jason fedin c notları

15 ocak 2022

- bu dokümanda kullanılan kaynaklar:
    1. websiteleri:
        - dataflairs.com - C Tutorials
    2. kitaplar:
        - C Handbook - Flavio Copes
          eh işte. very broad narration about the hottest topics in C. can't really learn anything since it heavily lacks practice and use of real-code but overall good for having a general grasp of the matter.
        - c for dummies
          first time finishing a for-dummies-book. didn't really like it. i can't remember learning anything new, perhaps a one or two. doesn't recommend but this is perhaps i was not totaly novice in C. if you are, you may benefit from this book idk.
        - deitel, how to C program - harvey & paul deitel. 8th edition
    3. kurslar:
        - C Programming - Jason Fedin (24+ Hours)
          a great teacher. pretty obvious has a great knowledge of the language coupled with greater teaching skills. but pointers topic was a torture probably not because of the teacher but because of the content. also the examples he gave in functions was quite similar to the 42piscine projects. doesn't matter if you are 0 in C or not. a great way to start the language. highly recommend.
        -
    4. youtube:
        -
    5. muhtelif cheatsheets, PDFs, infographs, videos vs.

- C topics:
    1. intro
- some programming languages:

1. python - 2001 by Guido van Rossum. Python is a general-purpose, object-oriented, server-side interpreted, dynamic, multi-platform, high-level, open-source, non-compiled, scripting language. It uses the CPython Interpreter to compile the Python code to byte code.
2. java - 1995 by James Gosling. used to create applications in computers. original name was Oak. 360 derece turlar attığın siteler java ile yazılır.
3. C - 1972 by Dennis Ritchie. general purpose, imperative language. bad thing is C doesn't support OOP, that's why C++ invented. C biliyorsan bütün programlama dillerini temel düzey de olsa biliyorsundur. çoğunun babası.
4. C++ - 1983 by Bjarne Stroustrup. viewed by many as the best choice for large-scale projects. öğrenmesi çok zordur.
5. javascript - 1995 by Netscape. JS code is written into an HTML page. Front end web işlerinde kullanılır.
6. C# - 2000. windows uygulamaları, bankacılıkta vs. kullanılıyor. öğrenmesi zor.
7. ruby - 1995. dynamic, OO, general-purpose programming language. one of the best to start with. ruby is a mix of LISP, SmallTalk, Ada, Perl, Eiffel languages
8. php - server-side language designed for web development.
9. objective-C - ios ve apple OS X'in ana dili. sadece ios geliştirmek için öğreniliyor.

- some usages of C
    1. C is used almost all of electronic devices.
    2. It is used in the developement of Linux.
    3. It is the father of programming languages.
    4. C is a subset of C++. Difference is that C++ has OOP tools embedded which is not existant in C. by learning C you know much about C++ so it is easy to learn that too.
    5. almost all OS are written in C and/or C++
- C is probably the most widely known programming language which has evolved from an older language called B. C is a procedural programming language as well as a general-purpose programming language that was developed by Dennis Ritchie at AT&T's Bell laboratories in 1972. It is used as the reference language for computer science courses all over the world, and it's probably the language that people learn the most in school among with Python and Java. C is widely used in embedded devices, and it powers most of the Internet servers, which are built using Linux. The Linux kernel is built using C, and this also means that C powers the core of all Android devices. We can say that C code runs a good portion of the entire world. C is a compiled programming language, like Go, Java, Swift or Rust. Other popular programming languages like Python, Ruby or JavaScript are interpreted unlike C. The difference is consistent: a compiled language generates a binary file that can be directly executed and distributed. C is a statically typed language. This means that any variable has an associated type, and this type is known at compilation time. This is very different than how you work with variables in Python, JavaScript, PHP and other

interpreted languages. When you create a variable in C, you have to specify the type of a variable at the declaration.

1. it is also multi-platform
2. it uses imperative language - this is the most modern type of language
3. top-down reading by the computer
4. structured programming
5. because it is a very low level language it is fast

- After the introduction of the C language, previously developed programming languages like ALGOL, COBOL, B soon lost its significance and drastically shook their well-established niche in the domain of programming. C was developed by Dennis Ritchie in 1972. Ritchie aimed at improving B language as it was considered slow and lacked features such as byte addressability. The development of C language was followed by the origin of Unix, the first operating system implemented in a high-level language. The first Operating System developed using C was Unix.
- C'nin avantajları:

  1. Simple and efficient - The syntax style is easy to comprehend. We can use C to design applications that were previously designed by assembly language.

  2. Memory Management - It allows you to allocate memory at the runtime, that is, it supports the concept of dynamic memory allocation.

  3. Dynamic Memory Allocation- When you are not sure about the memory requirements in your program and want to specify it at the run time, that is, when you run your program, you can do it manually. 4. Pointers - C language provides a pointer that stores the memory address as its value. Pointers are useful in storing and accessing data from memory. We will study this in detail in our upcoming tutorials.

  5. Case Sensitive - It is pretty clear that lowercase and uppercase characters are treated differently in C. It means that if you write "program" and "Program", both of them would connote different meanings in C. The 'p' in "program" is in lowercase format whereas, the 'P' in Program is in uppercase format.

  6. Compiler Based - C is a compiler based language, that is, to execute a code we first need to compile it.

  7. Structure Oriented/Modular - C is a structured programming language. This means you can divide your code and task within a function to make it interactive. These functions also help in code reusability.

  8. Portable - It is easy to install and operate and the result file is a .exe file that is easy to execute on any computer without any framework.

  9. Compiles faster - C has a faster compiler that can compile 1000 lines of code in seconds and optimize the code to give speedy execution.

  10. User-defined functions - C has many header files that define a lot of functions, making it easier for you to code. You can also create your functions; these are called user-defined functions (UDFs).

11. C has a lower level of abstraction - C is a very clear and descriptive language. You can, in a way, directly see into the machine without any conceptual hiding and so learning C first makes the concepts very clear for you to proceed. Note: Abstraction means Data Hiding

however, some disadvantages of C:

1. it lacks OOP
2. it lacks exception handling
3. low level of abstraction
4. run-time checking

- C dili Tiobe endeksinde uzun zamanlar birinci sıradaki programlama dili olmuştur. Dolayısıyla sektörde iş imkanı çoktur.
- Fun fact: Sanskrit is regarded as the most suitable language for programming as each and every word in Sanskrit connotes a logical meaning to the compiler.
- Basic C Syntax:
  1. Header Files (#include <stdio.h>)
  2. Main Function (int main())
  3. Tokens in C
     - keywords: words like for, int, String etc.
     - identifiers/variables
     - constants: Constants are those values that cannot be changed during program execution.
     - punctuators: [ ] { } ( ) , ; : * = etc.
     - operators
  4. Comments in C
  5. Whitespaces in C
  6. Semicolons in C
- tıpkı Google gibi C'nin de versiyonları vardır (C89, C90, C99, C11 etc.) fakat bunlar daha sonra standartlaştırılmıştır. En sık kullanılan versiyon C89'dur (1989).
- C programs written on one system work on other systems, it is a portable language
- python, fortran, perl, pascal, basic vs. gibi diller C'de yazılmıştır.
- The C programming language is one of the most widely used programming languages and has huge importance in Computer Science. Because of its fundamental structure, it is being preferred by Google and Algorithm Development. Initially, it was developed for working on operating systems (i.e. UNIX OS) for minicomputers, but lately, it gained much importance in every field. C language is a general-purpose, portable, and easy-to-use programming language that makes it important for everyone. To provide better learning resources, IncludeHelp has several C programming tutorials from beginners to advanced algorithmic

problems with examples and explanations. In addition to learning C tutorials, we have provided articles for the best Job preparation, including Interview Question sets. What is C Language? C is a procedural computer programming language that has been used throughout the world for over four decades. Bell Labs have a significant contribution to the development of Electrical Engineering and Computer Science, and C language is one of those contributions. Dennis Ritchie, between 1972 and 1973, initially developed C. During the 1980s, C gradually started gaining popularity around the globe, and it has become one of the most widely used programming languages. Since 1989, C has been standardized by the ANSI (ANSI C) and by the ISO (International Organization for Standardization). Being a procedural language, C facilitates with structured programming and allows us to implement recursion and lexical variable scope. C has a definitive memory management system by providing three different ways to allocate memory for object: Static memory allocation Automatic memory allocation Dynamic memory allocation Moreover, C as a Programming language exhibits other characteristics listed below: It consists of a large number of bitwise, logical and arithmetic operators: +, +=, ++, &, ||, etc. C has a fixed number of keywords (i.e. small set), so it also provides us a free hand to use multiple terms throughout the code. It also includes a full set of control flow primitives i.e. if/else as conditional and, for, do-while, while as loop primitives. C has a definitive feature i.e. Functions: Functions in C permit run-time polymorphism. Values returned by a function can be ignored, when not needed. Functions can be defined with different scopes Applications of C Language C is not an outdated language and many world's leading companies are using C programming for their computational development. Most of the software is based on C, and it laid the foundation of other programming languages. Many algorithms are implemented in C and therefore, it also opens wide opportunities in research. Moreover, because of its fundamental features and its ability to be a foundation of computer science, It is still the most preferred programming language for programmers and back-end developers. Following are the key applications of C as a Programming Language: Software Development Operating Systems Graphical User Interface Development Gaming and Animation Hardware Manipulation Embedded Systems Robotics Competitive Programming Job at Google!!

- there are 4 fundamental tasks in the creation of any program:

    1. editing
    2. compiling
    3. linking
    4. executing

  editing: the process of creating and modifying your code.

- mingw + VSC kurulum:

    1. VSC indir
    2. C/C++ extension indir
    3. sourceforge'dan mingw64 indir
    4. env ⟶ system variables ⟶ path'e bin'i ekle ve bu kadar
    5. terminalde gcc -version ve g++ --version dediğinde hata vermiyosa C compiler is Okay

- CPU: Central Processing Unit. It does most of the computing. in a way brain of the computer where executions are being done.
- RAM: Random Access Memory. It is not to be confused with hard drive. RAM stores data while the program is running. Hard drives on the other hand stores them for good.
- Hard Drive: Stores data even while computer is turned off.
- higher-level programming languages make it easier to use and learn. They are opposite of Assembly. C is higher level language (wasn't it Low level?)
- A compiler's function is to convert your the code, that is high level language into machine language, that is 1s and 0s aka binary. C is a compiled language. To run the program we must first compile it. Any Linux or macOS computer already comes with a C compiler built-in. For Windows, you can use the Windows Subsystem for Linux (WSL). Compilers' main use is to find errors in your code and turn these source codes into machine codes. The compilation is a two-fold process

    1. preprocessing phase: during which your code may be modified or added to
    2. second phase: where the actual compilation that generates the object code

    the output from compiler is known as object code (stored in .obj or .o extensioned files). the standard command in C to compile your code is cc/gcc. After your code is translated into object code, it is ready for the 3rd phase called linking. Purpose of Linking phase is to get the program into a final for execution on PC.
- steps of C:

    1. editing
    2. compiling
    3. linking
    4. executing
- how to write code?

    1. define the program objectives
        - understand the requirements of the program
        - get a clear idea of what you want your program to accomplish
    2. design
        - decide how the program will meet the above requirements
        - what should the user interface be like?
        - how should the program be organized
    3. write the code
        - start implementation, translate and design the syntax in C
        - you need to use a text editor to create what is called a source code file
    4. compile

- translate the source code into machine code.

5. run the program
    - execute the code you wrote
6. test and debug
    - just because a program is running it doesn’t mean it works fully as intended
    - debuggin enables you to find bugs and helps you solve them.
7. maintain and modify the program
    - maintaining sometimes can be harder than writing the whole app/program

#intro

- C’de basit hello world yazdırma:

```c
#include <stdio.h>


int main()
{
    1. yol
    printf("merhaba dünya");


    2. yol
    char *a = "merhaba dünya";
    printf("%s", a);


    3. yol
    char a[] = "merhaba dünya";
    printf("%s", a);


    return 0;
}
```

- the basic structure for all C programs:

```c
#include <stdio.h>


int main() {}
```

  1. basically it is a function named main() and everything is executed between {} when ran
  2. there can and must be merely one main() function in all C programs

- my first C program:

```c
#include <stdio.h>


int main() {
    printf("my name is bugra");
    return 0;
}
```

1. we first import the stdio library (the name stands for standard input-output library) This library gives us access to input/output functions. stdio is the libraries that provides the printf() function among with other things.
2. This function is wrapped into a main() function. The main() function is the entry point of any C program.
3. every statement in C must end with ;

- comments are useful in reminding you of the use of the code you wrote 6 months ago. they are ignored by the IDE. there two sorts of comments (/* abc */ and // abc) in C:

```
// this is a single line comment


/*
this is
multiple line
comment
*/
```

- what is a preprocessor? it is unique for C. Before the source code is compiled, it gets automatically processed due to the presence of preprocessor directives in C. Key takeaway: In the C language, all of the preprocessor directives begin with a hash/pound (#) symbol. it allows for programs to be easier to read, develope and modify. it analyses your code beforehand and sends the necessary instructions to the actual compiler before the actual compilation. proprocessor statements are identified with symbol # and used in the beginning of the line. #include is an example. with #define we can create our own constants and macros or build our own library with #include statement.

  there are 5 types of preprocessors in C:

  1. Macros: In layman language, macros are the substitutes for strings that are used while defining a constant value. In programming terminology, a macro is a segment of code that has the ability to provide the inclusion of header files and specifies how to map a replacement output sequence in accordance to a well-defined series of steps a particular input sequence. For example,

     #define MAX 100

     Here, the string MAX has the assigned constant value of 100. Key takeaway: #define macro_template macro_expression does not terminate with a semicolon. There are 2 types of Macros:

     - Object-like macros: These macros are not capable of taking parameters.
     - Function-like macros: These macros are capable of taking parameters.

  2. File Inclusion: File inclusion is responsible for instructing the compiler to include a specific file in the source code program. Depending on the type of file included, file inclusion is categorized into two types, namely:

- Standard header files – These files refer to the pre-existing files, which convey a specific meaning to the compiler before the actual compilation has taken place.
- User-defined files – The C language gives the programmer the provision to define their own header files in order to divide a complex code into small fragments.

3. Conditional Compilation: Just like we use if-else statements for the flow of control over specific segments of code, in the same way, we use the concept of conditional compilation. Conditional compilation is a type of preprocessor that gives the programmer the power to control the compilation of particular segments of codes. It is done with the help of the 2 popular preprocessing commands, namely:
   - ifdef
   - endif

   also
   - #undef: As eccentric as it sounds, we use #undef directive to undefine the pre-existing, standard header or a user-defined header file.
   - #pragma: We use this type of preprocessor directive to enable or disable certain features. It is important to note that #pragma varies from compiler to compiler.

#include is a preprocessor directive. it is like import in other languages. in the above example the compiler is instructed to include in your program the contents of the file named “stdio.h”. It is called a header file because it usually places at the top. “.h” is its extention. In C++ all the header files may or may not end with the .h extension but in C, all the header files must necessarily end with the.h extension. Stdio.h, short for "standard input/output" is a standard C library that gives us chance to use many commands such as printf(). there are 2 ways to #include header files in C:

```
// < > is usually used for C's libraries
1. #include <stdio.h>


// " " is usually used for user-defined libraries
2. #include "stdio.h"
```

you can also create your own header file by creating it with .h extension in the same folder of your main.c and then import it in code. header files should always be in lowercase. Header files offer these features by importing them into your program with the help of a preprocessor directive called #include. Every C program should necessarily contain the header file <stdio.h> which stands for standard input and output used to take input with the help of scanf() function and display the output using printf() function.

Here is a table that explains missions of some preprocessors in C:

Preprocessor	Elucidation

#include	Used to insert a specific header from a file.
#define	Used as a replacement of a preprocessor macro.
#ifdef	Used when dealing with conditions. If the macro is defined, it returns true.
#endif	Used to close the preprocessor directive in accordance with a given condition.
#undef	Used to undefine a standard or user-defined header.
#pragma	Used to enable and disable certain features.
#ifndef	If the macro is not defined, it returns true.

- printf() metodu ile you display output to the screen:

```c
#include "stdio.h"

int main() {
    printf("the die is cast.");
    return 0;
}
```

now that we know how to press output to the screen, what about how to take input? Well, there are couple ways of accepting input from user in C but most common one is

- printf'teki f means formatted.
- scanf() function.

  It reads the input from standard stdin and scans the input according to the format provided. These formats are:

  - %s ⟶ strings
  - %d ⟶ integers
  - %c ⟶ characters
  - %f ⟶ floats (...)

  while printf() uses variable names, constants and expressions as argument, scanf() uses pointers. Here is 3 rules to use scanf():

  1. returns the number of items that it successfully reads
  2. if you use scanf() to read a value for one of the basic variable types we've discussed, precede the variable name with an & .

3. if you use scanf() to read a string into a character array, don't use a &.

scanf() örnek:

```c
#include <stdio.h>

int main() {
    //creating a string array of 100 max
    char str[100];

    // creating an integer named i
    int i;

    // asking for an input
    printf("enter an age: ");
    scanf("%d", &i);

    // asking for name
    printf("enter a name: ");
    scanf("%s", &str);

    // using what we have collected thus far
    printf("your name is %s and you are %d years old.", str, i);

    // saying C that code is finished
    return 0;
}
```

- #include is known as *preprocessive director*.
- C keywords: https://prnt.sc/QZxE5Xw-8SFa
- data stored in RAM is lost when computer turned off unlike harddrives. 8bit = 1byte.
- data types in C:

1. int

   it includes integral values.
   it can also take - sign preceding to denote the number is negative

   int type is a signed integer, that is it can be either positive, negative or zero

   you assign to ints numbers (10, 200, -5 etc.) and hexadecimal numbers (0xFFEF0D etc.)

   you cannot use spaces or comma between digits (1,200,000 is not a valid usage)

2. char
   tek bir ifadeden oluşan veri tipleri. Written ALWAYS in single quotes '' like 'a', '5', '\n' etc. It is used with %c in printf().

3. short …

   for instance short int is preferably used with small int numbers like 3, 5, 10 and uses less RAM than int type.

4. long …
   long double US_deficit_2017;
   1.234e+7L

5. float

   it contains values that have decimal places (30.5, 20., -0.0001 etc.)

   1.7e4 means 1.7 * 10^4

   printf(“%.2f”, myFloat) dersen virgülden sonra 2 basamak yazdırır.

   printf(“%5f”, myFloat) dersen totalde ilk 5 rakamı yazdırır, decimal kısmı dahil.

6. double
   it is the same with float only with roughly twice the precision

   represented by 64 bits

   all floating-point constants are taken as double values by C. if you want to express explicitly a float constant, append either an F or f to the end of the number (12.5f). double is much more precise than float (up to 12-16 digits after .).

7. long double

8. _Bool
   stores the values 0 and 1
   used for indication of yes/no, zero/one, true/false etc.
   it is used with %i or %u in printf() char.

9. enum
   enums are data types that allow a programmer to define a variable and specify the valid values that could be stored into that variable. for instance we can create a variable named myColor and it can only contain one of primary colors; red, blue, yellow and no other. to do this you first have to define enum type and give it a name and finally variable list in { }:

   ```
   enum primaryColors {red, yellow, blue};
   ```

   so in this primaryColors variable you can only store these three values like:

```
enum primaryColors = red;
enum primaryColors = yellow;
enum primaryColors = blue;
```

mesela enum veri tipini month adıyla yılın 12 ayını tutacak şekilde yazabilirsin.
örnek:

```
int main() {
    enum gender {male, female};
    enum gender myGender = male;
    enum gender yourGender = female;
}
```

enum örnek:

```
#include <stdio.h>

int main() {

    enum Company {
        GOOGLE, FACEBOOK = 500, XEROX, YAHOO, EBAY, MICROSOFT
    };

    enum Company xerox = XEROX;
    enum Company google = GOOGLE;
    enum Company ebay = EBAY;
    enum Company facebook = FACEBOOK;

    printf("value of google is: %d\n", google);
    printf("value of xerox is: %d\n", xerox);
    printf("value of ebay is: %d\n", ebay);
    printf("value of facebook is: %d\n", facebook);

    // facebook'a 500 değerini atadıgımız icin sonrakiler 501, 502 diye gidiyor
    return 0;
}
```

10. char

represents a single character like ‘a’, ‘6’, ‘;’ etc. anything between single quotes ‘’ is referred as character datatype. you can also declare character variables to be unsigned (just means they can’t be negative). double quotes “” in C however means it is a string. character strings and character datatype is not to be confused.

```
int main() {
    char myCharacter = 's';
    return 0;
}
```

- örnek:

```c
#include <stdio.h>
int main()
{
   char myChar;
   puts("what is your fav char?");
   scanf("%c", &myChar);
   printf("your fav char is %c", myChar);
   return 0;
}
```

ya da getchar() da kullanabilirsin:

```c
   myChar = getchar();

   // instead of

   scanf("%c", &myChar);
```

char yazdırmak için de putchar():

```c
#include <stdio.h>
int main()
{
   printf("my fav char is: ");
   putchar('a');
   return 0;
}
```

Type	Constant Examples	printf chars
char	'a', '\n'	%c
_Bool	0, 1	%i, %u
short int	—	%hi, %hx, %ho
unsigned short int	—	%hu, %hx, %ho
int	12, -97, 0xFFE0, 0177	%i, %x, %o
unsigned int	12u, 100U, 0XFFu	%u, %x, %o
long int	12L, -2001, 0xffffL	%li, %lx, %lo
unsigned long int	12UL, 100ul, 0xffeeUL	%lu, %lx, %lo
long long int	0xe5e5e5e5LL, 50011	%lli, %llx, &llo
unsigned long long int	12ull, 0xffeeULL	%llu, %llx, %llo
float	12.34f, 3.1e-5f, 0x1.5p10, 0x1P-1	%f, %e, %g, %a
double	12.34, 3.1e-5, 0x.1p3	%f, %e, %g, %a
long double	12.341, 3.1e-51	%Lf, $Le, %Lg

örnek:

```c
#include <stdio.h>
#include <stdbool.h>

int main()
{
    float x = 20.332;
```

```
    double y = 3.14159265358979e+11;
    bool boolVar = true;  // C99'da bool + true/false kullanmak istiyosan
                          // stdbool.h import etmelisin
}
```

float örnek:

```
#include <stdio.h>

int main() {
    float x = 3.24258894784562;
    printf("%f", x);
} >>> 3.242589
```

eğer virgülden sonra belirli sayıda küsurat yazdırmak istiyosan %.number metodunu kullan:

```
#include <stdio.h>

int main() {
    float x = 3.24258894784562;
    printf("%.3f", x);
} >>> 3.243 (yuvarladı)
```

- Tokens: Tokens in C language are the smallest possible unit of a program, that conveys a specific meaning to the compiler. It is the building blocks of a programming language. There are 6 types of tokens in C:

    1. keywords
    2. operators
    3. strings
    4. constants
    5. special characters
    6. identifiers

1. Keywords: Keywords in C language are the pre-defined & reserved words, each having its own significance and hence has a specific function associated with it. We can't simply use keywords for assigning variable names, as it would connote a totally different meaning altogether and would be erroneous. There are a total of 32 keywords offered in C:

auto break case char continue do default const double else enum extern for if goto float int long register return signed staticsizeofshort structswitchtypedef union voidwhile volatile unsigned

2. Identifiers: The C programmer has the provision to give names of his own choice to variables, arrays, and functions. These are called identifiers in C. The user may use the combination of different character sets available in the C language to name an identifier but, there are certain rules to be abided:

    1. First character: The first character of the identifier should necessarily begin with either an alphabet or an underscore. It cannot begin with a digit.

2. No special characters: C does not support the use of special characters while naming an identifier. For instance, special characters like comma or punctuation marks can't be used.
3. No keywords: The use of keywords as identifiers is strictly prohibited, as they are reserved words which we have already discussed.
4. No white space: White spaces include blank spaces, newline, carriage return, and horizontal tab, which can't be used.
5. Word limit: The identifier name can have an arbitrarily long sequence that should not exceed 31 characters, otherwise, it would be insignificant.
6. Case sensitive: Uppercase and lowercase characters are treated differently.

3. Constants: Often referred to as literals, constants, as the name itself suggests, are fixed values i.e. they cannot change their value during program run once they are defined.

```
const data_type variable_name = value;
```

there are 6 types of constants in C:

1. Integer constants: These are of the integer data type. For example, const int value = 400;
2. Floating constants: These are of the float data type. For example, const float pi = 3.14;
3. Character constants: These are of the character data type. For example, const char gender = 'f';
4. String constants: These are also of the character data type, but differ in the declaration. For example, const char name[] = ''DataFlair'';
5. Octal constants: The number system which consists only 8 digits, from 0 to 7 is called the octal number system. The constant octal values can be declared as, const int oct = 040; (It is the octal equivalent of the digit "32" in the decimal number system.)
6. Hexadecimal constants: The number system which consists of 16 digits, from 0 to 9 and alphabets 'a' to 'f' is called hexadecimal number system. The constant hexadecimal values can be declared as, const int hex = 0x40; (It is the hexadecimal equivalent of the digit 64 in the decimal number system.)

4. Strings: Just like characters, strings are used to store letters and digits. Strings in C are referred to as an array of characters. It is enclosed within double quotes, unlike characters which are stored within single quotes. The termination of a string is represented by the null character that is '\0'. The size of a string is the number of individual characters it has. In C, a string can be declared in the following ways:

```
// The compiler reserves 30 bytes of memory
char name[30] = ''DataFlair";

// The compiler reserves the required amount of memory
char name[] = "DataFlair";
```

```
// How a string is represented as a set of characters.
char name[30] = { 'D' , 'a' , 't' , 'a' , 'F' , 'l , 'a' , 'i' , 'r' , '\0' };
```

5. Special Symbols: Apart from letters and digits, there are some special characters in C:

- [] : The opening and closing brackets of an array indicate single and multidimensional subscripts.
- () : Used to represent function declaration and calls, used in print statements.
- {} : Denote the start and end of a particular fragment of code which may be functions or loops or conditional statements.
- , : Separate more than one statements, like in the declaration of different variable names in C.
- # : A preprocessor directive, utilize for denoting the use of a header file.
- * : To declare pointers, used as an operand for multiplication.
- ~ : As a destructor to free memory.
- . : To access a member of a structure.

6. Identifiers (Operators): Operators in C are tools or symbols, which are used to perform a specific operation on data. Operations are performed on operands. Operators can be classified into three broad categories, according to the number of operands used. Which are as follows:

- 1. Unary: It involves the use of one a single operand. For instance, '!' is a unary operator which operates on a single variable, say 'c' as !c which denotes its negation or complement.
- 2. Binary: It involves the use of 2 operands. They are further classified as: Arithmetic Relational Logical Assignment Bitwise Conditional
- 3. Ternary: It involves the use of 3 operands. For instance, ?: Is used in place of if-else conditions.

- we know that main() function is the starting point of any C program. When it is run, two arguments are actually passed to the function:
    1. argc (argument count) is an integer value that holds the number of arguments typed on the command line. it is the number of parameters.
    2. argv (argument vector) is an array of character pointers (strings). can either be shown as char **av or char *av[]

- backslash (escape) characters in C:

```
\n (New line) - We use it to shift the cursor control to the new line
\t (Horizontal tab) - We use it to shift the cursor to a couple of spaces to the right in the same line.
```

```
\a (Audible bell) - A beep is generated indicating the execution of the program to alert the user.
\r (Carriage Return) - We use it to position the cursor to the beginning of the current line.
\\ (Backslash) - We use it to display the backslash character.
\’ (Apostrophe or single quotation mark) - We use it to display the single-quotation mark.
\” (Double quotation mark)- We use it to display the double-quotation mark.
\0 (Null character) - We use it to represent the termination of the string.
\? (Question mark) - We use it to display the question mark. (?)
\nnn (Octal number)- We use it to represent an octal number.
\xhh (Hexadecimal number) - We use it to represent a hexadecimal number.
\v (Vertical tab)
\b (Backspace)
\e (Escape character)
\f (Form Feed page break)
```

variables

- variables: değişken tanımlama. unlike Python, variables can start with _ in C but you can’t:
    1. start with a number in the name of your variable (2var =)
    2. use space (. as =)
    3. use special characters (&, $, % etc.) (myVar^^ =)
    4. use keywords (int, def etc.) (for =)

unlike Python when you create a variable you have to specify the type of the data you’ll assign to that variable. you can assign value to a variable by 2 ways:

1. you may first create the variable and then assign a value later on:

```c
#include <stdio.h>
int main() {
    int i;

    i = 110;
}
```

2. you may create&assign it at the same time which is called initialization:

```c
#include <stdio.h>

int main() {

    int i = 110;

}
```

you can also initialize multiple variables on the same line:

```c
#include <stdio.h>

int main() {

    int i = 110, j = 200, f = 250;

}
```

or

```c
#include <stdio.h>


int main() {

    int i = 110, j = 200, f = 250;

    char s[] = "bugra";

    printf("%d %d %d %s", i, j, f, s);

} >>> 110 200 250 asd
```

but you cannot assign them respectively in one line like Python:

```c
int myVar, yourVar, herVar = 500, 600, 700; >>> ERROR
```

- char, int, short ve long integer tanımlamak için kullanılır.Most of the times, you'll likely use an int to store an integer. But in some cases, you might want to choose one of the other 3 options. The char type is commonly used to store letters of the ASCII chart, but it can be used to hold small integers from -128 to 127 . It takes at least 1 byte. int takes at least 2 bytes. short takes at least 2 bytes. long takes at least 4 bytes.
- For all the above data types, we can prepend unsigned to start the range at 0, instead of a negative number. This might make sense in many cases.

    1. unsigned char will range from 0 to at least 255
    2. unsigned int will range from 0 to at least 65535
    3. unsigned short will range from 0 to at least 65535
    4. unsigned long will range from 0 to at least 4294967295

  eğer unsigned diyip yukarıdaki sınırları geçersen sıfırlar (mesele 255 değerli char'ı 1 arttırırsan 0 verir)

```c
#include <stdio.h>

int main(void) {


    unsigned char j = 255;
    j = j + 10;
    printf("%u", j);


} >>> 9
```

- sizeof() metodu ile bir şeyin boyutunu öğrenebilirsin:

```c
#include <stdio.h>

int main(void) {
    printf("char size: %lu bytes\n", sizeof(char));
    printf("int size: %lu bytes\n", sizeof(int));
    printf("short size: %lu bytes\n", sizeof(short));
    printf("long size: %lu bytes\n", sizeof(long));
    printf("double size: %lu bytes\n", sizeof(double));
}
```

- constant: A constant is declared similarly to variables, except it is prepended with the const keyword, and you always need to specify a value. it is common to declare constants uppercase:

```c
const int AGE = 32;
```

another way to define constants is:

```c
#define AGE 32
```

örnek:

```c
#include <stdio.h>

int main()
{
    const int myConst = 50;
    printf("%d", myConst);
    return 0;
} >>> 50
```

or

```c
#include <stdio.h>
#define myNum 500

int main(void) {
   printf("your number is %d", myNum);
}
```

- matematiksel operatörler python'daki gibidir

    1. ! ⟶ not
    2. && ⟶ and
    3. || ⟶ or

- there are 3 kinds of operands:

    1. binary: like + or *. It takes at least 2 operands to operate. a + b gibi
    2. unary: - (negatif) gibi. It only needs 1 operand to denote. -4 gibi.

3. ternary: The ternary operator is the only operator in C that works with 3 operands, and it's a short way to express conditionals. Ternary operator is consist of two symbols: ? and :

   that's how it looks:

```
condition ? expression1_if_true : expression2_if_false
```

   If condition is evaluated to true , then expression1 statement is executed, otherwise expression2 is executed.

   örnek:

```
// x will be 25 if y is greater than 7

// otherwise x will be 50

x = y > 7 ? 25 : 50;
```

   this is same as saying:

```
if (y > 7)

    x = 25;

else

    x = 50;
```

   örnek:

```
#include <stdio.h>

int main(void) {
    int x, y;

    y = 6;

   x = y > 7 ? 25 : 50;

    printf("x is: %d and y is: %d", x, y);
}
```

   örnek2:

```
#include <stdio.h>

int main()
{

    // if a < b then c=a, if it is not c=b
    int a = 10, b = 20, c;

    c = (a < b) ? a : b;

    printf("%d", c);
```

```
} >>> 10
```

örnek3:

```
#include <stdio.h>

int main()
{
    int m = 5, n = 4;

    (m > n) ? printf("m is greater than n that is %d > %d", m, n)
    : printf("n is greater than m that is %d > %d", n, m);

    return 0;
}
```

-

basic operators

- operator is the actual symbols and operand are the arguments to the symbol. operands can be constants, variables or blending of two.
    1. logical operators (&&, ||, !)
    2. arithmetic operators (+, -, *, /)
    3. assignment operators
    4. relational operators (<, >, ≠)
    5. bitwise operators
    6. Increment/Decrement Operators : ++,-

1. logical operator: sometimes called a boolean operator returns a boolean result that's based on the expressions given:
    a. &&: called AND operator. If both the operands are non-zero, then the condition becomes true.
    b. ||: called OR operator. If any of the operands are non-zero, then the condition becomes true.
    c. NOT: called NOT operator. It is used to reverse the logical state of the operand. If a condition is true, then logical NOT operator will make it false.

2. arithmetic operator: is a mathematical function that takes two operands and performs a calculation on them. (+, -, *, /, %, ++, --)

3. assignment operator: sets variables equal to given values (=, +=, -=, *=, /=, %=, ⟸, ⟹, &=, ^= ⊨ etc.)

4. relational operator: compares variables against each other (=, ≠, <, >, ≥, ≤)

5. bitwise operator: looks like logical operators but actually completely different. Operates on the bits in integer values. it is not usually used (&, |). It is used for efficiency of data. For example a single integer (which is usually 4 bytes max [32bits]) can hold several characteristics of a person:
   a. store whether person is male of female with one bit
   b. use other 3 bits to specify whether s/he speaks GER, FRA, ENG etc.
   c. another bit to record if s/he earns 50.000+ etc. etc.

6. increment/decrement operators:
   a. ++
   b. --

- statements:

   1. declaration statement:

      ```
      int myVar;
      ```

   2. assignment statement:

      ```
      myVar = 5;
      ```

   3. function call statement:

      ```
      printf("myVar");
      ```

   4. structure statement:

      ```
      while (myVar < 20) myVar++;
      ```

   5. return statement:

      ```
      return 0;
      ```

   in C, every line that ends with ; is a statement.

control flow

- statements in your code are generally read from top to bottom, that is *procedural*. control flow statements however, break up the flow of execution by employing decision making, looping and branching, hence enabling your program to conditionally execute particular blocks of code.

   1. decision-making statements (if-then, switch, goto)

   2. looping statements (for, while, do-while)

   3. branching statements (break, continue, return)

   1. decision-making statements: they are structures that require programmer to specify one or more conditions to be evaluated or tested by the program. If a condition is true then a statement will be executed, if a condition is false then other statements are executed.

   2. looping statements: the statements are executed sequentially until they are stopped. Allows for multiple executions in a very short time. When loop is stopped (break) it will

skip all the loop block and go on with the next lines of code. A loop becomes an infinite loop if a condition is never false.

a. while loop: It repeats a function while the given condition is true, checking it everytime

b. for loop: It executes repeatedly for a given range.

c. do-while loop: same with while except that it tests the condition at the end of the body

d. nested loop: you can use multiple loops, do-while, for loops inside each other

3. branching: Branching statements allow the flow of execution to jump to a different part of the program. these are

* continue
* break
* return
* goto

conditionals

- C offers us 3 ways to use conditionals.

1. if statement:

basic syntax of if:

```
if (...) {
    printf("aaa");
}
else if {
    printf("bbb");
}
else {
    printf("ccc");
}
```

First it checks if statement. if it is false it goes to else if statement, if that too is false than else block will be executed. Notice that unlike python you don't put : at the end of the if line

If Else statements can also be nested inside each other. Nested If-Else's syntax is:

```
if (condition1) {
    //Nested if else inside the body of "if"
    if(condition2) {
       // state1
    }
```

```
    else {
        // state2
    }
}
else {
    // state3
}
```

eğer condition1 doğruysa onun içine girer ve daha da spesific bir şart olan condition2'yu kontrol eder. eğer o da doğruysa state1'i çalıştırır değilse state2'yi çalıştırır. eğer condition1 doğru değilse state3'ü çalıştırır.

nested if-else örnek:

```
#include <stdio.h>

int main() {

   int var1, var2;
   printf("Input the value of var1:");
   scanf("%d", &var1);
   printf("Input the value of var2:");
   scanf("%d",&var2);

   if (var1 =/= var2) {
           printf("var1 is not equal to var2\n");

           if (var1 > var2)
                   printf("var1 is greater than var2\n");


           else {
                   printf("var2 is smaller than var1\n"); }
   }

   else
   printf("var1 is equal to var2\n");

   return 0;
}
```

2. switch statement: If you need to do too many if / else / if blocks to perform a check, perhaps because you need to check the exact value of a variable, then switch can be very useful to you. It uses 4 keywords; switch, case, default and break. Its syntax is:

```
switch (expression)
{
    case value1:
        program statement
        ...
        break;
    case value2:
        program statement 2
        ...
        break;

    default:
        program statement 3
        ...
        break;
}
```

You should end every statement with break. Default here works as else.

You can provide a variable as condition, and a series of case entry points for each value you expect:

```
int a = 1;
switch (a) {
    case 0:
        /* does something */
        break;
    case 1:
        /* does something else */
        break;
    case 2:
        /* does something else */
        break;

   default:
        /* handles all the other cases */
        break;
}
```

We need a break keyword at the end of each case, to avoid the next case to be executed when the one before ends. Everything you can do in switch you can in If but it is easier if there are lots of Ifs.

```
#include <stdio.h>

int main()
{
    float value1, value2;
    char operator;

    printf("type in your expression:");
    scanf("%f %c %f", &value1, &operator, &value2);
```

```
    switch (operator)
    {

    case '+':
        printf("%.2f\n", value1 + value2);
        break;

    case '-':
        printf("%.2f\n", value1 - value2);
        break;

    case '*':
        printf("%.2f\n", value1 * value2);
        break;

    case '/':
        if (value2 == 0)
            printf("division by zero \n");
        else
            printf("%.2f\n", value1 / value2);
        break;

    default:
        break;
    }

    return 0;
}
```

3. goto statement: has 2 parts; goto and label name. Its syntax:

```
goto label;
... .. ...
... .. ...
label:
statement;
```

The label is an identifier. When the goto statement is encountered, the control of the program jumps to label: and starts executing the code.
goto örnek:

```
// Program to calculate the sum and average of positive numbers
// If the user enters a negative number, the sum and average are displayed.

#include <stdio.h>

int main() {

   const int maxInput = 3;
```

```
   int i;
   double number, average, sum = 0.0;

   for (i = 1; i ⩽ maxInput; ++i) {
      printf("%d. Enter a number: ", i);
      scanf("%lf", &number);

      // go to jump if the user enters a negative number
      if (number < 0.0) {
         goto jump;
      }
      sum += number;
   }

jump:
   average = sum / (i - 1);
   printf("Sum = %.2f\n", sum);
   printf("Average = %.2f", average);

   return 0;
}
```

- 0’dan 100’e yazdırma:

```
#include <stdio.h>

int main()
{
   int a;

   for (a=0; a⩽100; a++)
   {
      printf("%d\n", a);
   }
   return 0;
}
```

- info:

```
if body consists of a single statement you don’t have to use {}
```

- örnek program to determine if a number is even or odd:

```
#include <stdio.h>

int main()
{
    int i;

    printf("please enter a number:");
```

```
    scanf("%d", &i);

    if (i % 2 == 0)
        printf("girdigin sayı cift sayi");
    else
        printf("girdigin sayi tek sayi");

    return 0;
}
```

- örnek program to determine if a number is -, + or 0:

```
#include <stdio.h>

int main()
{
    // declared 2 var to hold number and its sign
    int number;
    int sign;

    // prompting user to enter a number
    printf("please enter a number: ");
    scanf("%i", &number);

    // conditional block
    if (number < 0)
        sign = -1;
    else if (number == 0)
        sign = 0;
    else  // must be a positive if none of the above
        sign = 1;

    printf("Its sign is= %i\n", sign);

    return 0;
}
```

-

#loops

- there are 2 kinds of loops:
    1. counter control loops: repeats a certain number of times; 10, 20, 30 et.c
    2. sentinel control loops: number of repeat is not determined. Depends on the given condition
- there are 4 names for loops in C:
    1. for loops
    2. while loops

3. do while loops

4. nested loops

1.for loops: are used with the for keyword. for(a; b; c) {function} olmak üzere 4 parçadan oluşur

- a; initial condition as in i = 1
- b; continuation condition: is checked for its trueness at the beginning (called pretest loop) as in i ⩽ 30
- c; increment as in i++

```
for (ilk durum; devam koşulu; değişim) {
    /* do something */;
}
```

2. while loop: everything you can do in the for loop, you can do in the while loop

```
while (expression) {
    /* do something */

     /* do something */
}
```

3. do-while loop: While loops are great, but there might be times when you need to do one particular thing: you want to always execute a block, and then maybe repeat it. this is done with do while loops. unlike for and while this time condition is at the bottom (thus called post-test loop) and hence it is checked after executing the body block.

```
do {
    /* do something */
}

while (expression);
```

4. nested loop: Loops that are put inside one another, consists of inner and outer loop(s). You can use a for loop inside a while loop or vice versa. You can nest many many loops in each other.

```
#include <stdio.h>

int main() {
    for(a; b; c)
    {
        // initialize variables used in inner loop
        i = ...;
        j = ...;
```

```
        for (i, j, ++j) {
            body1
            body2
        }
    }
}
```

continue: situation may arise where you do not want to end a loop, but you want to skip the current iteration in case condition is true:

```
#include <stdio.h>

int main() {
    enum Day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
    for(enum Day day = Monday; day ⩽ Sunday ; ++day)
    {
        if (day == Wednesday)
            continue;
        printf("It is not Wednesday! \n", day);

    }
}
```

enums under the hood are integers (Monday⟶0, Friday⟶4 etc.), that is why we could increment it by 1

break: helps you exterminate the loop whenever break is read.

```
for (int i = 0; i ⩽ 10; i++) {
    if (i == 4 && someVariable == 10) {
        break;
    }
}
```

Having this option to break out of a loop is particularly interesting for while loops (and do while too), because we can create seemingly infinite loops that end when a condition occurs, and you define this inside the loop block:

```
#include <stdio.h>

int main() {

    int i;
    i = 1;
    while (i < 20)
    {
        printf("%d", i);
        i++;
        if (i == 10) break;
```

```
    }
}
```

normalde 20'ye kadar yazdırması gerekiyordu ama 10'u görünce break etti. and also if you use break in an inside loop, it will only terminate the inner loop.

- while örnek:

```
#include <stdio.h>

int main(void) {


    int sayi = 1;


    while (sayi ⩽ 8) {
        printf("%d\n", sayi);
        ++sayi;


    return 0;


} >>> 1 2 3 4 5 6 7 8. this loop is a counter controlled loop. bc it is always executed certain times.
```

örnek2:

```
#include <stdio.h>

int main(void) {

    int num;

    while (num ≠ -1)
    {
        printf("please enter a positive num:");
        scanf("%d", &num);

        while (num > 0)
        {
           printf("%d ", num);
           num--;
      }
      break;
    }
    return 0;
}

// this is a sentinel controlled loop because
// this loop may execute 5 times or a hundred
// times. It all depends on the input from the keyboard.
```

- do-while örnek:
-
- what loop should you use?
    - first decide if you need a pre-test or post-test (if it is pre-test it may not ever be executed if its condition is not true because its condition is checked before the execution but if it is post-test it is at least executed once because it checks the condition after its execution)
    - a for loop is better option when loop involves initializing and updating a variable
    - while is better when conditions are otherwise
    - for is usually used for counter controlled loops whereas while with sentinel controlled loops
-

arrays

- an array in C is a variable that holds multiple values. every value however, must be of the same type in an array. what is the structure of an array like?

```
int myArray[] = {};
```

- you can define an array of int values like that:

```
int prices[5] = {1,2,3,4,5};
```

but you can also give values later on:

```
int prices[5];
prices[0]= 1;
prices[1]= 2;
prices[2]= 3;
prices[3]= 4;
prices[4]= 5;
```

or you can use a loop to assign values

```
int prices[5];


for (int i= 0; i ⩽ 5; i++)
{
    prices[i] = i + 1;
}
```

- you must specify the size of the array. You can use a constant to define the size:

```
int myArray[50];
```

or

```
const int myArray = 5;
```

- initialization in array:

```
int sayi[4] = {0, 0, 0, 0};
```

you don't have to initialize all the elements, you can initialize (give value) to first few of them:

```
// ilk 3üne değer verdik kalan 97 değer 0'a eşitlendi
int sayi[100] = {15, 10, 22};
```

c99'dan sonra spesifik indexlere de value atayabilirsin:

```
int sayi[100] = {[43] = 15, [10] = 20};
```

- you can also use 2-dimensional arrays as in a matrix.

```
#include <stdio.h>

int main() {
    int matrixExample[3][4] = {
            {10, 20, 30, 40},
            {5, 20, 23, 80},
            {101, 102, 103, 104}
    };
}
```

that matrix consists of 3 rows and 4 columns for a total of 12 elements. You should separate matrices with a , except for the final row. If you had [][][] you'd have a 3-dimensional array, 4 brackets for 4 etc. etc.

as with the 1-dimensional arrays it is not necessary to initialize to every element of every row, the rest of the elements will set to 0:

```
#include <stdio.h>

int main() {

    int matrixExample[3][4] = {
            {10},
            {5, 20},
            {101, 102, 103},
    };

    return 0;
}
```

you can also set values for spesific indexes in matrices starting from C89:

```
int matrix[4][3] = { [0][0] = 1, [1][1] = 3, [2][2] = 5};
```

- arrays are fixed sizes. Once the length of the array is specified, it cannot be changed.

Valid and invalid declarations of an array

```
int n = 5;
int m = 8;
float a1[5];          // yes
float a2[5*2 + 1];    // yes
float a3[sizeof(int) + 1]; // yes
float a4[-4];         // no, size must be > 0
float a5[0];          // no, size must be > 0
float a6[2.5];        // no, size must be an integer
float a7[(int)2.5];   // yes, typecast float to int constant
float a8[n];          // not allowed before C99, VLA
float a9[m];          // not allowed before C99, VLA
```

-

POINTERS

- when you create things in some programming languages (Java for example) language automatically allocates memory and afterwards deletes this memory space for you. It is called garbage collector. C however, doesn't do this so we have to do this manually. Every memory spaces are called addresses and usually shown with hexidecimal numbers. so what we call pointer is but an int variable storing a spesific address.

```
int *pntr = 40;
```

  - *pntr diyerek pntr isminde bir pointer yarattık. başında * olmazsa pointer oldugunu anlayamaz program
  - int de bu adresin refere ettiği değerin türü
  - 40'da value

- A pointer is the address of a block of memory that contains a variable. yani ram'deki kaydettiğin bir bilginin kayıtlı olduğu konumun kodudur. mesela:

  int age = 37;

  bunun pointer ını öğrenmek için:

  printf("%p", &age); /* 0x7ffeef7dcb9c */

  nasıl ki integerlarla %d ya da %i kullanıyosak pointerlarla da %p kullanırız.

- örnek

```
// an int valued 50
int i = 50;


// a pointer (bc it has *) containing the adress of i (50)
// with the help of & symbol (it means it is an adress)
int *myPointer = &i;
```

  * symbol is always used when declaring a pointer

  to get the value of the variable i, you can use * to *dereference* the pointer

- you can also initialize a pointer with the value NULL meaning that it doesn't point to anything:

```
int *pntr = NULL;
```

- * is used to dereference a pointer whereas & is used for taking the pointer address. never derefence an uninitialized pointer. when declaring a pointer that does not point to anything, we should initialize it to NULL.
- we can also assign the pointer address to a variable:

```
int adress = &age;
```

- We can use the pointer operator * to get the value of the variable an address is pointing to:

```
int age = 37;
int *address = &age;
printf("%u", *address); /* 37 */
```

- örnek:

```
#include <stdio.h>

int main()
{
   int num = 0;
   int *pNum = NULL;

   num = 10;
   printf("number's address is: %p\n", &num);
   printf("number's value is: %d\n\n", num);

   pNum = &num;

   printf("pNum's address is: %p\n", (void*)&pNum);
   printf("pNum's size is: %d bytes\n", sizeof(pNum));
   printf("pNum's value is: %p\n", pNum);
   printf("value pointed to %d\n", *pNum);
   return (0);
}
```

- void pointers: type named void means absence of any type and a pointer with void* type can containt the address of any data type. it is like joker.

summary

- basically, our RAM (memory) consists of contiguous address numbers such as 0x1000, 0x1004, 0x1008, 0x100C ... The cells in memory are used to store the values of variables that the user define. Pointer of a variable then, is the int variable that stores the *address number* of the cell that stores the aforementioned variable.

```
int main()

{

   int a = 4;

   int *aPtr = &a;
```

```
    printf("a is: %d\n", a);        // 4
    printf("aPtr is: %d\n", *aPtr); // 4. if not for *, it'd give random nums


    /* pointerlarda * 2 şekilde kullanılır
        1) bir pointer degiskeni tanımlarken onun pointer oldugunu belirtmek icin
        int *myPtr = &myVar;
        2) değişkene pointer tanımladıktan sonra değişkenin değerine tekrardan ulaşmak için
        printf("pointer'ımın tuttugu adresteki deger: %d", *pointer); AKA dereferencing
    */
    int b = *aPtr;
    printf("b is: %d", b); // 4
}
```

pointer arithmetics

- the real power of using pointers to arrays comes into play when you want to sequence through the elements of an array. for instance *valuesPtr refers to the first integer of the values array, that is, values[0]. for instance if you want to refer to values[3], you can use *(valuesPtr + 3). so the formula is:

```
*(valuesPtr + i) == values[i]
```

functions

- Functions are the way we can structure our code into subroutines that we can:

    1. they have a name
    2. they specify a return value
    3. they can have arguments
    4. they have a body, wrapped in {}. Execute edilecek buna yazılır

  if the function has no return value we simply use the void before the function name. Otherwise we must specify the function return value (int for an integer, const char * for a string etc.). You cannot return more than one value from a function.

- basic outline of a function:

```
return_type function_name(parameters separated by commas)
{
    statements
}
```

  if there are no statements in the body block, the return type must be "void" and the function will not do anything.

- the name of the function must start with a letter. It can be anything except keywords or the name of another function, that is you can't have two functions with the same name. Such a case is called function overloading.
- hello world yazdırırken bile fonksiyon kullanırız (main() fonksiyonu):

```c
#include <stdio.h>
int main(void) {
    printf("Hello, World!");
}
```

printf() function takes 2 arguments:

  - first is the string/int etc. that is going to be printed
  - variables or expressions of this type with an &

- A function can have arguments. They are optional. If it does not have them, inside the parentheses we insert void , like this:

```c
void funcName(void) {
    /* code block */
}
```

In this case, when we invoke the function we'll call it with nothing in the parentheses:

```c
doSomething();
```

if we have one parameter, we specify the type and the name of the parameter:

```c
void doSomething(int value) {
    /* code block */
}
```

when we invoke the function we'll call it like:

```c
doSomething(3);
```

- We can have multiple parameters, and if so we separate them using a comma, both in the declaration and in the invocation:

```c
void doSomething(int value1,int value2) {
    /* ... */
}
```

and

```c
doSomething(3,4);
```

- Inside a function, you can declare variables:

```c
void myFun(int value) {
    int doubleValue = value * 2;
}
```

A variable is created at the point of invocation of the function, and is destroyed when the function ends, and it's not visible from the outside.

- Inside a function, you can call the function itself. This is called recursion and it's something that offers peculiar opportunities
- how to define a function in C:

  first line of any function is called header. It contains the name and parameters of the function.

  difference between argument and parameter: arguments have to do with when invoke a function while parameters have to do with function definition and declaration. if a function takes no parameters you should (but not have to) write void between (). Parameters are good for passing data to a function.

- gets() & puts():

  bu iki fonksiyondan birincisi klavyeden string input alır ikincisi de alınanı yazdırır. printf() & scanf()'in daha basit versiyonudur. gets() sadece string alırken scanf() sayısal veri de alabilir fakat sadece metin için gets() kullanmak daha mantıklıdır.

```c
#include <stdio.h>

int main()
{
   char name[20];
   printf("who are you?: ");
   gets(name);
   printf("hi, %s. Nice to meet you.", name);
   return (0);
}
```

  - gets(name) is same as scanf("%s", name)

  unlike printf(), puts() always put a \n at the end automatically.

```c
#include <stdio.h>

int main()
{
   puts("hello, world");
   return (0);
}
```

  fakat puts()'ta printf()'teki gibi ("%s", name) diyemezsin çünkü 1 parametre alabilir.

- fonksiyon örnek:

```c
#include <stdio.h>

void multiplyNums(int a, int b)
{
    int result = a * b;
    printf("the product of %d multiplied by %d is: %d\n", a, b, result);

}
```

```
int main(void)
{
    multiplyNums(5, 10);
    multiplyNums(2, 5);
    multiplyNums(20, 3);
}

>>>

the product of 5 multiplied by 10 is: 50

the product of 2 multiplied by 5 is: 10

the product of 20 multiplied by 3 is: 60
```

- Ideally a function should take 5 parameters at most.
- tanımladığımız değişkenlerin tıpkı python'daki gibi farklı scope'ları olabiliyor. yani bir kod blogunun altında geçerli ve tanımlı olan bir değişken o kod blogunun dışında kullanıldığında C tarafından tanınmayabiliyor. 2 tür değişken vardır:
    - global variables
    - local variables

1. global variables are accepted everywhere in the code whereas the local variables are special for a function - that is, if you declare a variable inside a function that variable will be a local one. globals however are declared outside of function block

```
#include <stdio.h>
int main(void) {
    int age = 32;
}
```

2. Local variables are only accessible from within the function, and when the function ends they stop their existence. They are cleared from the memory (with some exceptions).

   on the other hand, a variable declared outside of a function will be called global variable:

```
#include <stdio.h>
int age = 24;
int main(void) {
    /* ... */
}
```

Global variables are accessible from any function of the program, and they are available for the whole execution of the program, until it ends.

- örnek:

```
// this is a global variable
int myGlobal = 0;

int main() {
```

```
    // this is a local variable and can access myGlobal and myLocal
    int myLocal = 0;
    return 0;
}

void myFunc() {
    // this as well is a local variable and can myGlobal and x, but
    // cannot access myLocal
    int x;

}
```

- generally global variables is to be avoided for it is hard to find a bug when a global variable is problematic. It also complicates your code. C++ and Java even abandoned this concept for its dangers.
- mutlak değer hesaplayan basit formül:

```
// mutlak değer bulma

#include <stdio.h>

float absoluteValue (float a)
{
    if (a < 0)
        a = -a;
    return a;
}

int main()
{
    float result;
    result = absoluteValue(-324);
    printf("%.2f", result);
}
```

-
- EBOB (Greatest Common Diviser) hesaplayan fonksiyon:

```
// iki sayının EBOBUNU bulma

#include <stdio.h>

// obeb hesaplayacak fonksiyonun prototipini çağırdık
int gcd(int a, int b);

int main()
{
    int result = 0;
    // 150 ile 35'in ebobu
```

```
    result = gcd(150, 36);
    printf("%d", result);
    return (0);
}

int gcd(int a, int b)
{
    int temp;

    while (b != 0)
    {
        temp = a % b;
        a = b;
        b = temp;
    }
    return a;
}
```

-
- atoi(): turns Ascii to Integer

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

int main()
{
   int deneme;
   deneme = atoi("8700");
   printf("%d", deneme);
   return (0);
} >>> 8700
```

-

character strings

- string " " arasındaki her şeydir.

  char ise ' ' arasındaki tek haneli girdidir

```
myChar = '+';
myStr = "bugra";
```

- printf(“ “) içinde “ ya da \ yazdırmak için başına \ koymalısın aka escape sequence:

```
#include <stdio.h>

int main() {
    printf("\" \"");
} >>> " "
```

- her stringin sonunda null character denilen \0 kodlu bir sembol vardır. C stringin bittiğini bu koddan anlar. so the length of a string is always one greater than the number of characters in the string. don't mix NULL keyword with null character (\0):

```
#include <stdio.h>

int main() {
    printf("character known as null whose code is \0 ends a string");
} >>> character known as null whose code is
```

charlarda durum bu değildir. o yüzden 'x' 1 karakterden oluşurken "x" 2 karakterden oluşur

- C'de String veri tipi yoktur bunun yerine array of char tanımlarız.

```
char myStr[20];
```

max 19 karakterden (+ \0) oluşan bir string tanımladık.

```
char word [] = {'b', 'u', 'g', 'r', 'a'};
// or
char word[] = "bugra";
```

içine sayı koymazsan compiler otomatik uzunlugunu hesaplar.

- örnek:

```
#include <stdio.h>

char message[] = "benim adim bugra";

int main() {
    printf("%s", message);
}
```

- kullanıcıdan bir stringi input olarak alma:

```
#include <stdio.h>

int main()
{
    char input[10];
    printf("please enter your name:");
    scanf("%s", input);
    printf("your name is: %s.", input);
}
```

- str_length hesaplama:

```
#include <stdio.h>

int ft_strlen(char str)
{
    unsigned int a;
    a = 0;
```

```
    while (str[a] != '\0')
        a++;
    return a;
}
```

- breaking a string sentence into words (separated by , - etc.) is called tokenizing and its function is strtok().
- most used string functions
    - strlen (string uzunluğunu hesaplar)
    - strcpy & strncpy (stringi kopyalar & belirli bir karaktere kadar)
    - strcmp & strncmp (stringi karşılaştırır & belirli bir karaktere kadar)
    - strcat & strncat (stringi birleştirir & belirli bir karakter kadar)
    - strchr & strstr (string içinde char & string arar bulunca indexi verir)
    - strtok() (stringi kelimelere böler, tokenizing)
    - isalpha(), isdigit(), isupper(), islower() vs. (analyzes string)

  tüm bunlar <string.h> kütüphanesinde mevcuttur.
- how to convert strings?
    - toupper()
    - tolower()
    - atoi(): ascii to integer. turns ‘430’ of char into 430 of integer.
    - atof(): ascii to double. turns '10.5' of char into 10.5 of double.
    - atol(): ascii to long.
    - atoll(): ascii to long long.
-

CONSTANTS

- constant değeri değişmeyen değişken demektir (seems ironic).

  ```
  int speed = 50;
  ```

  bundan sonra kodda 50 yerine speed de yazabilirsin. bu değeri değiştirmek için bu satırda 50’yi silip 40 yazarsan altındaki tüm speed’ler 40 olacaktır. ama bunun yerine #define metodu kullanmak daha mantıklıdır:

  ```
  #define speed 50
  ```

  or

  ```
  #define name "Bugra"
  ```
- if you do not want a variable to change, you can create it in form of constant. the keyword we use to create constants is #define as in:

  ```
  #define PiNumber 3.14
  ```

all the PiNumber usages after this line will be calculated as 3.14 float.

- #define örnekleri:

```
#define piNum 3.14
#define myName "bugrahan karamollaoglu"
#define nullCharacter '\0'
#define myChar 'a'
```

- #define usage in action:

```
#define piNum 3.14

#include <stdio.h>

int main()
{
    int result;
    result = piNum * 2;
    printf("%d", result);
} >>> 7
```

#define method cannot be used locally.

- normal variable'ı constant yapmak istiyosan const keywordunu kullanırsın:

```
const int MONTHS = 12;
```

this is called symbolic constant and by doing that you cannot alter the value of months later on.

- constant kullanmanın üçüncü bir yolu ise enums denilen şeydir.
- örnek

```
const char unchangedStr[] = "hello";
```

- when using const with pointers things get shapy. There are two things that you may have in mind:
  - you use const because you don't want the address of the pointer to change

    ```
    int myVal = 30;
    int *const myPointer = &myVal;
    ```

  -
  - you use const because you don't want the value the pointer points to to change

    ```
    long myVal = 9999L;
    const long *myPointer = &myVal;
    ```

    compiler 9999L değerini değiştiren herhangi bir satır bulursa hata verecektir.
    mesela şu satır hata verir:

    ```
    *myPointer = 8888L;
    ```

    but you can still modify the value:

```
myVal = 6666L;
```

- o

INPUT & OUTPUT

- C is a small language, and the "core" of C does not include any Input/Output (I/O) functionality. This is not something unique to C, of course. It's common for the language core to be agnostic of I/O. In the case of C, Input/Output is provided to us by the C Standard Library via a set of functions defined in the stdio.h header file. You can import this library using:

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

  this library provides us among many,

  - o printf()
  - o scanf()
  - o sscanf()
  - o fgets()
  - o fprintf()

- we have 3 kinds of I/O streams in C:

  - o stdin (standard input)
  - o stdout (standard output)
  - o stderr (standard error)

- Some functions are designed to work with a specific stream, like printf() , which we use to print characters to stdout . Using its more general counterpart fprintf() , we can specify the stream to write to.

- printf(): is one of the most common methods in C. but when you print, you should also add the type of the value for instance %d for a decimal integer:

  ```
  ing age = 32;
  printf("my age is %d", age);
  ```

  - o %c for a char
  - o %s for a string
  - o %f for a float
  - o %p for pointers

  örnek:

  ```
  #include <stdio.h>

  int main() {


      int sum = 30;
      printf("the sum is %d", sum);
  ```

```
}

>>> the sum is 30
```

as you see using printf(“text”); is easy, but when you want to use a variable in printf(), you have to specift the type of the content of that variable with %... for every variable. for instance above what %d is doing is that it gives C a hint about the content of what will come later as the 2nd argument, that is sum. %i also represents integers.

- scanf(): python’daki input() metodunun karşılığıdır. to get a value from the user we must first define a variable that will hold the value we get:

```
int age;
```

Then we call scanf() with 2 arguments: the format (type) of the variable, and the address of the variable:

```
scanf("%d", &age);
```

örnek:

```
#include <stdio.h>

int main() {

    int number;
    printf("please enter a number:");
    scanf("%d", &number);
}
```

yukarıdakinin python hali:

```
number = input("please enter a number:")
```

If we want to get a string as input, remember that a string name is a pointer to the first character, so you don't need the & character before it:

```
char name[20];
scanf("%s", name);
```

- Here's a little program that uses both printf() and scanf():

```
#include <stdio.h>
int main(void) {
    char name[20];
    printf("Enter your name: ");
    scanf("%s", name);
    printf("you entered %s", name);
}
```

-
-

STATIC VARIABLE

- Inside a function, you can initialize a static variable using the static keyword. I said "inside a function", because global variables are static by default, so there's no need to add the keyword. A static variable is initialized to 0 if no initial value is specified, and it retains the value across function calls.

```
int incrementAge() {
    int age = 0;
    age++;
    return age;
}
```

if we call incrementAge() once, we'll get 1 as the return value. If we call it more than once, we'll always get 1 back, because age is a local variable and it's re-initialized to 0 on every single function call. If we change the function to:

```
int incrementAge() {
    staticint age = 0;
    age++;
    return age;
}
```

now every time we call this function, we'll get an incremented value:

```
printf("%d\n", incrementAge());

printf("%d\n", incrementAge());

printf("%d\n", incrementAge()); >>> 1 2 3…
```

- We can also have static arrays. In this case, each single item in the array is initialized to 0:

```
int incrementAge() {
    static int ages[3];
    ages[0]++;
    return ages[0];
}
```

- The typedef keyword in C allows you to defined new types. Starting from the built-in C types, we can create our own types, using this syntax:

  typedef existingtype NEWTYPE

  The new type we create is usually, by convention, uppercase. This it to distinguish it more easily, and immediately recognize it as type. For example we can define a new NUMBER type that is an int : typedef int NUMBER

  and once you do so, you can define new NUMBER variables:

  NUMBER one = 1;

  Now you might ask: why? Why not just use the built-in type int instead? Well, typedef gets really useful when paired with two things: enumerated types and structures.

- Using the typedef and enum keywords we can define a type that can have either one value or another. It's one of the most important uses of the typedef keyword. This is the syntax of an enumerated type:

```
typedef enum { //...values } TYPENAME;
```

örnek:

```
typedef enum { monday, tuesday, wednesday, thursday, friday, saturday, sunday } WEEKDAY;
```

yani WEEKDAY isminde bir veri tipi tanımladık ve haftanın günlerini onun örnekleri olarak yazdık tıpkı 1, 20, 55 vs.nin integer veri tipinin örnekleri olması gibi

-

STRUCTURES

- In layman language, a structure is nothing but a cluster of variables that may be of different data types under the name name. Using the struct keyword we can create complex data structures using basic C types. A structure is a collection of values of different types. Arrays in C are limited to a type, so structures can prove to be very interesting in a lot of use cases. This is the syntax of a structure:

```
struct structname {
    //...variables
};
```

örnek:

```
struct date {
    int day;
    int month;
    int year;
};

// now you can declare variables of the type of struct date:

struct date today;
```

2nd way of creating structure is that you can both declare a variable and define a structure at the same time like:

```
struct date {
    int day;
    int month;
    int year;
} today ;
```

3rd way of creating structure is called unnamed structure which means you don’t have to give it a tag name:

```
struct {
    int day;
    int month;
```

```
    int year;
} today ;
```

fakat böyle yaparsan tek bir variable (today) tanımlayabilirsin.

- then you can give values with dot Operator:

```
today.day = 14;
today.year = 2022;
```

- a wholesome örnek:

```
int main()
{
    struct date
    {
        int day;
        int month;
        int year;
    };

    struct date today;

    today.day = 2;
    today.month = 4;
    today.year = 2022;

    printf("today's date is %i.%i.%i\n", today.day, today.month, today.year);
}
```

or you could just say:

```
struct date today = {2, 4, 2022};
```

or if you don't want the order:

```
struct date today = {.year = 2022, .month = 4, .day = 3};
```

- A structure is created outside the main function, preferably before the main function. In order to access the data members of the structure, variable(s) are created either inside or outside the main function depending upon the choice of the programmer. In structures we do something similar to OOP wherein after creating the class we create an object to use this class's attributes. Because without creating an object first we can't directly access the data members of a structure. We need to create at least one variable to access the structure. The variable is responsible for reserving a particular block of memory for the structure according to its size. There are 2 ways of creating this structure object:
    - inside the main function

```
#include <stdio.h>

struct employee {
    char name[50];
    int age;
```

```
    float salary;
};

int main() {

    // here we create an object within main() function
    struct employee employee1 = {"bugra kara", 20, 1500};
    printf("employee1's name is %s\n:", employee1.name);

}
```

- outside the main function

```
#include <stdio.h>

struct employee {

    char name[50];
    int age;
    float salary;

} employee1;
```

there are basically two ways to access data through a structure's object:

1. with the help of (.)

```
employee1.name

employee1.age

employee1.salary
```

2. with the help of (→)
3.

- örnek:

```
#include <stdio.h>
struct date
{
    // unsigned prefix means they start at 0
    unsigned int day, month, year;
};
    int main()
{

struct date d = {5, 2, 2022}; // Here d is an object of the structure time

    printf("Welcome to DataFlair tutorials!\n\n");
    printf("The Date is %d / %d / %d\n", d.day, d.month, d.year);
    return 0;
}
```

- örnek2:

```c
#include <stdio.h>

struct employee {
    char name[50];
    int age;
    float salary;
};

int main() {
    struct employee employee1 = {"bugra kara", 20, 1500};
    printf("an example of how structures are used in C\n");
    printf("employee1's name is: %s\n", employee1.name);
    printf("employee1's age is: %d\n", employee1.age);
    printf("employee1's salary is: %.2f\n", employee1.salary);

    return 0;
}
```

- It is important to note that in structures, only data members can be defined, they can't be initialized a specific value inside its definition. For instance you can't do this:

```c
#include<stdio.h>


struct person
{
    char name[30] = "bugra";
    int age = 25;

};
```

- Also, functions cannot be declared within a structure.
- nested structures

  structures embedded within each other are called nested structures. their normal syntax is:

```c
struct Marks
{
    float physics;
    float chemistry;
    float mathematics;
};

struct Student
{
    char name[20];
    int age;
```

```
    // Here m is a variable for the structure Marks
    Marks m;

    // Here s is a variable for the structure Student
}; Student s;
```

- you can also create structures in form of array:

```
struct date myDates[10];
```

- for instance to set the 3rd date in myDates array to 08.07.2001:

```
myDates[2].day = 8;
myDates[2].month = 7;
myDates[2].year = 2001;
```

ya da daha kısa yolu:

```
struct date myDates[5] = { [2] = {8, 7, 2001}};
```

or you can set multiple dates (elements):

```
struct date myDates[5] = { {8, 7, 2001}, {5, 3, 2011}, {9, 11, 2001}};
```

- nested structure örnek:

```
struct dateAndTime
{
    struct date sDate;
    struct time sTime;
};
```

değişkenler artık direkt dateAndTime olarak tanımlanabilir ayrı ayrı sDate ve sTime ile tanımlamana gerek kalmadı. mesela event isimli bir değişkenin sDate'inde month'ı 10 olarak ayarlamak istiyosak:

```
event.sDate.month = 10;
```

- you can define a variable to be a pointer to a struct:

```
struct date *datePtr;
```

- the variable datePtr can be assigned just like other pointers. you can set it to point to todaysDate with the assignment statement as in:

```
datePtr = &todaysDate;
```

- you should dereference datePtr pointer to get the value of what today's date is pointing:

```
(*datePtr).day = 21;
```

paranthese are required because otherwise things will get squiky with the precedence problem of dot

- → operator: x→y is the same as (*x).y

for instance

```
if ((*datePtr).month == 12) // is the same as
if (datePtr → month == 12)
```

- örnek:

```c
#include <stdlib.h>
#include <stdio.h>

// we created a structure named date
struct date
{
    int day;
    int month;
    int year;
};

int main(void)
{
    // created a normal variable from the structure date and a pointer
    struct date today, *datePtr;

    // assigned this pointer an adress
    datePtr = &today;

    // assigning values
    datePtr -> day = 22;
    datePtr -> month = 3;
    datePtr -> year = 2022;

    printf("today's date is %i/%i/%.2i.\n", (*datePtr).day, datePtr->month,
datePtr->year);
    return 0;
}
```

- şu iki kod aynı şeydir

```c
struct names {
    char firstName[20];
    char lastName[20];
};

// OR

struct pNames {
    char * firstName;
    char * lastName;
};
```

- how to pass a structure as an argument to a function?

```c
struct Family {

   char name[20];

   char surname[20];
```

```
    int age;
    char father[20];
};


bool siblings(struct Family sibling1, struct Family sibling2) {
    if (strcmp(sibling1.father, sibling2.father) == 0)
        return true;
    return false;
}
```

-
-

#command line parameters

- In your C programs, you might have the need to accept parameters from the command line when the command launches. For simple needs, all you need to do so is change the main() function signature from:

```
int main(void)
// to
int main (int argc, char *argv[])
```

argc is an integer number that contains the number of parameters that were provided in the command line. argv is an array of strings.

- ilk parametreyi yazdıran program:

```
#include <unistd.h>

int main(int ac, char **av)
{
    int i;
    i = 0;
    if (ac > 1)
    {
        while (av[1][i])
        {
            write(1, &av[1][i], 1);
            i++;
        }
    }
}
```

-
-

structures

- In layman language, a structure is nothing but a cluster of variables that may be of different data types under the name name. In programming terminology, a structure is a composite data type (derived from primitive data types such as int and float) that we use in order to define a collection of similar or different data types under one same name in a particular block of computer memory. Syntax of a structure is:

```
struct structure_name
{


data_type data_member1;
data_type data_member2;
data_type data_member3;


};
```

as in

```
// struct keyword + sctructure name
struct myStruct
{
    // members of structure
    char name[30];
    int roll-number;
    float marks;
};
```

- örnek:

```
#include <stdio.h>


struct date {


unsigned int day, month, year;


};


int main() {

struct date d = {12, 11, 2022}; // Here d is an object of the structure time

printf("This is Structures Example!\n\n");

printf("The Date is %d / %d / %d\n", d.day, d.month, d.year);
return 0;
}
```

- örnek2:

```c
#include <stdio.h>

struct employee {

char name[30];
int age;
float salary;

};

int main() {
struct employee e = {"Rachel", 29, 60000}; // Here e is an object of the structure employee

printf("The name is: %s\n",e.name);
printf("The age is: %d\n",e.age);
printf("The salary is: %0.2f\n",e.salary);

return 0;
}
```

- örnek3:

```c
 #include <stdio.h>

struct myCar {
    char brand[20];
    int year;
    float price;
};

int main() {
    struct myCar c = {"fiat", 2015, 150.400};

    printf("the brand of my car is: %s\n", c.brand);
    printf("the year of my car is: %d\n", c.year);
    printf("the price of my car is: %.3f\n", c.price);
} >>>

the brand of my car is: fiat

the year of my car is: 2015

the price of my car is: 150.400
```

- we can’t directly access the data members of a structure. We need to create at least one variable to access the structure. The variable is responsible for reserving a particular block of memory for the structure according to its size. There are 2 ways to create a variable to access data members in a structure:

1. inside the main function as in

```
struct book {


    char book_name[30];
    char author[30];
    int book_id;
    float price;


};

int main() {
struct book b; // Here b is a variable of structure book
}
```

2. outside the main function as in

```
struct book {
    char book_name[30];
    char author[30];
    int book_id;
    float price;
} b; // Here b is a variable of structure book.
```

- There are basically two ways in order to access the data members in a structure through variable(s):
    1. With the help of the member operator/ dot symbol (.)
    2. With the help of structure pointer operator (→)
- örnek:

```
#include<stdio.h>

struct book {
    char book_name[30];
    char author[30];
    int book_id;
    float price;
};

int main() {

struct book b; // Here b is a variable of structure book

printf("Enter the book name: ");
fgets(b.book_name, 30, stdin);
printf("Enter the author name: ");
fgets(b.author, 30, stdin);
printf("Enter the book ID: ");
```

```
scanf("%d",&b.book_id);
printf("Enter the book price: ");
scanf("%f",&b.price);

printf("\nThe details of the book are:\n\n");
printf("The book name is: ");
puts(b.book_name);
printf("The author name is: ");
puts(b.author);
printf("The book ID is: %d\n\n",b.book_id);
printf("The book price is: %0.2f\n",b.price);
return 0;
}
```

- It is important to note that in structures, only data members can be defined, they can't be initialized a specific value inside its definition. Also, functions cannot be declared within a structure. Yani struct blogunda sadece int var; diyebilirsin, value atamaya kalkarsan hata verir.
-

DEBUGGING

- debugging is the process of finding the errors in your program.
    1. logic errors
    2. compiler errors
    3. syntax errors
    4. memory corruption
    5. performance / scalability
- hatayla karşılaşırsan yapman gereken şeyler
    1. problemin ne olduğunu tam olarak kavramaya çalış
    2. reproduce the problem
    3. simplify the problem / divide and conquer
        - remove parts of your code to see if problem persists
        - comment out dubious sections of your code
        - turn a large problem into smaller programs (unit testing)
        - check your code step by step (10-15 lines of code)
    4. debugging
- a trace is generated whenever your program crashes fatally.
-
-

FILE INPUT&OUTPUT

-

HEADER

-
-
-
-

MUHTELİF

- fgets() metodu scanf() gibidir ama daha teferruatlı ve kullanışlıdır:
-
- dynamic memory allocation: There are 4 library functions provided by C defined under <stdlib.h> header file to facilitate dynamic memory allocation in C programming. They are:

    1. malloc()
    2. calloc()
    3. free()
    4. realloc()

1. malloc(): stands for memory allocation. It is used to allocate memory dynamically. It is used to allocate memory dynamically.

    ```
    void *malloc(size_t size)
    ```

    You reserve only as much memory as you need. If you need more later, You get some more from a place called heap. If you don't require a particular record anymore, you free that memory. So there is no memory wastage or shortage. malloc function is used to get memory from heap.

    ```
    ptr = (int*) malloc(100 * sizeof(int));
    Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And,
    the pointer ptr holds the address of the first byte in the allocated memory.
    ```

    if space is insufficient, allocation fails and returns a NULL pointer.

- örnek:

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

    int main()
    {
    char *str;
    ```

```
/* Initial memory allocation */
str = (char *) malloc(15);
strcpy(str, "Hello");
printf("String = %s, Address = %u\n", str, str);

/* Reallocating memory */
str = (char *) realloc(str, 25);
strcat(str, ".com");
printf("String = %s, Address = %u\n", str, str);

free(str);

return(0);
}
```

- örnek2:

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

int main()
{

   // This pointer will hold the
   // base address of the block created
   int* ptr;
   int n, i;

   // Get the number of elements for the array
   printf("Enter number of elements:");
   scanf("%d",&n);

   // Dynamically allocate memory using malloc()
   ptr = (int*)malloc(n * sizeof(int));

   // Check if the memory has been successfully
   // allocated by malloc or not
   if (ptr == NULL) {
      printf("Memory not allocated.\n");
      exit(0);
   }
   else {

      // Memory has been successfully allocated
      printf("Memory successfully allocated using malloc.\n");

      // Get the elements of the array
      for (i = 0; i < n; ++i) {
```

```
        ptr[i] = i + 1;
      }

      // Print the elements of the array
      printf("The elements of the array are: ");
      for (i = 0; i < n; ++i) {
         printf("%d, ", ptr[i]);
      }
   }

   return 0;
}
```

-

2. calloc(): stands for contiguous allocation and used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:

   a. It initializes each block with a default value ‘0’.

   b. It has two parameters or arguments as compare to malloc().

- its syntax is:

```
ptr = (cast-type*) calloc(n, element-size);
for example:
ptr = (float*) calloc(25, sizeof(float));
This statement allocates contiguous space in memory for 25 elements each with the size of the float.
```

- örnek:

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

int main()
{

   // This pointer will hold the
   // base address of the block created
   int* ptr;
   int n, i;

   // Get the number of elements for the array
   n = 5;
   printf("Enter number of elements: %d\n", n);

   // Dynamically allocate memory using calloc()
   ptr = (int*)calloc(n, sizeof(int));
```

```
   // Check if the memory has been successfully
   // allocated by calloc or not
   if (ptr == NULL) {
      printf("Memory not allocated.\n");
      exit(0);
   }
   else {

      // Memory has been successfully allocated
      printf("Memory successfully allocated using calloc.\n");

      // Get the elements of the array
      for (i = 0; i < n; ++i) {
         ptr[i] = i + 1;
      }

      // Print the elements of the array
      printf("The elements of the array are: ");
      for (i = 0; i < n; ++i) {
         printf("%d, ", ptr[i]);
      }
   }

   return 0;
}
```

3. free(): is used to dynamically de-allocate the memory for memory save.

```
free(ptr);
```

örnek:

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

int main()
{

   // This pointer will hold the
   // base address of the block created
   int *ptr, *ptr1;
   int n, i;

   // Get the number of elements for the array
   n = 5;
   printf("Enter number of elements: %d\n", n);

   // Dynamically allocate memory using malloc()
   ptr = (int*)malloc(n * sizeof(int));
```

```
    // Dynamically allocate memory using calloc()
    ptr1 = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL || ptr1 == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Free the memory
        free(ptr);
        printf("Malloc Memory successfully freed.\n");

        // Memory has been successfully allocated
        printf("\nMemory successfully allocated using calloc.\n");

        // Free the memory
        free(ptr1);
        printf("Calloc Memory successfully freed.\n");
    }

    return 0;
}
```

-

4. realloc(): used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory.

```
ptr = realloc(ptr, newSize);
```

örnek:

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;
```

```
// Get the number of elements for the array
n = 5;
printf("Enter number of elements: %d\n", n);

// Dynamically allocate memory using calloc()
ptr = (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL) {
   printf("Memory not allocated.\n");
   exit(0);
}
else {

   // Memory has been successfully allocated
   printf("Memory successfully allocated using calloc.\n");

   // Get the elements of the array
   for (i = 0; i < n; ++i) {
      ptr[i] = i + 1;
   }

   // Print the elements of the array
   printf("The elements of the array are: ");
   for (i = 0; i < n; ++i) {
      printf("%d, ", ptr[i]);
   }

   // Get the new size for the array
   n = 10;
   printf("\n\nEnter the new size of the array: %d\n", n);

   // Dynamically re-allocate memory using realloc()
   ptr = realloc(ptr, n * sizeof(int));

   // Memory has been successfully allocated
   printf("Memory successfully re-allocated using realloc.\n");

   // Get the new elements of the array
   for (i = 5; i < n; ++i) {
      ptr[i] = i + 1;
   }

   // Print the elements of the array
   printf("The elements of the array are: ");
   for (i = 0; i < n; ++i) {
      printf("%d, ", ptr[i]);
```

```
    }

    free(ptr);
  }

  return 0;
}
```

- programming languages split in 2:

  low level languages: Low-level languages provide abstraction from the hardware and are represented in the binary form i.e. 0 or 1 which are the machine instructions.

    machine level

    assembly level

  high level languages

- there are 5 types of programming:

  procedural programming languages: This programming paradigm, derived from structured programming specifies a series of well-structured procedures and steps to compose a program. It provides a set of commands by segregating the program into variables, functions, statements & conditional operators. Various Programming editors or IDEs help users develop programming code using one or more programming languages. Some of them are Adobe Dreamweaver, Eclipse or Microsoft visual studio, BASIC, C, Java, PASCAL, FORTRAN are examples of Procedural Programming Language.

  functional programming languages: A functional programming language is a declarative programming paradigm where programs are constructed by applying and composing functions. The language emphasizes expressions and declarations than on the execution of statements. The foundation of functional programming is lambda calculus which uses conditional expressions and recursion to perform the calculations. It does not support iteration like loop statements & conditional statements like if-else. Some of the most prominent functional programming languages are Haskell, SML, Scala, F#, ML, Scheme, and More.

  OOP languages: This programming paradigm is based on the “objects” i.e. it contains data in the form of fields and the code in the form of procedures. OOPs, offer many features like abstraction, encapsulation, polymorphism, inheritance, classes, and Objects. Encapsulation is the main principle as it ensures secure code. It also emphasizes code reusability with the concept of inheritance and polymorphism allows the spreading of current implementations without changing much of the code. Most multi-paradigm languages are OOPs languages such as Java, C++, C#, Python, Javascript, and more.

  scripting programming languages: All scripting languages are programming languages that do not require a compilation step and are rather interpreted. The instructions are written for a run time environment. The languages are majorly used in web applications, System administration, games applications, and multimedia. It is used to create plugins and extensions for existing applications. Some of the popular scripting languages are:

    Server Side Scripting Languages: Javascript, PHP, and PERL.

Client-Side Scripting Languages: Javascript, AJAX, Jquery

System Administration: Shell, PERL, Python

Linux Interface: BASH

Web Development: Ruby

logic programming: The programming paradigm is largely based on formal logic. The language does not tell the machine how to do something but employs restrictions on what it must consider doing. PROLOG, ASAP(Answer Set programming), and Datalog are major logic programming languages, rules are written in the form of classes.

sağdan sola doğru decimaller bir hane artar (1-10-100). alfabedeki her bir harfe karşılık 8 tane 0 ve 1'lerden oluşan bir kod vardır ve bilgisayarlar kod halinde gelen veriyi harfe çevirip sunarlar. mesela Hi mesajındaki H = 01001000 i = 01001001 dir. buna 8byte denir. bu kodların her bir rakamı 1 byttır.

1024 byte = 1 kb

1024 kb = 1 mb

1024 mb = 1 gb

1024 gb = 1 tb

her şey aslında rakamlardan oluşur. mesela gülen emoji unicode'da 128154'tir.

- pseudocode nedir:
- arguments ve parameters aynı anlama gelir
- things they look for:

  • 2+ years hands-on native iOS or Android development experience

  • Proficient with any of the Mobile Programming languages like Swift, Objective-C, Kotlin or Java.

  • Experience in Using Xcode or Android Studio

  • Experience with REST APIs

  • Object Oriented Design & Design Patterns knowledge,

  • Experience in delivering high-volume, high-availability and internet-facing applications

  • Strong optimization, debugging and technical documentation skill

  • Ability to develop functional mobile applications by modern architectures like VIPER and MVVM,

  • Working knowledge of the human interface guidelines, UX best practices, general mobile landscape, architectures, trends and emerging technologies

  • Working with Git version control system, including branching and merging strategies

  • Professional experience designing clean, maintainable codebase and writing tests

  • Strong knowledge of unit and automated testing for Mobile platforms

  • Have published and supported one or more apps in the App Store or Google Play Store

• B.Sc. in Computer Science or a similar discipline

• Fluent in English

* 1 year of experience in mobile Development, with at least 6 months experience in purely Flutter development

* Understanding and experience with CI/CD

* Experience with creating an SDK

Flutter framework'ünü orta seviyede kullanabilen

Mobil uygulama geliştirme alanında +1 yıl deneyimli

UI/UX konularında inisiyatif alabilen

IOS ve Android platformlar için markette en az bir uygulama geliştirmiş ve yayınlamış

Experience building object oriented web applications in Javascript, HTML5, SASS/LESS, and CSS3

Deep knowledge of ReactJS, Redux, Javascript

Proficiency in ECMAScript 6+

Experience with revision control systems such as Git

- There are essentially four major parts to any bachelor's level course of study, in any given field: pre-requisites, core requirements, concentration requirements and electives. Pre-requisites are what you need to know before you even begin. For many courses of study, there are no pre-requisites, and no specialized prior knowledge is required or presumed on the part of the student, since the introductory core requirements themselves provide students with the requisite knowledge and skills. Core requirements are courses that anyone in a given field is required to take, no matter what their specialization or specific areas of interest within the field may be. These sorts of classes provide a general base-level knowledge of the field that can then be built upon in the study of more advanced and specialized topics. Concentration requirements are classes that are required as part of a given concentration, focus or specialization within an overall curriculum. For example, all students who major in computer science at a given university may be required to take two general introductory courses in the field, but students who decide to concentrate on cryptography may be required to take more math classes, while students interested in electrical engineering may take required courses on robotics, while others interested in software development may be required to study programming methodologies and so on. Finally, electives are courses within the overall curriculum that individuals may decide to take at will, in accordance with their own particular interests. Some people may prefer to take electives which reenforce sub-fields related to their concentration, while others may elect to sign on for courses that may only be tangentially related to their concentration.
-

-
-
- five skills you must know
    - Git & GitHub
    - Databases (SQL)
    - Command-Line (Terminal, PowerShell etc.)
    - Unit Testing
    - learning multiple languages
    - excel
    - VSC basics
    - Jira
    - Vim or Emac

- how to dynamically allocate a string?