

CPE102 Programming II

Week 1

*Introduction, Scope Rules,
and Storage Classes*



Dr. Nehad Ramaha,
Computer Engineering Department
Karabük Universities

Topics During the Semester

- ▶ Recursive Functions
- ▶ Pointers (Call by Value, Call by Reference, Dynamic Memory Allocation)
- ▶ Struct, Enum and Typedef Definitions
- ▶ Singly Linear Linked Lists
- ▶ Sort and Search Algorithms
- ▶ String and Mathematical Functions
- ▶ Sequential and Random Access Files
- ▶ Bitwise Operators
- ▶ Basic Graphics Operations

Program Development Environments



- ▶ Download Dev-C++ from https://sourceforge.net/projects/dev-cpp/files/Binaries/Dev-C%2B%2B%204.9.9.2/devcpp-4.9.9.2_setup.exe/download

- ▶ Or download the Code::Blocks 17.12 installer from <https://www.codeblocks.org/downloads/binaries/>

If you know you don't have MinGW installed, download the package which has MinGW bundled.



- ▶ Download eclipse Neon from <http://www.eclipse.org/downloads/>

Choose Eclipse IDE for C/C++ Developers and install. You should download and install MinGW GCC

<http://www.mingw.org/>

- ▶ <https://visualstudio.microsoft.com/tr/vs/features/cplusplus/>



C Programming and Functions

- ▶ Functions break large computing tasks into smaller ones.
- ▶ Taking a problem and breaking into small, manageable pieces is critical to writing large programs.
- ▶ Functions return values to where it's invoked.
`Type function_name(parameter list) {declarations statements}`
- ▶ The parameter list is a comma-separated list of declarations.

Functions

```
void nothing(void) { }      /* this function does nothing */  
  
double twice(double x)  
{  
    return (2.0 * x);  
}  
  
int all_add(int a, int b)  
{  
    int    c;  
    .....  
    return (a + b + c);  
}
```

Datatype of data returned,
any C datatype.

"void" if nothing is returned.

Function name

Parameters passed to
function, any C datatype.

```
int myMultiplyFunction(int x, int y){  
    int result;  
    result = x * y;  
    return result;  
}
```

Return statement,
datatype matches
declaration.

Curly braces required.

Function Return Statement

- ▶ `return;` // `return ++a;` // `return (a*b)`
- ▶ When a return statement is encountered, execution of the function is terminated and control is passed back to calling environment.
- ▶ If the return statement contains an expression, then the value of the expressions is passed to the calling environment as well.


Function Return Statement

```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ..
    result = multiply(i, j);
    ... ..
}

int multiply(int a, int b)
{
    ... ..
    return a*b;
}
```



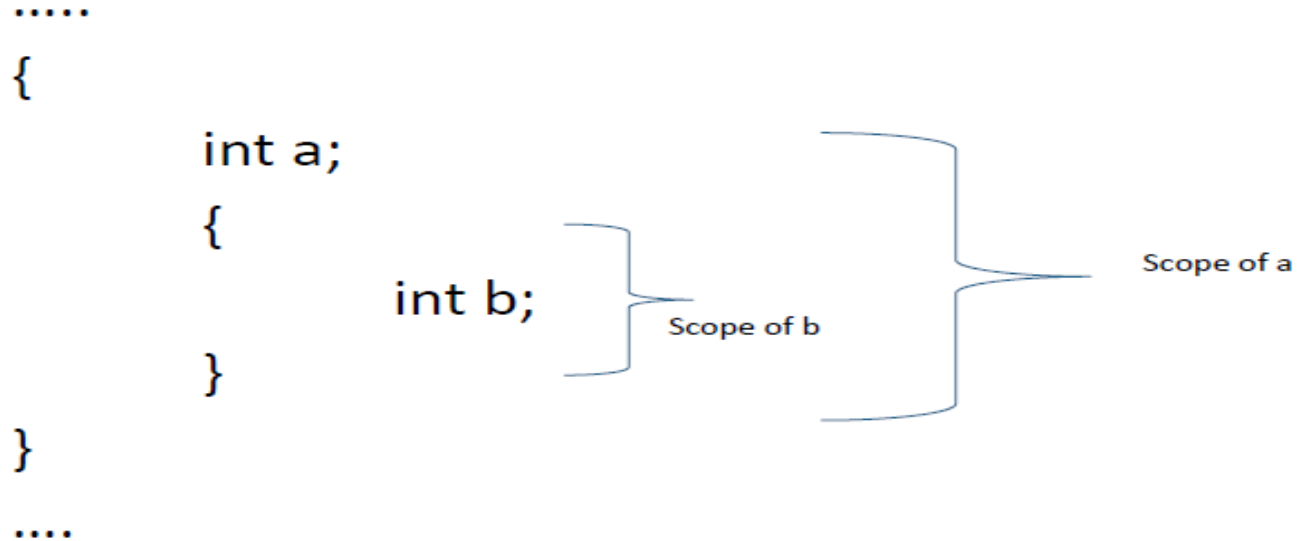
The value returned by the function must be stored in a variable.

Object Scope Area

- ▶ **Scope:** Program range for recognizing an object
- ▶ The object scope relates to the place where it is defined in the program.
 - Block Scope: Recognition in only one block.
 - Function Scope: Recognition of only one function.
 - File Scope: Recognition within the entire file.
- ▶ Variables are examined in 3 sections according to the scope:
 - Local variables.
 - Global variables.
 - Parameter variables.

Local Variables

- ▶ Local variables follow the block scope rule. They are valid only in the block where they are defined.



Local Variables

- ▶ Variables with the same name can be defined with different scope.
- ▶ The compiler maintains these variables at different addresses (in the memory).
- ▶ The variables with the same name defined in the inner blocks mask the variables with the same name defined in the outer block.
- ▶ Two variables with the same name cannot be defined in the same block.

```
int main()
{
    int a = 10;
    printf("a = %d\n", a);
    {
        int a = 20;
        printf("a = %d\n", a);
    }
    printf("a = %d\n", a);
    return 0;
}
```



Output:

a=10

a=20

a=10

Global Variables

- ▶ These are the variables defined outside all blocks.
- ▶ Global variables follow the file scope.
- ▶ The global variable can be initialized.
- ▶ Only the local variable can be accessed in the block where the global and local variable with the same name is recognizable.

```
#include <stdio.h>
int a;

void fonk1(void)
{
    a = 20;
}

int main()
{
    a = 10;
    printf("a = %d\n", a);
    fonk1();
    printf("a = %d\n", a);

    return 0;
}
```

```
#include <stdio.h>
int a=10;

void fonk1(void)
{
    a = 40;
    printf("a = %d\n", a);
}

int main()
{
    int a;
    a = 30;
    printf("a = %d\n", a);
    fonk1();
    printf("a = %d\n", a);

    return 0;
}
```

Parameter Variables

- ▶ These are function parameters.
- ▶ Adheres to the function scope.
- ▶ Only the parameter is valid in the function they are.

```
#include <stdio.h>
int a=10;

void fonk1(int a)
{
    a = 40;
    printf("a = %d\n", a);
}

int main()
{
    printf("a = %d\n", a);
    fonk1(a);
    printf("a = %d\n", a);

    return 0;
}
```

Object Lifecycle

- ▶ **Object Lifecycle:** Defines the time interval in which objects operate.
- ▶ Objects are divided into two parts as static and dynamic life.
- ▶ **Static life object:**
 - They operate until the end of the program.
 - They are stored in the data segment region.
1-Global variables 2-Strings 3-Static local variables
- ▶ **Dynamic life object:**
 - They operate in a certain part of the program within a certain time period and disappear.
 - 1-Local variables 2-Parameter variables 3-Objects created with dynamic memory functions
 - Local variables and parameter variables are stored in the Stack segment region.

Storage Classes and Type Specifiers

- ▶ Local variables are defined when the block they are defined is run, and they disappear when the block ends.
- ▶ In C, the secondary properties of objects are identified by determinants.
- ▶ Determinants: 1–Locator 2–Species
- ▶ **Specifiers**: 1–Storage classes specifier 2–Type specifier
- ▶ **4 storage classes specifiers**:
 - 1–auto 2–register 3–static 4–extern
- ▶ **2 type specifiers**:
 - 1–const 2–volatile
- ▶ General variable definition format:
[storage classes specifiers] [type specifier] [type] object;
- ▶ It can be in any order.
 - auto const int a = 10; (recommended)
 - const auto int a = 10;
 - int const auto a=10;

Storage Classes

- ▶ Every variable and function in C has two attributes. Type and storage class.
- ▶ **auto**: (allocated and de-allocated automatically when program flow enters and leaves the variable's scope)
 - Variables declared within function are automatic by default.
 - These variables can be used in scope of the function.
auto double x, y;
 - They're stored in the **Stack**.
 - Global variables and parameter variables cannot take auto property.

Storage Classes

- ▶ **extern:** (is used to declare a global variable that is a variable without any memory assigned to it. It is used to declare variables and functions in header files)
- ▶ One method of transmitting information across blocks and functions is to use external variables.
- ▶ When a variable is declared outside a function, storage is permanently assigned to it, and its storage class is extern.
- ▶ The C compiler automatically accepts the function defined in another module as extern.
- ▶ Extern use for functions is unnecessary.
- ▶ If the variable defined by extern is not given the initial value, the compiler does not allocate space in memory.

Examples: auto & extern

```
#include <stdio.h>
extern int a = 1, b = 2;
c = 3;
int f(void);
int main(void) {
    printf("%3d\n", f());
    printf("%3d%3d%3d\n", a, b, c);

    return 0;
}
int f(void) {
    auto int b, c;
    a = b = c = 4;
    return (a + b + c);
}
```

Output

Examples: auto & extern

example1.c

```
#include <stdio.h>

int a = 1, b = 2;
c = 3;
int f(void);
int main(void) {

    printf("%3d\n", f());
    printf("%3d%3d%3d\n", a, b, c);

    return 0;
}
```

file2.c

```
int f(void) {
    extern a;
    int b, c;
    b = c = a;
    return (a + b + c);
}
```

Storage Classes

- ▶ **register:** Storage class register tells the compiler that the association variables **should be stored in high-speed memory registers**.
- ▶ Specifies that the variable is kept in the registers of the CPU, not in memory.
- ▶ Keeping variables in the register allows the program to accelerate.
- ▶ Access to memory is slower than access to registers. Because a certain machine time is required to access the memories.
- ▶ Registers are limited.

Example: register

```
#include <time.h>
#include <stdio.h>

int main () {
    clock_t start_t, end_t;
    double total_t;

    // register variable will make the loop to execute faster.
    register int i;
    start_t = clock();
    printf("Starting of the program, start_t = %ld\n", start_t);
    printf("Going to scan a big loop, start_t = %ld\n", start_t);
    for(i=0; i< 100000000; i++);
    end_t = clock();
    printf("End of the big loop, end_t = %ld\n", end_t);

    total_t = (double) (end_t - start_t) / CLOCKS_PER_SEC;
    printf("Total time taken by CPU: %f\n", total_t );
    printf("Exiting of the program...\n");

    return(0);
}
```

Storage Classes

- ▶ **static:** It can be used with both variables and functions. An ordinary variable is limited to the scope in which it is defined, while the scope of the static variable is throughout the program.
- ▶ When to use static declarations:
 - 1 – To allow a local variable to retain its previous value when the block is reentered.
 - 2 – with external declarations to restrict the scope of the variable.
- ▶ Static local and global variables are kept in the **data segment region**.
- ▶ If we do not assign any value to the static variable, then the default value will be 0.

static local variable example

```
withoutStatic.c x
#include <stdio.h>
int main()
{
    printf("%d", func());
    printf("\n%d", func());
    return 0;
}
int func()
{
    int count=0; // variable initialization
    count++; // incrementing counter variable

    return count;
}
```

Output

1
1

```
withStaticDeclaration.c x
#include <stdio.h>
int main()
{
    printf("%d", func());
    printf("\n%d", func());
    return 0;
}
int func()
{
    static int count=0; // variable initialization
    count++; // incrementing counter variable

    return count;
}
```

Output

1
2

Static function

- ▶ **Non-static functions are global by default** means that the function can be accessed outside the file also, but the **static function** can be accessed within a file only.

```
static void func()  
{  
    printf("Hello world!!");  
}
```

Static variables

▶ Differences b/w static and global variable

- **Global variables** are the variables that are **declared outside the function**. These global variables exist at the beginning of the program, **and its scope remains till the end of the program**. It **can be accessed outside** the program also.
- Static variables are **limited to the source file** in which they are defined, i.e., they are **not accessible by the other** source files.

▶ Differences b/w static local and static global variable

- Static global variable: If the variable declared with a static keyword outside the function, then it is known as a static global variable. **It is accessible throughout the program.**
- Static local variable: The variable with a static keyword is declared **inside a function** is known as a static local variable. **The scope** of the static local variable will be the **same as the automatic local variables, but its memory will be available throughout the program execution**. When the function modifies the value of the static local variable during one function call, then it will remain the same even during the next function call.

Storage Classes

Storage Class	Declaration	Storage	Default Initial Value	Scope	Lifetime
auto	Inside a function/block	Stack	Garbage	Within the function/block	Within the function/block
register	Inside a function/block	CPU Registers	Garbage	Within the function/block	Within the function/block
extern	Outside all functions	Data Segment	Zero	Entire the file and other files where the variable is declared as extern	program runtime
Static (local)	Inside a function/block	Data Segment	Zero	Within the function/block	program runtime
Static (global)	Outside all functions	Data Segment	Zero	Global (inside the file)	program runtime

Scoping Example (p217/the book)

ScopingExample.c X

```
1  /* A scoping example */
2  #include <stdio.h>
3
4  void useLocal( void );      /* function prototype */
5  void useStaticLocal( void ); /* function prototype */
6  void useGlobal( void );    /* function prototype */
7
8  int x = 1; /* global variable */
9  /* function main begins program execution */
10 int main()
11 {
12     int x = 5; /* local variable to main */
13
14     printf("local x in outer scope of main is %d\n", x );
15
16     { /* start new scope */
17         int x = 7; /* local variable to new scope */
18
19         printf( "local x in inner scope of main is %d\n", x );
20     } /* end new scope */
21
22     printf( "local x in outer scope of main is %d\n", x );
23
24     useLocal(); /* useLocal has automatic local x */
25     useStaticLocal(); /* useStaticLocal has static local x */
26     useGlobal(); /* useGlobal uses global x */
27     useLocal(); /* useLocal reinitializes automatic local x */
28     useStaticLocal(); /* static local x retains its prior value */
29     useGlobal(); /* global x also retains its value */
30
31     printf( "\nlocal x in main is %d\n", x );
32
33     return 0; /* indicates successful termination */
34 } /* end main */
```

```
34
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
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 */
67
```

Scoping Example-output

```
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
```

Thanks 😊