the player wins*/
39          int legal_coords = FALSE;
40          while(! legal_coords) { /* loop until legal coordinates are entered */
41                  row = get_coordinate(sign, "row");
42                  col = get_coordinate(sign, "col");
43                  legal_coords = check_legal_coords(row, col,mat);
44          }
45          mat[row][col] = sign; /* put the sign on the game board */
46          flag = check_win(row, col, sign, mat); /*check if this player has won */
47          print_mat(mat); /* print the game board */
48          return flag;
49  }
```

Example: Tic-Tac-Toe.c – cont'd

אוניברסיטת אריאל בשומרון

```
56  void declare_winner(char sign)
57  {
58          printf ("Player \'%c\' won !\n", sign);
59  }
60
61  int get_coordinate(char sign, char direction[]){
62          int coord = -1;
63          while(! (coord >= 0 && coord <= 2) )
64          {   /* the coordinate must be in the borders of the matrix */
65                  printf("\nPlayer %c, Enter a %s => ", sign,
66                          direction);
67                  scanf("%d", &coord);
68          }
69          return coord;
70  }
```

Example: Tic-Tac-Toe.c – cont'd

אוניברסיטת אריאל בשומרון

```
73  int check_legal_coords(int row, int col, char mat[3][3]) {
74  /* returns 1 if the place on the board is free, 0 otherwise */
75          return (!mat[row][col]);
76  }
77  int check_win(int row, int col, char sign, char mat[3][3])
78  {
79          int win = FALSE;
80          /* check if it is a winning row */
81          if(mat[row][(col+1)%3] == sign && mat[row][(col+2)%3]== sign)
82                  win = TRUE;
83
84          /* check if it is a winning column */
85          if(mat[(row+1)%3][col] == sign && mat[(row+2)%3][col]== sign)
86                  win = TRUE;
```

Example: Tic-Tac-Toe.c – cont'd

אוניברסיטת אריאל בשומרון

```
90  /* check if there is a winning diagonal.In order to have a
91  wining diagonal, the player must 'own' the center of the matrix. */
92          if(mat[1][1] == sign){
93                  if(mat[0][0] == sign && mat[2][2] == sign)
94                          win = WIN;
95                  if(mat[0][2] == sign && mat[2][0] == sign)
96                          win = WIN;
97          }
98          return win;
99  }
100 void print_mat(char mat[3][3]){
101         int row, col;
102         for(row = 0 ; row < 3 ; ++row){
103                 for(col = 0 ; col < 3 ; ++col)
104                         printf("%3c", mat[row][col] ?
105                 mat[row][col] : '-');
106                 putchar('\n');
107         }
108 }
```

TA Exercise

אוניברסיטת אריאל בשומרון

- Write a function that returns the number of digits of a long integer (which may be positive, negative or zero).
  a. The **main( )** of the program will accept two such integers, print them and the number of digits in each of them.
  b. The **main( )** will also print the product of the entered numbers and the number of digits in the product.

- Write a function with 2 parameters. The first is an array (of **int**) and the second is the number of elements in the array. The function will delete all zeros from the array and return the number of remaining elements (that is, the number of non-zero elements).
  a. The **main( )** will print, nicely - including indexes, the array before and after the deleting of zeros.
  b. The array in **main( )** that is used as an argument will be defined and initialized, something like:

     int v[ ] ={0,5,2,0,7,3,0,0,33,0,0,0};

     Hint: you will require the number of elements of v. Use operator **sizeof**, twice, to calculate it.

TA Exercise – cont'd

אוניברסיטת אריאל בשומרון

- Write a function with 2 parameters:
  a. The first is a 2-dimensional array (matrix) with 4 columns, the second being the number of rows.
  b. The function will return the minimum of the maximums of the lines of the first parameter.
     Note: The declaration of the first parameter must include the number of columns (but not of rows).
  c. Write **main( )** to call that function. The matrix used as an argument is defined and initialized so:

```
int mat[ ][4] =        {{ 2, -4, 14, 5 },
                        {17, -27, 9,  0 },
                        { 1,  2,  6, -1 },
                        { 2,  8, -9,  7 }
                        {11, -2, -3, -1 }};
```

TA Exercise – cont'd

אוניברסיטת אריאל בשומרון

- Take any 4 numbers for instance: 10 5 7 1,

- Produce series of (cyclic) subtractions until 0 0 0 0 is produced. In this example:

5 2 6 9

3 4 3 4

1 1 1 1

0 0 0 0

the length of the series until zeros are produced is 4.

- Write a program that produces the 4 numbers (in the range 5 to 10) that generate the longest series. Write the program so that it can be changed very easily to work with a different range.

TA Exercise – cont'd

אוניברסיטת אריאל בשומרון

- Write a program that accepts input lines until **EOF** is encountered and:

  a. Finds the words within the line.

  b. Prints a list of the words appearing in the input and the Number of times that they appear.

  c. Note: before writing the program, design your data structure and the functions to be used.