Week 13
Functions

Assist. Prof. Dr. Caner ÖZCAN
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
  }
  ```

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

• The *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 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 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`. 
Defining Functions

/* Fig. 5.4: fig05_04.c */
Finding the maximum of three integers */
#include <stdio.h>

int maximum( int x, int y, int z ); /* function prototype */

/* function main begins program execution */
int main( void )
{
    int number1; /* first integer */
    int number2; /* second integer */
    int number3; /* third integer */

    printf( "Enter three integers: " );
    scanf( "%d%d%d", &number1, &number2, &number3 );

    /* number1, number2 and number3 are arguments
    to the maximum function call */
    printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );
    return 0; /* indicates successful termination */
} /* end main */
Defining Functions

/* Function maximum definition */
/* x, y and z are parameters */
int maximum( int x, int y, int z )
{
    int max = x; /* assume x is largest */
    if ( y > max ) { /* if y is larger than max, assign y to max */
        max = y;
    } /* end if */
    if ( z > max ) { /* if z is larger than max, assign z to max */
        max = z;
    } /* end if */
    return max; /* max is largest value */
} /* end function maximum */

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

• Identity of a function.  
• The compiler uses function prototypes to validate function calls.  
• Prototype only needed if function definition comes after use in program.  
• Improper function call without control of compiler is avoided with prototype.  
• 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
Promotion Rules

• In general, argument values that do not correspond precisely to the parameter types in the function prototype are converted to the proper type before the function is called.
• These conversions can lead to incorrect results if C’s promotion rules are not followed.
• The promotion rules specify how types can be converted to other types without losing data.
• The type of each value in a mixed-type expression is automatically promoted to the “highest” type in the expression.
# Promotion Rules

<table>
<thead>
<tr>
<th>Data type</th>
<th><code>printf</code> conversion specification</th>
<th><code>scanf</code> conversion specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>long double</td>
<td><code>%Lf</code></td>
<td><code>%Lf</code></td>
</tr>
<tr>
<td>double</td>
<td><code>%f</code></td>
<td><code>%lf</code></td>
</tr>
<tr>
<td>float</td>
<td><code>%f</code></td>
<td><code>%f</code></td>
</tr>
<tr>
<td>unsigned long int</td>
<td><code>%lu</code></td>
<td><code>%lu</code></td>
</tr>
<tr>
<td>long int</td>
<td><code>%ld</code></td>
<td><code>%ld</code></td>
</tr>
<tr>
<td>unsigned int</td>
<td><code>%u</code></td>
<td><code>%u</code></td>
</tr>
<tr>
<td>int</td>
<td><code>%d</code></td>
<td><code>%d</code></td>
</tr>
<tr>
<td>unsigned short</td>
<td><code>%hu</code></td>
<td><code>%hu</code></td>
</tr>
<tr>
<td>short</td>
<td><code>%hd</code></td>
<td><code>%hd</code></td>
</tr>
<tr>
<td>char</td>
<td><code>%c</code></td>
<td><code>%c</code></td>
</tr>
</tbody>
</table>
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
Mathematical Library Functions

• Mathematical 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
Mathematical Library Functions

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pow(x, y)</code></td>
<td>$x$ raised to power $y$ ($x^y$)</td>
<td><code>pow(2, 7)</code> is 128.0, <code>pow(9, 0.5)</code> is 3.0</td>
</tr>
<tr>
<td><code>fmod(x, y)</code></td>
<td>remainder of $x/y$ as a floating-point number</td>
<td><code>fmod(13.657, 2.333)</code> is 1.992</td>
</tr>
<tr>
<td><code>sin(x)</code></td>
<td>trigonometric sine of $x$ ($x$ in radians)</td>
<td><code>sin(0.0)</code> is 0.0</td>
</tr>
<tr>
<td><code>cos(x)</code></td>
<td>trigonometric cosine of $x$ ($x$ in radians)</td>
<td><code>cos(0.0)</code> is 1.0</td>
</tr>
<tr>
<td><code>tan(x)</code></td>
<td>trigonometric tangent of $x$ ($x$ in radians)</td>
<td><code>tan(0.0)</code> is 0.0</td>
</tr>
</tbody>
</table>
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.
/* Fig. 5.12: fig05_12.c */
A scoping example */
#include <stdio.h>

void useLocal( void ); /* function prototype */
void useStaticLocal( void ); /* function prototype */
void useGlobal( void ); /* function prototype */

int x = 1; /* global variable */

/* function main begins program execution */
int main( void )
{
    int x = 5; /* local variable to main */

    printf("local x in outer scope of main is %d\n", x);

    { /* start new scope */
        int x = 7; /* local variable to new scope */

        printf("local x in inner scope of main is %d\n", x);
    } /* end new scope */
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     useLocal(); /* useGlobal uses global x */
29     useStaticLocal(); /* static local x retains its prior value */
30     useGlobal(); /* global x also retains its value */
31
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.)
/* useStaticLocal initializes static local variable x only the first time
the function is called; value of x is saved between calls to this
function */

void useStaticLocal( void )
{
    /* initialized only first time useStaticLocal is called */
    static int x = 50;

    printf("local static x is %d on entering useStaticLocal\n", x);
    x++;
    printf("local static x is %d on exiting useStaticLocal\n", x);
}

/* function useGlobal modifies global variable x during each call */

void useGlobal( void )
{
    printf("global x is %d on entering useGlobal\n", x);
    x *= 10;
    printf("global x is %d on exiting useGlobal\n", x);
}

/* 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
Sample: Square function

```c
#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

```c
#include <stdio.h>

int toplam(int, int);
int cikar(int, int);
int carp(int, int);
float bol(int, int);

void main()
{
    int secim,s1,s2;
    while(1)
    {
        printf("1-Topla\n2-Cikar\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", cikar(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 topla(int a, int b)
{
    return a+b;
}

int cikar(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

/* Fig. 6.13: fig06_13.c
   Passing arrays and individual array elements to functions */
#include <stdio.h>
#define SIZE 5

/* function prototypes */
void modifyArray( int b[], int size );
void modifyElement( int e );

/* function main begins program execution */
int main( void )
{
    int a[ SIZE ] = { 0, 1, 2, 3, 4 }; /* initialize a */
    int i; /* counter */

    printf( "Effects of passing entire array by reference:
    "
            "values of the original array are:
            ");

    /* output original array */
    for ( i = 0; i < SIZE; i++ ) {
        printf( "%3d", a[ i ] );
    } /* end for */
Passing Arrays to Functions

```c
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 */
```
Passing Arrays to Functions

```c
/* in function modifyArray, "b" points to the original array "a" in memory */

void modifyArray( int b[], int size )
{
    int j; /* counter */

    /* multiply each array element by 2 */
    for ( j = 0; j < size; j++ ) {
        b[ j ] *= 2;
    }/* end for */
} /* end function modifyArray */

/* in function modifyElement, "e" is a local copy of array element a[ 3 ] passed from main */

void modifyElement( int e )
{
    /* multiply parameter by 2 */
    printf( "Value in modifyElement is %d\n", e *= 2 );
} /* 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 Subscripted 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 Subscripted Arrays to Functions

/* Fig. 6.21: fig06_21.c */

#include <stdio.h>

void printArray( const int a[][3] ); /* function prototype */

/* function main begins program execution */
int main( void )
{
    /* initialize array1, array2, array3 */
    int array1[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
    int array2[2][3] = { 1, 2, 3, 4, 5 };
    int array3[2][3] = { { 1, 2 }, { 4 } };

    printf( "Values in array1 by row are:\n" );
    printArray( array1 );

    printf( "Values in array2 by row are:\n" );
    printArray( array2 );

    printf( "Values in array3 by row are:\n" );
    printArray( array3 );
    return 0; /* indicates successful termination */
} /* end main */
Passing Multiple Subscripted Arrays to Functions

```c
/* function to output array with two rows and three columns */
void printArray( const int a[][ 3 ] )
{
    int i; /* row counter */
    int j; /* column counter */

    /* loop through rows */
    for ( i = 0; i <= 1; i++ ) {

        /* output column values */
        for ( j = 0; j <= 2; j++ ) {
            printf( "%d", a[ i ][ j ] );
        } /* end inner for */
    } /* end outer for */

    printf( "\n" ); /* start new line of output */
} /* 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
References


► Paul J. Deitel, “C How to Program”, Harvey Deitel.

► Bayram AKGÜL, C Programlama Ders notları