CME 112- Programming Languages II

Week 1

Introduction, Scope Rules and Generating Random Numbers

Assist. Prof. Dr. Caner Özcan

You have two options at any given moment. You can either: Step forward into growth or Step back into safety. ~A.Maslow

Introduction

Course Web Site: www.canerozcan.net

Office Hours: **Wednesday 14:00-15:30**

Thursday 13:00-15:30

or appointment by email: canerozcan@karabuk.edu.tr

Textbooks:

“Algoritma Geliştirme ve Programlamaya Giriş”, Doç. Dr. Fahri Vatansever.

“C how to Program ”, Paul J. Deitel, Harvey Deitel.

“A book on C”, All Kelley, Ira Pohl

“A’dan Z’ye C Klavuzu’’, Kaan Aslan.

Introduction

- Work hard and practice!
- Grading
  - Midterm Exam + Homework + Lab. Exam: %40
  - Final Exam + Homework + Lab. Exam: : %60
- Study to learn, not for grade. Grade is achieved anyway.

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

How to Become a Good Programmer?

- Practice is habit.
- Practice is routine.
- It is obtained by practicing.
- Practice requires great devotion and dedication.
- Some skills acquired by practicing.
  - Shooting
  - Driving a car
  - Writing

Read Written Program Codes

- If you want to be a novelist, can you start writing novels without reading the best written novels?
- If you want to be a film screenwriter, can you make the best film without reading the best movie scripts?
- If you want to be a software engineer, how can you be a software engineer without reading program codes?
- If you read the written code, you can use the techniques you like. If you see the wrong situation, you may not do the wrong.

Read Written Program Codes

- Software codes have many features.

- Indents, description lines, naming, function structures, etc.

- Review the code written by experienced software engineers. In a short time you can start writing better than the codes you wrote yourself.

Before You Start Writing Code

- Complete the documentation before you start writing code.
- Design must be done.
- Specifications, design documents, constraints and assumptions, algorithms, flow diagrams.
- You must prepare for tomorrow with what you have learned today.

Follow Specified Standards

- Follow established standards and do not create yourself.
  - File naming conventions
  - Function and module naming convention
  - Variable naming conventions
  - Date, indentation, reviews
  - Readability guidelines
  - List of things to do and don'ts.

Review Codes

- Codes should be written for review.
  - Poor coding and non-compliance
  - Performance not considered
  - Date, indentation, explanations not suitable
  - Poor readability
  - Open files are not closed
  - Allocated memory is not freed
  - Too many global variables and too much coding
  - Poor exception handling
  - No modularity and repeated code.

Program Development Environments

Program Development Environments

Download Dev-C++ from http://www.bloodshed.net/dev/devcpp.html and install it. https://en.wikiversity.org/wiki/Installing_and_using_Dev-C%2B%2B

Download the Code::Blocks 17.12 installer. 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 is invoked.

```
type function_name(parameter list)
{declerations statements}
```

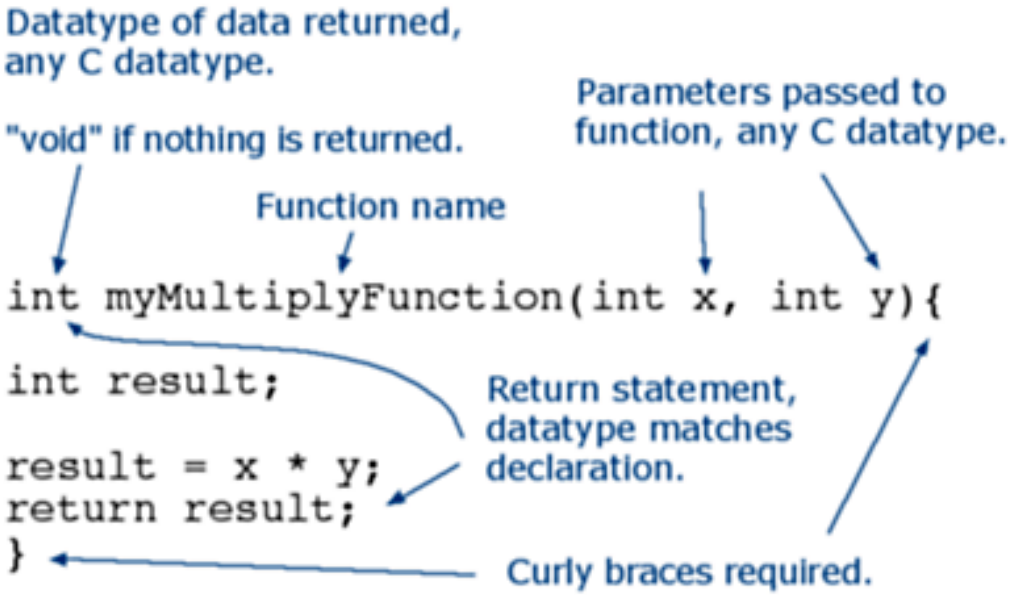
- The parameter list is a comma-seperated list of declarations.

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);
}
```

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);
}
```



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.

Even though a function returns a value, a program does not need to use it.

```
while (·····) {
    getchar();          /* get a char, but do nothing with it */
    c = getchar();      /* c will be processed */
    ·····
}
```

Functions


```
#include <stdio.h>

#define   N   7

long    power(int, int);
void    prn_heading(void);
void    prn_tbl_of_powers(int);

int main(void)
{
    prn_heading();
    prn_tbl_of_powers(N);
    return 0;
}
```

Functions


```
void prn_heading(void)
{
   printf("\n::::::  A TABLE OF POWERS  ::::::\n\n");
}

void prn_tbl_of_powers(int n)
{
   int   i, j;

   for (i = 1; i <= n; ++i) {
      for (j = 1; j <= n; ++j)
         if (j == 1)
            printf("%1d", power(i, j));
         else
            printf("%9ld", power(i, j));
      putchar('\n');
   }
}
```

Functions


```
long power(int m, int n)
{
    int     i;
    long    product = 1;

    for (i = 1; i <= n; ++i)
        product *= m;
    return product;
}
```

Here is the output of the program:

```
:::::  A TABLE OF POWERS  :::::

1         1         1          1          1          1          1
2         4         8         16         32         64        128
3         9        27         81        243        729       2187
.....
```

Object Scope Area

- Scope: Program range for recognizing an object
- The object scope relates to the place where it is defined in the program.

  1-Block Scope

  Recognition in only one block

  2-Function Scope

  Recognition of only one function

  3-File Scope

  Recognition within the entire file.
- Variables are examined in 3 sections according to the scope:

  Local variables
  Global variables
  Parameter variables

  Note: The fact that the two variables belong to the same group of fields (block, function, or file) does not necessarily mean that the fields of activity are exactly the same.

Local Variables

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

```
.....
{
        int a;
        {
                int b;
        }
}
....
```

Scope of b

Scope of a

Local Variables

- Variables with the same name can be defined with different scope.
- The compiler maintains these variables at different addresses.
- 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;
}
```

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.

```c
#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;
}
```

```c
#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.

```c
#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.

Object Lifecycle

- For example, local variables are defined when the block is run, and they disappear when the block ends.
- The working time is up to the duration of the block.
- The value of the static-life variable is 0, although it is not initialized.
- If the dynamic-life variables are not assigned the first value, then the current value is assigned to the variable in the memory.
- For example, if the global variable is not initialized, the value is 0. The value cannot be estimated if the local variable is not initialized.

Storage Classes and Type Specifiers

- For example, 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.
- Four storage classes are automatics, external, register, and static with corresponding.
- The object is created and destroyed in its own block.
  - **auto:** 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:** One methods 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);
}
```

Examples: auto & extern

- This use of extern is used to tell the compiler to ‘’look for it elsewhere’’ either in this file or in some other file.

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.

C code	Assembly
data3 = data1+data2	MOV reg, data1
	ADD reg, data2
	MOV data3, reg

- Access to memory is slower than access to registers. Because a certain machine time is required to access the memories.
- Registers are limited.

Storage Classes

- **static:** Local variables defined in functions.

  After the function ends, the variable value is stored.

  Only valid in the function they are defined.

  Static declarations have two important and distinct uses. One of them is to allow a local variable to retain its previous value when the block is reentered.

  The second and more subtle use of static is in connection with external declarations. It is use to restriction of the scope of the variable.

  Static local and global variables are kept in the data segment region.

Example: register


```
#include <stdio.h>
#include <time.h>
int a =1;
#define N 10000
int main(void) {
        clock_t start, end;
        double cpu_time_used;
        register double i;
        start = clock();
        for(i=0;i<N;i=i+0.0001);
        end = clock();
        cpu_time_used = ((double) (end - start)) /
        CLOCKS_PER_SEC;
        printf("Running time is %f",cpu_time_used);
                return 0;
}
```

* Running time is 0,163 second with register variable.
* Running time is 0,419 second without register variable.

Example: static


```
void f(void)
{
    static int   cnt = 0;

    ++cnt;
    if (cnt % 2 == 0)
        .....              /* do something */
    else
        .....              /* do something different */
}
```

Example: static


```
void f(void)
{
   .....      /* v is not available here */
}

static int     v;     /* static external variable */

void g(void)
{
   .....      /* v can be used here */
}
```

Example: static


```
static int   g(void);          /* function prototype */

void f(int a)                  /* function definition */
{                              /* g() is available here, */
    .....                      /* but not in other files */

}

static int g(void)             /* function definition */
{
    .....
}
```

Example


```
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
```

Example


```
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
```

Example


```
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 */
```

Example


```
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
```

Random Number Generator

rand function

- <stdlib.h> library is needed
- Returns a “random” number between 0 and RAND_MAX (at least 32767- max value for 16 bit integer)
- RAND_MAX is a symbolic constant defined in <stdlib.h>.
- Every number between 0 and RAND_MAX has equal probability of being chosen.
- The range of values produced by rand varies by what is needed in application

Random Number Generator

- Program simulating coin tossing might require only 1 for tails or 0 for heads.
- A dice rolling program that simulates six-sided die would requires random integers from 1 to 6.
- **Scaling:**
  - Values generated by rand is always between 0 and RAND_MAX

    0 ≤ rand ( ) ≤ RAND_MAX
  - Use remainder **%** operator with rand function for example to produce numbers between 0 and 5. It is called **scaling**

    rand( ) % 6

Random Number Generator

- The number 6 is called **scaling factor**.
- To **shift** the range just add 1 to the produced result.
  - **randNumber = 1 + rand ( ) % 6** produces numbers

    1 ≤ randNumber ≤ 6
- General rule:
  - **n = a + rand ( ) % b ;**

    a is shifting value (First number in desired range of consecutive integers)

    b is scaling factor (Equal to the width of desired range of consecutive integers)

Random Number Generator


```
1  /* Fig. 5.7: fig05_07.c
2     Shifted, scaled integers produced by 1 + rand() % 6 */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* function main begins program execution */
7  int main( void )
8  {
9     int i; /* counter */
10
11    /* loop 20 times */
12    for ( i = 1; i <= 20; i++ ) {
13
14       /* pick random number from 1 to 6 and output it */
15       printf( "%10d", 1 + ( rand() % 6 ) );
16
17       /* if counter is divisible by 5, begin new line of output */
18       if ( i % 5 == 0 ) {
19          printf( "\n" );
20       } /* end if */
21    } /* end for */
22
23    return 0; /* indicates successful termination */
24 } /* end main */
```

Random Number Generator

- Sample Program: Simulating 6000 rolls of a die
- Each integer from 1 to 6 should appear approximately with equal likelihood (1000 times)

```
1  /* Fig. 5.8: fig05_08.c
2     Roll a six-sided die 6000 times */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* function main begins program execution */
7  int main( void )
8  {
9     int frequency1 = 0; /* rolled 1 counter */
10    int frequency2 = 0; /* rolled 2 counter */
11    int frequency3 = 0; /* rolled 3 counter */
12    int frequency4 = 0; /* rolled 4 counter */
13    int frequency5 = 0; /* rolled 5 counter */
14    int frequency6 = 0; /* rolled 6 counter */
15
16    int roll; /* roll counter, value 1 to 6000 */
17    int face; /* represents one roll of the die, value 1 to 6 */
18
```

Random Number Generator


```
19      /* loop 6000 times and summarize results */
20      for ( roll = 1; roll <= 6000; roll++ ) {
21         face = 1 + rand() % 6; /* random number from 1 to 6 */
22
23         /* determine face value and increment appropriate counter */
24         switch ( face ) {
25
26            case 1: /* rolled 1 */
27               ++frequency1;
28               break;
29
30            case 2: /* rolled 2 */
31               ++frequency2;
32               break;
33
34            case 3: /* rolled 3 */
35               ++frequency3;
36               break;
37
38            case 4: /* rolled 4 */
39               ++frequency4;
40               break;
41
```

Random Number Generator


```
42            case 5: /* rolled 5 */
43               ++frequency5;
44               break;
45
46            case 6: /* rolled 6 */
47               ++frequency6;
48               break; /* optional */
49         } /* end switch */
50      } /* end for */
51
52      /* display results in tabular format */
53      printf( "%s%13s\n", "Face", "Frequency" );
54      printf( "    1%13d\n", frequency1 );
55      printf( "    2%13d\n", frequency2 );
56      printf( "    3%13d\n", frequency3 );
57      printf( "    4%13d\n", frequency4 );
58      printf( "    5%13d\n", frequency5 );
59      printf( "    6%13d\n", frequency6 );
60      return 0; /* indicates successful termination */
61   } /* end main */
```

```
Face    Frequency
   1         1003
   2         1017
   3          983
   4          994
   5         1004
   6          999
```

Random Number Generator

- Function **rand** actually generates pseudorandom numbers
- Calling **rand** repeatedly produces a sequence of numbers that appears to be random
- However, sequence repeats itself on each program execution
- To produce different sequence of integers on each program execution we use **srand** function
- **srand** takes an unsigned integer as an argument
- **srand** seeds function rand to produce different sequence of numbers on each execution of the program

Random Number Generator


```
1  /* Fig. 5.9: fig05_09.c
2     Randomizing die-rolling program */
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  /* function main begins program execution */
7  int main( void )
8  {
9     int i; /* counter */
10    unsigned seed; /* number used to seed random number generator */
11
12    printf( "Enter seed: " );
13    scanf( "%u", &seed ); /* note %u for unsigned */
14
15    srand( seed );  /* seed random number generator */
16
17    /* loop 10 times */
18    for ( i = 1; i <= 10; i++ ) {
19
20       /* pick a random number from 1 to 6 and output it */
21       printf( "%10d", 1 + ( rand() % 6 ) );
22
```

Random Number Generator


```
23         /* if counter is divisible by 5, begin a new line of output */
24         if ( i % 5 == 0 ) {
25            printf( "\n" );
26         } /* end if */
27      } /* end for */
28
29      return 0; /* indicates successful termination */
30   } /* end main */
```

```
Enter seed: 67
        6       1       4       6       2
        1       6       1       6       4
```

```
Enter seed: 867
        2       4       6       1       6
        1       1       3       6       2
```

```
Enter seed: 67
        6       1       4       6       2
        1       6       1       6       4
```

Random Number Generator

- To generate a random number each time without entering a seed value

```
srand ( time (NULL));
```

  - Reads system clock to obtain the value for seed automatically
  - Function time return the number of seconds that have passed since midnight on January 1970
  - The <time.h> library is used for the time function.

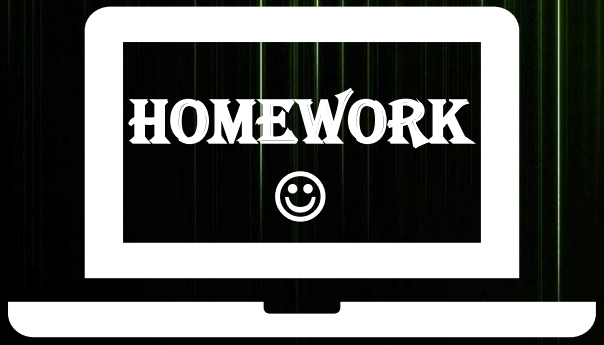
A Game of Chance: Craps

Rules:

- Roll two dice
- Sum of the spots on two upward faces is calculated
- 7 or 11 on first throw player wins
- 2, 3 or 12 on first throw player loses
- 4,5,6,8,9,10 becomes players point
- Player must roll his point before rolling 7 to win

#CRAPSGAME

KEEP
CALM
AND
CODE
ON



Next Week

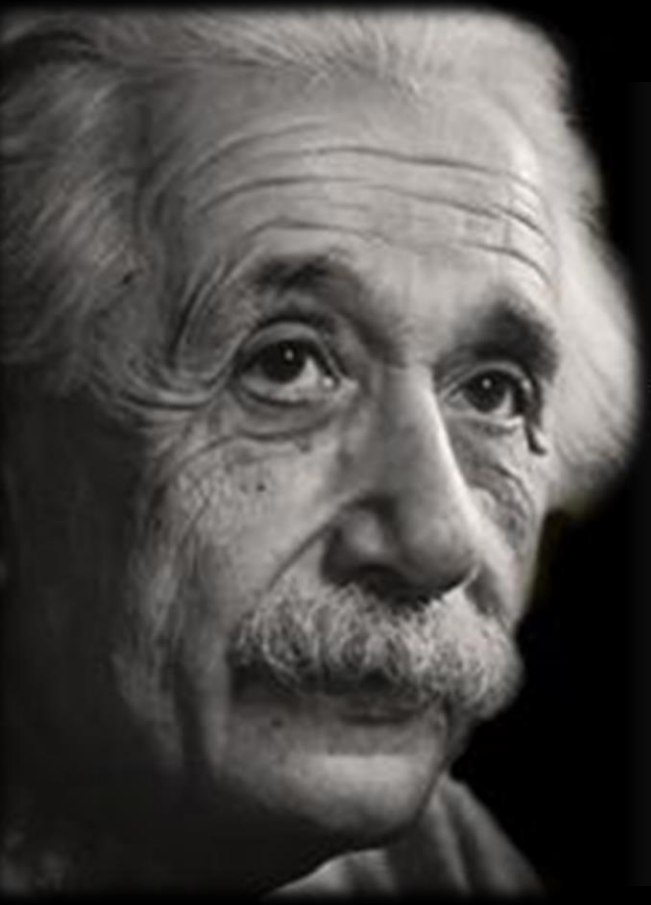
- Memory Layout of C Programs
- Recursive Functions

References

- Doç. Dr. Fahri Vatansever, “Algoritma Geliştirme ve Programlamaya Giriş”, Seçkin Yayıncılık, 12. Baskı, 2015.
- Kaan Aslan, “A’dan Z’ye C Klavuzu 8. Basım”, Pusula Yayıncılık, 2002.
- Paul J. Deitel, “C How to Program”, Harvey Deitel.
- “A book on C”, All Kelley, İra Pohl

Thanks for listening

CANER ÖZCAN
canerozcan@karabuk.edu.tr



I believe in intuition and inspiration. Imagination is more important than knowledge. For knowledge is limited, whereas imagination embraces the entire world, stimulating progress, giving birth to evolution. It is, strictly speaking, a real factor in scientific research.

— Albert Einstein —

Hayal gücü bilgiden daha önemlidir. Çünkü bilgi sınırlıyken, hayal gücü tüm dünyayı kapsar, ilerlemeyi teşvik eder, evrimi doğurur. Açıkçası, bilimsel araştırmada gerçek bir faktördür.