# Topic 4. C Functions

COMP ENG 2SH4

Principles of Programming

McMaster University, 2015

Instructor: Sorina Dumitrescu

# Textbook Required Reading

- Chapter 5 “C Functions”
- Specifically, sections: 1-6, 8,9, 12, 13
- Sections 7, 14-16 will be covered later when we study recursion.
- The remaining sections are optional.

# Motivation for Using Functions

- Functions are modules in a program.
- Software reusability → reduces development cost, improves reliability.
  - C Standard library functions.
  - Functions developed by the same or other programmers.
- Modularity → easier program design and maintenance.
- Avoiding repeating code.

# Functions

- There are three important notions related to a function:
- **function call** - when the function is actually used in the program
- **function definition** - contains the code to be executed when the function is used
- **function prototype** - the function declaration; specifies function name, types of inputs and type of output

# Example

```c
#include <stdio.h>
/* function prototype */
double my_mean_square( double x, double y);

int main(void)
{   double a = 10.0, b = 20.0, c;
    c = my_mean_square(a,b); /* function call */
    printf( "c=%f\n", c ); return 0;
}
/* function definition */
double my_mean_square( double x, double y) {
    double z;
    x = x*x;  y = y*y;
    z = ( x + y ) / 2.0;
    return z; }     /*end of function */
```

# Function Definition

```
return-value-type function-name ( parameter-list )
{
    optional-variable-declaration
    optional-statement-list
}
```

- **return-value-type**:
  - the data type of the result returned to the caller.
  - ***void*** -- if the function does not return a value.
  - a function can return **at most** one value.
- **function-name**
  - any valid identifier
- **parameter-list**:
  - A comma separated list of parameters expected by the function
  - A parameter is a variable representing an input (information from the caller)
  - a type must be specified for each parameter
  - ***void*** -- if no parameters.
- **Function definition must be outside any other function**.

# Function Prototype

- It is the function declaration. Like variables, functions have to be declared before being used.
- `return-value-type function-name ( parameter-list );`
- The names of parameters may miss in the prototype.
- Examples:
  - `int max( int x, int y);`
  - `int max( int , int );`
- Function prototype must match the function definition:
  - Same data type of return value;
  - Same number of parameters and the same types.
- The prototype is used by compiler to validate function calls:
  - Number of arguments in the function call has to be the same as in the function prototype.

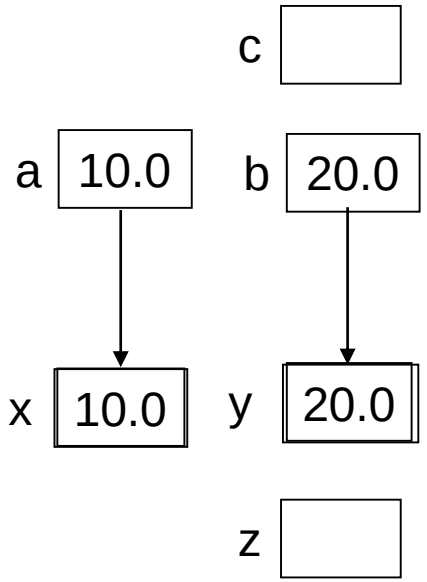
# Function Prototype - Coercion of Arguments

- Function prototype also cause “coercion of arguments”
  - if an argument value is not identical to type of the corresponding parameter, then it is converted to the appropriate type, in general

# Function Call

- When the function is used in the program.
- A function call is an expression.
- `function_name (comma separated list of arguments)`
  - Arguments are expressions.
- Arguments are passed to functions by value (pass by value):
  - Each argument is evaluated. Its value is assigned to the corresponding function parameter.
  - Type conversions occur as specified by prototype.
- Examples: c = my_mean_square(a, b);
- printf("Hello");

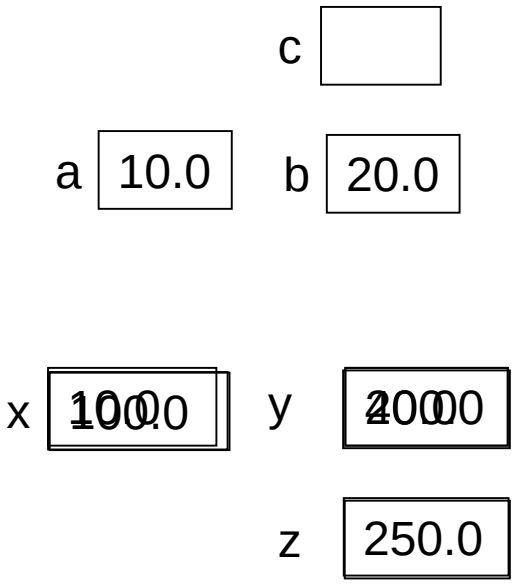
```
int main( void )
{     double a = 10, b = 20, c;
      c = my_mean_square( a, b);
      printf( "c=%f\n", c ); return 0;
}
double my_mean_square( double x, double y){
      double z;
      x = x*x;  y = y*y; z= ( x + y ) / 2.0;
      return z; }
```




**Executing the function call: my_mean_square( a, b)**

- Control is passed to function **my_mean_square()**.
- Memory is allocated for its **local variables (x,y,z).**
- Values of arguments are passed to the parameters.
  - x gets the value of a.
  - y gets the value of b.
- The code in the function body is executed.

```
int main( void )
{     double a = 10, b = 20, c;
      c = my_mean_square( a, b);
      printf( "c=%f\n", c ); return 0;
}
double my_mean_square( double x, double y){
      double z;
      x = x*x;  y = y*y; z= ( x + y ) / 2.0;
      return z; }
```




## Executing the function call: my_mean_square( a, b)

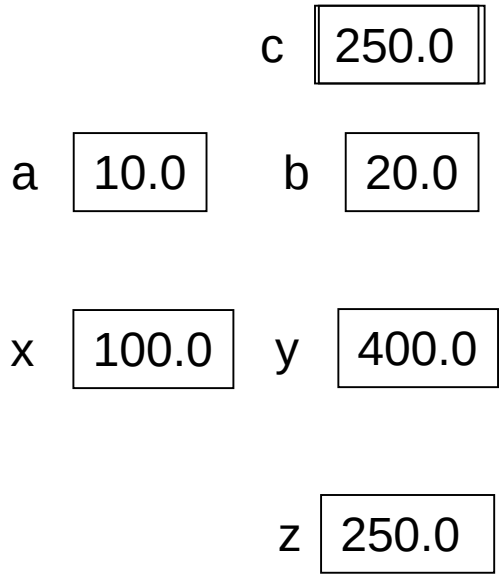
- The code in the function body is executed.
  - The values of **x** and **y** are modified.
  - **In the calling function the values of a and b are not modified.**

Execution ends when **return** or ending brace } of function definition, are reached.

**Note:** **If return type is void, then the return statement may be omitted.**

```
int main( void )
{    double a = 10, b = 20, c;
     c = my_mean_square( a, b);
     printf( "c=%f\n", c );
}

double my_mean_square( double x, double y) {
     double z;
     x = x*x;  y = y*y; z= ( x + y ) / 2.0;
     return z;
    }
```




After **my_mean_square()** finishes executing

- The local variables of **my_mean_square()** are destroyed (the memory is deallocated).
- Value of z is passed to main.
- Control is passed back to main.

# Notes

- "main()" function is the first called function.
- Control flow returns back to "main()" function.
- "***return***" statement: return back to the calling function.
- A non-*void* return type function must return one and only one value back to the calling function.

# Function Prototype

```
#include <stdio.h>

int max (int x, int y)
{
   printf (“x=%d, y=%d\n", x, y );
   return x>y?x:y;
}
/* invoking function max with
   floating point arguments is not
   a syntax error, but leads to
   incorrect results  */
int main( void )
{
   float a,b;
   scanf( "%f%f", &a, &b );
   printf ("a=%f, b=%f\n", a, b );
   printf ("max=%d\n", max (a,b));
   return 0;
}
```

- If a function definition appears in a file before any call to that function, the prototype may be omitted.
- Argument values of arithmetic types that do not correspond exactly to the types in the prototype are converted to the correct type.

```
16  int max (int x, int y)
17  {
18    printf ("x=%d, y=%d\n", x, y );
19    return x>y?x:y;
20  }
21  int main(int argc, int **argv)
22  {
23    float a,b;
24    printf("Please input two real numbers:\n");
25    scanf( "%f%f", &a, &b );
26    printf ("a=%f, b=%f\n", a, b );
27    printf ("max=%d\n", max (a,b));
28    return 0;
29  }
```

test

```
Please input two real numbers:
1.2    4.56
a=1.200000, b=4.560000
x=1, y=4
max=4
Press [Enter] to close the terminal ...
```

# Multiple Source Files

- Functions defined in different files may call each other.
- To call function *B* from one function *A, **include a prototype of B*** in the file where A is defined.
- To create a multiple source file, create
  - **Source files** (.c): contain function definitions.
  - **Header files** (.h): contain function prototypes, definitions of various data types, symbolic constants.
  - "**Include**" necessary header files in the source files.
    - #include <stdio.h>
    - #include "max.h" – header file located in the current directory.
  - **Compile** source files together.

```c
/* my math library header file: my_math.h*/
double my_add(double x, double y);
double my_sub(double x, double y);
```

```c
/* my math library source file: my_math.c */
double my_add(double x, double y)
{  return x + y; }
double my_sub(double x, double y)
{  return x - y; }
```

```c
/* testing program: test.c */
#include "my_math.h“
#include <stdio.h>
int main( void )
{  double a = 100, b = 200;
   printf(“%f\n”, my_sub(a, b));
   return 0; }
```

# C Standard Library Functions

- To use C standard library functions include the corresponding header file:
- **#include <math.h>** for math library functions
- **#include <string.h>** for string processing functions
- **#include <stdio.h>** for I/O functions
- **#include <stdlib.h>** for utility functions:
  - Conversions of numbers to text and vice versa
  - Dynamic memory allocation
  - Sorting , searching
  - Random number generation
- **#include <ctype.h>** for character handling functions
- **#include <time.h>** for manipulating time and date

# Some Math Library Functions

- For the following functions the return type and the type of the parameters is **double**
- sqrt( x ) – square root of x
- cbrt( x ) – cube root of x (C99 and C11 only)
- exp( x) -- $e^x$
- log( x ) -- $\log_e x$
- log10( x ) – $\log_{10} x$
- fabs( x ) – absolute value of x
- pow( x, y ) – x raised to power y
- sin( x ), cos( x ), tan( x )
- ceil( x ) - rounds x to the smallest integer not less than x
- floor( x ) - rounds x to the largest integer not greater than x

# Local Variable vs. Global Variable

- Function's parameters and the variables defined inside a function are **local** variables – they cannot be referenced outside the function.
- Variables defined outside any functions are **global** variables – they can be referenced anywhere in the program with proper declaration.
- Use of global variables is generally not recommended.
- **We will not use global variables in 2SH4.**

```
float pi = 3.14;
int main( void )
{
   int m = 0;
   ...
}
```

- **m is local**
- **pi is global**

# Local vs. Global Variables

```c
/* Ex1: alpha is global*/
#include <stdio.h>
double square( void );
double  alpha=1.5;

int main( void ){
     double a;
     a = square( );
     printf("a=%f\n", a);  }

double square( void ){
      return alpha * alpha;}
```

```c
/*Ex2: alpha is local to main;
square attempts to use it →
Compiler error*/
#include <stdio.h>
double square( void );

int main( void ){
double alpha=1.5;
double a;
     a = square( );
     printf("a=%f\n", a);  }

double square( void ){
      return alpha * alpha;}
```

# Scope Rules

- **Scope** of an identifier: the portion of the program where the identifier can be referenced.
- **File scope**
  - global variables, function names – can be referenced anywhere in the file after their definition or declaration.
- **Block scope**
  - variables defined inside a block – can be referenced only inside that block or in inner blocks. Local variables have block scope.
- **Function scope**
  - Labels (like case labels used in switch statement)
- **Function prototype scope**
  - identifiers in the parameter list of the prototype.

# Automatic Storage Duration

- **Storage duration** of an identifier: the period in which it exists in memory.
- A function's local variables have **automatic storage duration** by default:
  - created when the block in which they are defined is entered;
  - exist while the block is active;
  - are destroyed when the block is exited.

# Static Storage Duration

- Identifiers with static storage duration exist from the beginning to the end of program execution.
- Functions and global variable have static storage duration by default.
- A function's **local variable can acquire static storage duration** by declaration with specifier static:
  - static *char* letter;
- All **numeric static variables are initialized to 0** if they are not explicitly initialized by the programmer.
- Static variables **exist all the time in memory**, but they cannot necessarily be accessed from allover.

# Automatic vs. Static Local Variables

```
int function( int a1 )
{
   int b = 0;
   static int c;
   b++;
   c++;
   return ( a1 + b + c );
}
int main(void) /*main function*/
{
   int a = 2, i;
   for( i=0; i<3; i++ )
      printf( "%d ", function(a) );
   return 0;
}
```

```
4 5 6
```

| | beginnig of the call | | end of the call | | |
|---|---|---|---|---|---|
| | b | c | b | c | return |
| 1st call | 0 | 0 | 1 | 1 | 4 |
| 2nd call | 0 | 1 | 1 | 2 | 5 |
| 3rd call | 0 | 2 | 1 | 3 | 6 |

**Static local variables retain their value from one call to another.**

# Optional: static and extern Declaration for Global Variables and Functions

- Global variable **g** defined in file1 can be referenced in a function in file2, if **g** is declared in file2 with specifier **extern** – **external linkage.**
- A function defined in file1 can be referenced in file2 if its prototype is included in file2.
- A global variable **g** defined in file1 with specifier **static**, cannot be referenced outside file1 – **internal linkage.**
- A function declared or defined with **static** cannot be called from another file.
- **static** should be applied to the prototype or to the function definition – the one which appears first.