

CPE101 Programming Languages I

Week 13 Functions

Assoc. Prof. Dr. Caner ÖZCAN

Functions

► Functions

- Modules in C
- Programs combine user-defined functions with library functions
- C standard library has a wide variety of functions

Benefits of Functions

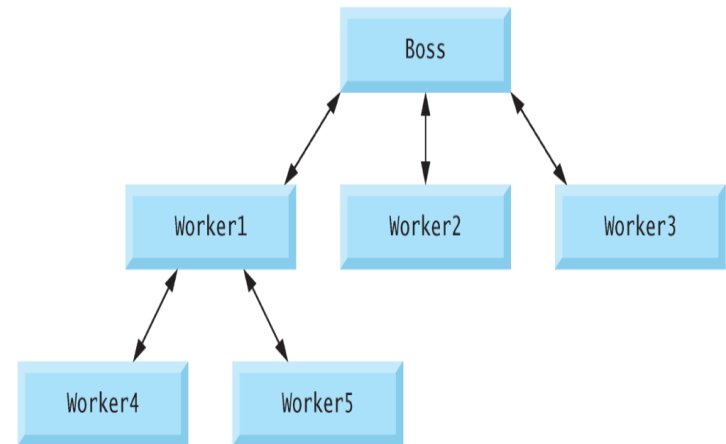
- Benefits of Functions
 - Divide and conquer
 - Construct a program from smaller pieces or components
 - These smaller pieces are called modules.
 - Functions allow you to modularize a program.
 - Experience has shown that the best way to develop and maintain a large program is to construct it from smaller pieces or **modules**, each of them is more manageable than the original program.
 - Software reusability
 - Use existing functions as building blocks for new programs
 - Abstraction - hide internal details (library functions)
 - Avoid code repetition

Functions

- The variables defined in a function are the local variables of this function.
 - Only known in the body of the function
- Parameters
 - Most functions have a list of **parameters** that provide the means for communicating information between functions
 - Also local variables of the function
- Function calls
 - Provide function name and arguments (data)
 - Function performs operations or manipulations
 - Function returns results

Functions

- Function call analogy:
- Boss asks worker to complete task
- Worker gets information, does task, returns result
- Information hiding: boss does not know details



Defining Functions

- Format of a function definition :

```
return_value_type function_name ( parameter_list )  
{  
    definitions_and_statements  
}
```

- *function_name* is any valid identifier.
- *return_value_type* is the data type of the result returned to the caller
- *return_value_type* void indicates that a function does not return a value.
- Together, the *return_value_type*, *function_name* and *parameter_list* are referred to as the function header.

Defining Functions

- **parameter_list** is a comma-separated list that specifies the parameters received by the function when it's called.
- If a function does not receive any values, *parameter-list* is **void**.
- A type must be listed explicitly for each parameter

Defining Functions

- The *definitions_and_statements* within fancy parentheses form the *function body*.
- The function body is also referred to as a **block**.
- Variables can be declared in any block, and blocks can be nested.
- **A function cannot be defined inside another function.**

Defining Functions

- There are three ways to return control from a called function to the point at which a function was invoked.
- If the function does not return a result
 - Control is returned simply when the function-ending right fancy bracket is reached.
 - or by executing the statement `return`;
- If the function does return a result, the statement `return expression;` returns the value of *expression to the caller*.

Function Prototype

- Identity of a function.
- Prototype only needed if function definition comes after use in program.
- The function that has a prototype given below:
 - `int maximum(int x, int y, int z);`
 - Takes 3 integer parameters.
 - Returns integer value.

Function Prototype

- If a function call does not match the function prototype compilation error is produced.
- An error is also generated if the function prototype and the function definition disagree.
- Another important feature of function prototypes is the **coercion of arguments**, i.e., the forcing of arguments to the appropriate type.
- For example, the math library function `sqrt` can be called with an integer argument even though the function prototype in `<math.h>` specifies a double argument, and the function will still work correctly.
 - The statement ;
 - `printf("%.3f\n", sqrt(4));`
 - correctly evaluates `sqrt(4)`, and prints the value 2.000

Defining Functions

```
1  /* Fig. 5.4: fig05_04.c
2     Finding the maximum of three integers */
3  #include <stdio.h>
4
5  int maximum( int x, int y, int z ); /* function prototype */
6
7  /* function main begins program execution */
8  int main( void )
9  {
10     int number1; /* first integer */
11     int number2; /* second integer */
12     int number3; /* third integer */
13
14     printf( "Enter three integers: " );
15     scanf( "%d%d%d", &number1, &number2, &number3 );
16
17     /* number1, number2 and number3 are arguments
18        to the maximum function call */
19     printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );
20     return 0; /* indicates successful termination */
21 } /* end main */
22
```

Defining Functions

```
23  /* Function maximum definition */
24  /* x, y and z are parameters */
25  int maximum( int x, int y, int z )
26  {
27      int max = x; /* assume x is largest */
28
29      if ( y > max ) { /* if y is larger than max, assign y to max */
30          max = y;
31      } /* end if */
32
33      if ( z > max ) { /* if z is larger than max, assign z to max */
34          max = z;
35      } /* end if */
36
37      return max; /* max is largest value */
38  } /* end function maximum */
```

Fig. 5.4 | Finding the maximum of three integers. (Part 3 of 4.)

Header Files

- Each standard library has a corresponding **header** containing the function prototypes and definitions of various data types.
- `<stdlib.h>` , `<math.h>` , etc
- Load with `#include <file name>`
 - `#include <math.h>`
- Custom header files
 - Create file with functions.
 - Save as `filename.h`
 - Load in other files with `#include "filename.h"`
 - Reuse functions.

Header Files

- **math.h** → Mathematics library functions
- **ctype.h** → Functions for testing characters for certain properties, functions to convert into uppercase or lowercase etc.
- **stdio.h** → Standard input/output functions
- **stdlib.h** → Functions for converting numbers to text or text to number, memory management, random number generation and other utility functions.
- **string.h** → String processing functions
- **time.h** → Time and date functions

Mathematic Library Functions

- Mathematic Library Functions
 - Perform common mathematical calculations.
 - `#include <math.h>`
- Format for calling functions
 - `Function_name(arguments);`
- If multiple arguments, use comma-separated list
- All math functions return data type `double`
- Arguments may be constants, variables, or expressions

Mathematic Library Functions

Function	Description	Example
<code>sqrt(x)</code>	square root of x	<code>sqrt(900.0)</code> IS 30.0 <code>sqrt(9.0)</code> IS 3.0
<code>exp(x)</code>	exponential function e^x	<code>exp(1.0)</code> IS 2.718282 <code>exp(2.0)</code> IS 7.389056
<code>log(x)</code>	natural logarithm of x (base e)	<code>log(2.718282)</code> IS 1.0 <code>log(7.389056)</code> IS 2.0
<code>log10(x)</code>	logarithm of x (base 10)	<code>log10(1.0)</code> IS 0.0 <code>log10(10.0)</code> IS 1.0 <code>log10(100.0)</code> IS 2.0
<code>fabs(x)</code>	absolute value of x	<code>fabs(13.5)</code> IS 13.5 <code>fabs(0.0)</code> IS 0.0 <code>fabs(-13.5)</code> IS 13.5
<code>ceil(x)</code>	rounds x to the smallest integer not less than x	<code>ceil(9.2)</code> IS 10.0 <code>ceil(-9.8)</code> IS -9.0
<code>floor(x)</code>	rounds x to the largest integer not greater than x	<code>floor(9.2)</code> IS 9.0 <code>floor(-9.8)</code> IS -10.0

Mathematic Library Functions

Function	Description	Example
<code>pow(x, y)</code>	x raised to power y (x^y)	<code>pow(2, 7)</code> is 128.0 <code>pow(9, .5)</code> is 3.0
<code>fmod(x, y)</code>	remainder of x/y as a floating-point number	<code>fmod(13.657, 2.333)</code> is 1.992
<code>sin(x)</code>	trigonometric sine of x (x in radians)	<code>sin(0.0)</code> is 0.0
<code>cos(x)</code>	trigonometric cosine of x (x in radians)	<code>cos(0.0)</code> is 1.0
<code>tan(x)</code>	trigonometric tangent of x (x in radians)	<code>tan(0.0)</code> is 0.0

Example: Square function

```
#include <stdio.h>
float kareAl(float);

void main()
{
    int sayac;
    for(sayac = 1; sayac<=10; sayac++)
    {
        printf("Sayi:%d Karesi:%d\n", sayac, kareAl(sayac));
    }

    printf("\n%.2f", kareAl(4.5));
}

float kareAl(float a)
{
    return a*a;
}
```

Example: Arithmetic functions

```
#include <stdio.h>
int toplam(int, int);
int cika(int, int);
int carp(int, int);
float bol(int, int);

void main()
{
    int secim,s1,s2;
    while(1)
    {
        printf("1-Topla\n2-Cika\n3-Carp\n4-Bol\n5-Cikis\n");
        scanf("%d", &secim);
        printf("Sayilari gir:");
        scanf("%d %d", &s1, &s2);

        if(secim == 1)
            printf("Sonuc = %d", toplam(s1,s2));
        else if(secim == 2)
            printf("Sonuc = %d", cika(s1,s2));
        else if(secim == 3)
            printf("Sonuc = %d", carp(s1,s2));
        else if(secim == 4)
            printf("Sonuc = %.2f", bol(s1,s2));
        else if(secim == 5)
            exit(0);
        else printf("Yanlis giris");
    }
}
```

```
int toplam(int a, int b)
{
    return a+b;
}

int cika(int a, int b)
{
    return a-b;
}

int carp(int a, int b)
{
    return a*b;
}

float bol(int a, int b)
{
    return (float)a/b;
}
```

Example: Exponent function

```
#include <stdio.h>
double usAl(double, double);

void main()
{
    double a,b;
    printf("Taban ve us degeri gir:");
    scanf("%lf %lf", &a, &b);
    printf("%.2f", usAl(a,b));
}

double usAl(double x, double y)
{
    int sayac;
    double sonuc=1.0;
    for(sayac=0;sayac<y;sayac++)
    {
        sonuc *= x;
    }
    return sonuc;
}
```

Passing Arrays to Functions

- To pass an array argument to a function, specify the name of the array without any brackets.
 - **int** myArray [**24**];
 - **myFunction** (myArray, 24);
- Unlike char arrays, other array types do not have a special terminator.
- Therefore, the size of the array is usually passed to the functions so functions can process proper number of elements

Passing Arrays to Functions

- Arrays passed **call-by-reference**
- Name of array is the address of the first element
- Function knows where the array is stored in memory.
 - Modifies original memory location.
- Passing an element to a function is **call-by-value**
 - Pass subscripted name to function
 - `myArray [3]`
- Function prototype that takes int array and int value and returns nothing;
 - `void myArray (int [], int)`

Passing Arrays to Functions

```
1  /* Fig. 6.13: fig06_13.c
2     Passing arrays and individual array elements to functions */
3  #include <stdio.h>
4  #define SIZE 5
5
6  /* function prototypes */
7  void modifyArray( int b[], int size );
8  void modifyElement( int e );
9
10 /* function main begins program execution */
11 int main( void )
12 {
13     int a[ SIZE ] = { 0, 1, 2, 3, 4 }; /* initialize a */
14     int i; /* counter */
15
16     printf( "Effects of passing entire array by reference:\n\nThe "
17            "values of the original array are:\n" );
18
19     /* output original array */
20     for ( i = 0; i < SIZE; i++ ) {
21         printf( "%3d", a[ i ] );
22     } /* end for */
23 }
```


Passing Arrays to Functions

```
24  printf( "\n" );
25
26  /* pass array a to modifyArray by reference */
27  modifyArray( a, SIZE );
28
29  printf( "The values of the modified array are:\n" );
30
31  /* output modified array */
32  for ( i = 0; i < SIZE; i++ ) {
33      printf( "%3d", a[ i ] );
34  } /* end for */
35
36  /* output value of a[ 3 ] */
37  printf( "\n\nEffects of passing array element "
38         "by value:\n\nThe value of a[3] is %d\n", a[ 3 ] );
39
40  modifyElement( a[ 3 ] ); /* pass array element a[ 3 ] by value */
41
42  /* output value of a[ 3 ] */
43  printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );
44  return 0; /* indicates successful termination */
45 } /* end main */
46
```

Passing Arrays to Functions

```
47  /* in function modifyArray, "b" points to the original array "a"
48     in memory */
49  void modifyArray( int b[], int size )
50  {
51     int j; /* counter */
52
53     /* multiply each array element by 2 */
54     for ( j = 0; j < size; j++ ) {
55         b[ j ] *= 2;
56     } /* end for */
57 } /* end function modifyArray */
58
59  /* in function modifyElement, "e" is a local copy of array element
60     a[ 3 ] passed from main */
61  void modifyElement( int e )
62  {
63     /* multiply parameter by 2 */
64     printf( "Value in modifyElement is %d\n", e *= 2 );
65 } /* end function modifyElement */
```

Passing Arrays to Functions

Effects of passing entire array by reference:

The values of the original array are:

0 1 2 3 4

The values of the modified array are:

0 2 4 6 8

Effects of passing array element by value:

The value of a[3] is 6

Value in modifyElement is 12

The value of a[3] is 6

Passing Multiple Dimensional Arrays to Functions

- Not different from passing single subscripted arrays to functions.
- Just indicate rectangle brackets for each dimension and specify the sizes for all dimensions other than first dimension.
 - `void writeMatrice (int [] [4], int rowNumber);`
 - This definition will work for all matrices (with different row numbers) having 4 columns.
 - `void writeMatrice (int [] [3] [4], int rowNumber);`

Passing Multiple Dimensional Arrays to Functions

```
1  /* Fig. 6.21: fig06_21.c
2     Initializing multidimensional arrays */
3  #include <stdio.h>
4
5  void printArray( const int a[][ 3 ] ); /* function prototype */
6
7  /* function main begins program execution */
8  int main( void )
9  {
10     /* initialize array1, array2, array3 */
11     int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
12     int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
13     int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
14
15     printf( "Values in array1 by row are:\n" );
16     printArray( array1 );
17
18     printf( "Values in array2 by row are:\n" );
19     printArray( array2 );
20
21     printf( "Values in array3 by row are:\n" );
22     printArray( array3 );
23     return 0; /* indicates successful termination */
24 } /* end main */
```

Passing Multiple Dimensional Arrays to Functions

```
25
26 /* function to output array with two rows and three columns */
27 void printArray( const int a[][ 3 ] )
28 {
29     int i; /* row counter */
30     int j; /* column counter */
31
32     /* loop through rows */
33     for ( i = 0; i <= 1; i++ ) {
34
35         /* output column values */
36         for ( j = 0; j <= 2; j++ ) {
37             printf( "%d ", a[ i ][ j ] );
38         } /* end inner for */
39
40         printf( "\n" ); /* start new line of output */
41     } /* end outer for */
42 } /* end function printArray */
```

values in array1 by row are:
1 2 3
4 5 6
values in array2 by row are:
1 2 3
4 5 0
values in array3 by row are:
1 2 0
4 0 0

Storage Classes

- **Automatic Storage**

- Object created and destroyed within its block

- **auto**: default for local variables

- `auto double x, y;`

- **register**: tries to put variable into high speed registers

- `register int counter= 1;`

Storage Classes

- **Static Storage**
 - Variable exists for entire program execution
 - Default value of zero.
 - **static**: local variables defined in functions.
 - Keep value after function ends
 - Only known in their own function

Storage Classes

- **File Storage**

- An identifier declared outside of a function has **file scope**.
- Such an identifier is known in all functions from the point at which identifier is declared until the end of file
- Global variables, function definitions placed outside a function all have file scope.

Storage Classes

- **Block Scope**

- Identifier declared inside a block
- Block scope begins at definition, ends at right brackets.
- Used for variables, local variables of function.
- Outer blocks hidden from inner blocks if there is a variable with the same name in the inner block.

Storage Classes

```
1  /* Fig. 5.12: fig05_12.c
2     A scoping example */
3  #include <stdio.h>
4
5  void useLocal( void ); /* function prototype */
6  void useStaticLocal( void ); /* function prototype */
7  void useGlobal( void ); /* function prototype */
8
9  int x = 1; /* global variable */
10
11 /* function main begins program execution */
12 int main( void )
13 {
14     int x = 5; /* local variable to main */
15
16     printf("local x in outer scope of main is %d\n", x );
17
18     { /* start new scope */
19         int x = 7; /* local variable to new scope */
20
21         printf( "local x in inner scope of main is %d\n", x );
22     } /* end new scope */
23 }
```

Storage Classes

```
24  printf( "local x in outer scope of main is %d\n", x );
25
26  useLocal(); /* useLocal has automatic local x */
27  useStaticLocal(); /* useStaticLocal has static local x */
28  useGlobal(); /* useGlobal uses global x */
29  useLocal(); /* useLocal reinitializes automatic local x */
30  useStaticLocal(); /* static local x retains its prior value */
31  useGlobal(); /* global x also retains its value */
32
33  printf( "\nlocal x in main is %d\n", x );
34  return 0; /* indicates successful termination */
35 } /* end main */
36
37 /* useLocal reinitializes local variable x during each call */
38 void useLocal( void )
39 {
40     int x = 25; /* initialized each time useLocal is called */
41
42     printf( "\nlocal x in useLocal is %d after entering useLocal\n", x );
43     x++;
44     printf( "local x in useLocal is %d before exiting useLocal\n", x );
45 } /* end function useLocal */
46
```

Fig. 5.12 | Scoping example. (Part 2 of 4.)

Storage Classes

```
47  /* useStaticLocal initializes static local variable x only the first time
48     the function is called; value of x is saved between calls to this
49     function */
50  void useStaticLocal( void )
51  {
52      /* initialized only first time useStaticLocal is called */
53      static int x = 50;
54
55      printf( "\nlocal static x is %d on entering useStaticLocal\n", x );
56      x++;
57      printf( "local static x is %d on exiting useStaticLocal\n", x );
58  } /* end function useStaticLocal */
59
60  /* function useGlobal modifies global variable x during each call */
61  void useGlobal( void )
62  {
63      printf( "\nglobal x is %d on entering useGlobal\n", x );
64      x *= 10;
65      printf( "global x is %d on exiting useGlobal\n", x );
66  } /* end function useGlobal */
```

Storage Classes

```
local x in outer scope of main is 5  
local x in inner scope of main is 7  
local x in outer scope of main is 5
```

```
local x in useLocal is 25 after entering useLocal  
local x in useLocal is 26 before exiting useLocal
```

```
local static x is 50 on entering useStaticLocal  
local static x is 51 on exiting useStaticLocal
```

```
global x is 1 on entering useGlobal  
global x is 10 on exiting useGlobal
```

```
local x in useLocal is 25 after entering useLocal  
local x in useLocal is 26 before exiting useLocal
```

```
local static x is 51 on entering useStaticLocal  
local static x is 52 on exiting useStaticLocal
```

```
global x is 10 on entering useGlobal  
global x is 100 on exiting useGlobal
```

```
local x in main is 5
```

QUIZ

Klavyeden Integer tipinde tek boyutlu bir dizi alan ve dizide kaç tane çift sayı olduğunu döndüren fonksiyonu ve ana programı yazın.

Write the function and the main program that takes an Integer type one-dimensional array from the keyboard and returns the number of even numbers in the array.

References

- ▶ Doç. Dr. Fahri Vatansever, “Algoritma Geliştirme ve Programlamaya Giriş”, Seçkin Yayıncılık, 12. Baskı, 2015.
- ▶ J. G. Brookshear, “Computer Science: An Overview 10th Ed.”, Addison Wisley, 2009.
- ▶ Kaan Aslan, “A’dan Z’ye C Klavuzu 8. Basım”, Pusula Yayıncılık, 2002.
- ▶ Paul J. Deitel, “C How to Program”, Harvey Deitel.
- ▶ Bayram AKGÜL, C Programlama Ders notları