

# Programozás alapjai tutorial

*Készítette: Martinák Mátyás*

Miskolc, 2022

# Tartalomjegyzék

<b>1. VSCode telepítés, Extension-ök, C compiler telepítés</b>	<b>4</b>
<b>2. C program szerkezete</b>	<b>5</b>
2.1. Függvénykönyvtárak . . . . .	5
2.2. Main függvény . . . . .	5
2.3. Konstansok, változók . . . . .	7
2.4. Az első C programok . . . . .	7
2.4.1. Első számítási feladat . . . . .	7
2.4.2. Második számítási feladat . . . . .	8
2.5. Vezérlési szerkezetek . . . . .	9
2.5.1. Hányadosszámítás "if"-fel . . . . .	9
2.5.2. Egy bonyolultabb kalkulátor program "switch"-csele . . . . .	10
2.5.3. Ternary operator . . . . .	11
2.5.4. Ciklusok . . . . .	11
2.5.5. While ciklus . . . . .	11
2.5.6. Do-while ciklus . . . . .	12
2.5.7. For ciklus . . . . .	13
2.6. Típuskonverzió . . . . .	14
2.7. #include<math.h> . . . . .	15
2.8. Kommentek . . . . .	16
<b>3. Ellenőrzött beolvasás</b>	<b>17</b>
3.1. Intervallum ellenőrzött beolvasása . . . . .	18
3.2. Telefonszámla kalkulátor feladat ellenőrzött beolvasással . . . . .	19
<b>4. Véletlenszámgenerálás</b>	<b>22</b>
<b>5. Tömbök</b>	<b>23</b>
5.1. Tömbelemek kiírása fordítva . . . . .	24
5.2. Tömbelemek összegzése . . . . .	24
5.3. Tömbök feltöltése véletlenszámokkal . . . . .	25
5.3.1. Lottósorsolás feladat . . . . .	25
5.4. Karaktertömb . . . . .	26
5.4.1. Keresés az ABC-ben . . . . .	27
<b>6. Alapalgoritmusok</b>	<b>28</b>
6.1. Összegzés . . . . .	28
6.2. Számlálás . . . . .	29
6.3. Minimum kiválasztás . . . . .	29
6.4. Minimumhely kiválasztás . . . . .	30
6.5. Maximum kiválasztás . . . . .	30

6.6. Maximumhely kiválasztás . . . . .	31
6.7. Keresés . . . . .	32
6.7.1. Linear search . . . . .	32
6.7.2. Binary search . . . . .	32
6.8. Számcseré . . . . .	33
6.9. Prímszám . . . . .	33
6.10. Monotonitás . . . . .	34
6.11. Rendezés . . . . .	35
6.11.1. Minimum sort . . . . .	35
6.11.2. Maximum sort . . . . .	35
<b>7. Függvények</b>	<b>36</b>
7.1. Függvényparaméter . . . . .	37
7.2. Ellenőrzött beolvasás függvénnyel . . . . .	38
7.3. Tökéletes szám keresése függvényekkel . . . . .	39
7.4. Tömbkezelés függvényekkel . . . . .	40
7.5. Stringkezelés függvényekkel . . . . .	42
<b>8. Pointerezés</b>	<b>44</b>
<b>9. Struktúrák</b>	<b>45</b>
9.1. Struktúra elemére mutató pointer . . . . .	49
9.2. Dinamikus memóiafoglalás . . . . .	50
<b>10.Fájlkezelés</b>	<b>51</b>

### Előszó

Kedves Hallgatótársam! Ha a Programozás alapjai kurzusra iratkoztál fel ebben a félévben, akkor ez a jegyzet neked szól. Itt elsajátíthatod a gyakorlathoz szükséges tudást. **Figyelem!** A jegyzet nem készít fel maximálisan az elméleti vizsgára, csupán utat mutat. Ha fejezetenként haladsz és párhuzamosan gyakorlod a tanárnő által kiadott házi feladatokat, valamint vizsga előtt megtanulsz egy elméleti jegyzetet, akkor könnyen jó jegyet szerezhetsz a tárgyból. A jegyzet tartalmát elsősorban Fazekas Levente tanszéki mérnök weboldaláról illetve a 2021/22. I. féléves Programozás alapjai gyakorlatokon készített programokból állítottam össze, részben tartalmaz saját programokat is. Használd egészséggel. Jó tanulást kívánok!

## 1. fejezet

# VSCode telepítés, Extension-ök, C compiler telepítés

Először is szükség lesz a Visual Studio Code programra, azon belül is a C/C++ Extension Pack és a Code Runner csomagokra. Töltsük le! Ezen kívül az alábbi linken érhető el a C compiler:

**<https://www.msys2.org/>**

Telepítés után a **PATH** környezeti változóhoz a C fordító elérési útvonalát (default telepítés esetén: **C:/msys64/mingw64/bin**).

Nyissuk meg a MSYS2 parancssort és adjuk ki az alábbi utasításokat:

```
rm -r /etc/pacman.d/gnupg/
```

```
pacman-key --init
```

```
pacman-key --populate msys2
```

```
pacman-key --refresh
```

```
pacman -S --needed base-devel mingw-w64-x86_64-toolchain
```

Kreáljunk egy saját mappát a C: meghajtóra, amibe a projektjeinket fogjuk menteni. (Ajánlott minden héten új mappát létrehozni az egyes projektekhez.)

Indítsuk el a VSCode-ot, majd hozzuk létre az első '.c' kiterjesztésű fájlunkat, és nézzük meg, hogyan is néz ki egy C program szerkezete.

## 2. fejezet

# C program szerkezete

### 2.1. Függvénykönyvtárak

Ismerkedjünk meg a C programmal. Ahhoz, hogy a beépített ún. függvénykönyvtárakat tudjuk használni - amik lehetőséget adnak nekünk pl. a felhasználóval kommunikálni, vagy akár egyéb matematikai számítások alkalmazására - be kell őket "include-álni". A C nyelv erre a makrókat használja és # prefixszel jelöli.

**pl. (`#include<stdio.h>`)**

Minden ami #-gel kezdődik, az az előfordítónak szól, azaz a precompilernek. include = ami a kacsacsőr közt van, azt mi fogjuk használni. Két féle include létezik:

1. `<lib.h>` Mindig azokat a könyvtárakat használjuk, ami már előre definiált, egy külső könyvtár.
2. `"ulib.h"` Saját könyvtárakat fogunk behúzni ide.

Nekünk először a `<stdlib.h>` illetve az `<stdio.h>` függvénykönyvtárakra lesz szükségünk. Ezeket inkludáljuk be!

### 2.2. Main függvény

Minden program belépési pontja a "main" függvény. C-ben ennek a visszatérési értéke egy szám. Ha sikeres volt a program futása, akkor 0, ha sikertelen, akkor -1. Ezen kívül a programozó is tudja szabályozni, milyen "kódokkal" térjen vissza a program, például, ha az OS-nek szeretnénk üzeni valamit.

2.1. táblázat. Változótípusok C-ben:

int	Egész típus
float	Egyszeresen lebegőpontos
double	Kétszeresen lebegőpontos
char	Karakter
bool(belítható)	Logikai

Miket is írhatunk a main függvénybe? Egyáltalán mire jó nekünk a program? Elsősorban adatok feldolgozására. A programozási nyelvekben, az adatokat kétféle csoportban tárolhatjuk el. Léteznek konstansok, melyeknek az értékeit nem tudjuk megváltoztatni, és léteznek változók. C nyelvben az adatoknak négy ún. primitív típusa létezik. Az egész szám, az egyszeresen lebegőpontos szám, a kétszeresen lebegőpontos szám és a karakter. Minden programozási nyelvben szabályozva van az, hogy egyes típusok hány byte-ot képesek tárolni. Amennyiben "túlcsordulás" történik, azt a rendszer jelzi nekünk. Mindig a megfelelő típusú változót használjuk! Az alábbi táblázatban láthatjátok a C nyelvi kulcsszavakat:

Írjuk is meg első programunkat!

```
C HelloWorld.c > main()
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main()
5  {
6      printf("Hello World!");
7      return 0;
8  }
```

## 2.3. Konstansok, változók

Fontos tisztázni a változók illetve konstansok tulajdonságait. Hogyan hozzuk létre őket?

Példa alapján:

1. definiálás: **int**
2. deklarálás: **number**
3. inicializálás: **= 2;**

Láthatjuk, hogy így épül fel a változó, ebből a három elemből. Definiáláskor a típust adjuk meg, deklaráláskor a nevét, (változóknál kisbetűvel, konstansoknál konvenció szerint nagybetűvel van az egész név), inicializáláskor az értékét.

A konstansoknál a **const** kulcsóval adjuk meg, hogy a változó értéke nem megváltoztatható.

## 2.4. Az első C programok

### 2.4.1. Első számítási feladat

Írjunk egy kalkulátor programot, amelyik a négy alpművelet végrehajtására képes!

```
C kalkulator.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int a = 5, b = 5;
7      printf("Sum of A and B = %d\n", a + b);
8      printf("Subtraction of A and B = %d\n", a - b);
9      printf("Multiplication of A and B = %d\n", a * b);
10     printf("Division of A and B = %d", a / b);
11     return 0;
12 }
```



A **printf()** függvénnyel irathatjuk ki a szabványos kimenetre (konzolra) az adatainkat. A szöveget ""-ök közé kell írunk, illetve, ha számot is meg akarunk jeleníteni, akkor szükségünk van bizonyos formázásra is, amit a %-el adjuk meg, illetve az adott típushoz tartozó betű(k)vel. Ezek a következők:

- int: %d
- float: %f
- double: %lf
- char: %c
- string(hosszabb szöveg): %s

Használhatunk sortörést illetve tabulálást is a printf()-en belül.

Láthatjuk azt is, hogy az operátorok ugyan úgy működnek, mint bárhol máshol. (+,-,\*,/) Létezik maradékos osztás is, azt a (%) -el végezzük.

### 2.4.2. Második számítási feladat

Adjuk meg egy téglalap két oldalát, majd számítsuk ki a kerületét ill. területét!

```
int beolvas1, beolvas2;
printf("Adja meg egy teglalap ket oldalat es a program kiszamolja a teruletet!\n");
scanf("%d %d", &beolvas1, &beolvas2);
int kerulet = beolvas1 + beolvas2;
int terulet = beolvas1 * beolvas2;
printf("A teglalap kerulete: %d\n", kerulet);
printf("A teglalap terulte: %d\n", terulet);
```

Itt már a szabványos bemenetről kérjük a felhasználótól az inputot. Ezt a **scanf()** függvénnyel tehetjük meg. Itt is a bemeneti típus formázását ""-ök közt adjuk meg, illetve a második ún. függvényparaméterek azok a változók, amikbe olvasni akarjuk az értéket. Mivel ezeknek a változóknak a memóriacíme megváltozik a beolvasáskor (lásd később), ezért az & jelet tesszük eléjük, így adjuk be a függvénynek paraméterbe.

## 2.5. Vezérlési szerkezetek

Szekvencia, szelekció, iteráció. Ezeket kell megvalósítania egy programnak. De mik is ezek?

Amikor egy programkódot logikailag egymás alá tördelünk, minden lépésről tudjuk, hogy mit mikor csinál, az a szekvencia.

De van olyan, amikor nekünk valamilyen feltételhez kell kötnünk a program továbbfutását. Ilyenkor van szükségünk a szelekciókra, azaz elágazásokra. Kettő darab elágazásunk lesz, az if és a switch. Nézzünk is erre egy feladatot.

### 2.5.1. Hányadosszámítás "if"-fel

Adjunk meg két számot és a program kiszámolja a hányadosukat!

```
int beolvaska1, beolvaska2;
printf("Adjon meg ket szamot es a program kiszamolja a hanyadosukat!\n");
scanf("%d %d", &beolvaska1, &beolvaska2);
double hanyados;
hanyados = beolvaska1/beolvaska2;
printf("A ket szam hanyadosa:\n");
printf("%lf\n", hanyados);

if (hanyados > 5) {
    printf("A szam nagyobb, mint 5.");
} else {
    printf("A szam kisebb, mint 5.");
}
```

Látjuk a szintaktikát. Az "if"-en belül logikai kifejezést kell írunk, majd következik az a rész, ami akkor teljesül, ha a logikai kifejezés igaznak mutatkozik. Az "else" ágra az jut, ami nem igaz. Létezik még az "else if" kifejezés is, amibe szintén egy logikai kifejezést tudunk megadni és így tovább...

Így már el tudunk készíteni egy bonyolultabb kalkulátort is.

### 2.5.2. Egy bonyolultabb kalkulátor program "switch"-csel

```
int main()
{
    double a, b, result = 0;
    char op;
    bool ok;

    printf("Calculator\n");

    // Input
    do
    {
        printf(">");
        ok = true;
        // Check for right input format.
        if (scanf("%lf %c %lf", &a, &op, &b) != 3)
        {
            printf("Wrong input format!\n");
            // clear input buffer
            while (getchar() != '\n');
            ok = false;
        }

        // Check for division by 0.
        if (op == '/' && b == 0)
        {
            printf("Division by zero is not permitted!\n");
            // clear input buffer
            while (getchar() != '\n');
            ok = false;
        }

        // Caluclate
        switch (op)
        {
            case '+':
                result = a + b;
                break;
            case '-':
                result = a - b;
                break;
            case '*':
                result = a * b;
                break;
            case '/':
                result = a / b;
                break;
            default:
                printf("Operator %c not defined\n", op);
                return 1;
        }

        printf("%.4lf %c %.4lf = %.4lf\n", a, op, b, result);
    } while (ok);

    return 0;
}
```

Itt a switch vezérlési szerkezettel döntjük el, hogy mi legyen a program kimenetele. A case labeleket, mindig le kell zárni a break; utasítással, hogy ne folyjanak egybe.

### 2.5.3. Ternary operator

Egy elágazást leírhatunk egy sorban is akár, a "ternáris" operátor használatával. Ez a következőképpen néz ki:

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

int main()
{
    int num;
    printf("Give me a number: ");
    scanf("%d", &num);
    printf((num % 2) == 0 ? "even\n" : "odd\n");
    return 0;
}
```

Feladat: Döntsük el a számról, hogy páros vagy páratlan. Megnézzük a printf-en belül, hogy a szám osztható e 2-vel maradékosan és annak eredménye kiadja e a nullát. Ezután használjuk a ternáris operátort, és ami igaz az rögtön utána kerül, ami hamis az pedig a : után.

### 2.5.4. Ciklusok

Ha egy adott feladatot többször szeretnénk megismételni a program futása során, akkor ciklusra van szükségünk. Mik lehetnek ezek? Tömb feltöltése, olvasása, számok 1-N-ig való kiírása, ellenőrzött beolvasásnál egy feladat ismétlése, amíg a feltétel nem igaz... stb. Lássuk milyen ciklusaink vannak.

### 2.5.5. While ciklus

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

int main()
{
    int N, sum = 0;
    printf("Give me a number: ");
    scanf("%d", &N);
    int i = 1;
    while (i <= N)
    {
        sum += i;
        i++;
    }
    printf("Sum of numbers 1 to %d = %d\n", N, sum);
    return 0;
}
```

A program bekér a felhasználótól egy számot, majd 1-től a bekét számig összeadja a számokat. A **while** kulcsszóval adjuk meg, hogy ciklust definiálunk a programban, majd utána következik a feltétel, jelen esetben az, hogy 1-től a számunkig csinálja "ezt". Az "ez" pedig nem más mint a

`sum += i;` és az `i++`; Na ezek a kifejezések is magyarázatra szorulnak.

A `sum = sum + i;` kifejezésünk mit csinál? Fogja a `sum` nevű integer változónkat, ami majd az összeadásért fog felelni, és ehhez hozzáadja az `i`-t. Itt értékváltoztatás történik. Emlékezzünk vissza! A változók értékei megváltoztathatók, csak a konstansok értékei nem. Itt pont ez történik. Fontos kiemelni, hogy a vezérlési szerkezeteken belül, (tehát elágazásokon, ciklusokon belül) ha változót definiálunk, azt csak az adott blokkon belül tudjuk használni. Mit értünk blokk alatt? Mindent, ami a kapcsos zárójelen belül van. **A blokkon belüli változókat lokálisnak, a blokkon kívülieket globális változóknak nevezzük.** Ebben az esetben globális változókat kell használnunk. Vezérlési szerkezeteken belül pedig csak akkor definiálunk változót, ha azt az adott blokkon belül használjuk. Egyébként a program fordítási hibát dob. Visszatérve a `sum += i`-re. Ez egy összevont kifejezés, működik az összes többi operátorral is. Ezen kívül van még itt az `i++`, ami pedig mindig egyel növeli meg az aktuális változó értékét. Ellentétpárja az `i--`, ami pedig egyel csökkenti azt. A `while` egy előtesztelő ciklus.

### 2.5.6. Do–while ciklus

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int N, sum = 0;
7      printf("Give me a number: ");
8      scanf("%d", &N);
9      do
10     {
11         sum += N;
12         N--;
13     } while (N > 0);
14     printf("Sum of numbers 1 to %d = %d\n", N, sum);
15     return 0;
16 }
```

A do-while ciklust általában ellenőrzött beolvasásnál szoktuk használni. Hátutesztelő ciklus, ami azt jelenti, hogy előbb lefut a blokkon belüli kifejezés, majd kiértékelődik a feltétel. Ha hamis, akkor újra fut a blokkon belüli programrészlet. Ezért is használjuk általában ellenőrzött beolvasásnál, mert egyszer mindenképp le kell futtatni az adott programrészletet és utána nézzük meg, hogy igaz vagy hamis e a feltétel.

### 2.5.7. For ciklus

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int N, sum = 0;
7      printf("Give me a number: ");
8      scanf("%d", &N);
9      for (int i = 1; i <= N; i++)
10     {
11         sum += i;
12     }
13     printf("Sum of numbers 1 to %d = %d\n", N, sum);
14     return 0;
15 }
```

És elérkezünk ciklusaink végére, jöjjön a leghíresebb, a **for**.

Általában ezt használjuk, ez a legelterjedtebb. Egy ciklusváltozó segítségével végigjárjuk az elemeket, majd a blokkban már csak azt a részt kell leprogramoznunk, ami tényleg fontos. Jelen esetben összeadni a summot az *i*-vel. A *for*-on belül növelgetjük a ciklusváltozót, tehát azzal a blokkon belül már nem kell foglalkoznunk. A ciklusokkal későbbiekben még többet fogunk foglalkozni.

## 2.6. Típuskonverzió

C-ben két fajta típuskonverzió létezik.

- Implicit típuskonverzió - A fordító végzi. Eldönti, milyen típusúak lesznek az adott változók.
- Explicit típuskonverzió - A programozó végzi ún. "castolással".

Itt egy példa:

Azt szeretnénk, hogy egy integert több tizedes alakban írjon ki a programunk, akkor az értékkiírítás előtt zárójelben megadjuk azt, hogy mivel szeretnénk, hogy visszatérjen az adott változó.

```
int szam = 3;
double konvertalt = (double)szam;
printf("Szam = %lf ", konvertalt);
```

A kimenet 3.000000 lesz, mivel double értékke konvertáltuk a számunkat.

**Figyelem! Nem minden típuskonverzió működik oda-vissza.**

## 2.7. #include<math.h>

Egy fontos függvénykönyvtár. Mindenféle matematikai számítás elvégezhető vele. Lássuk a következő abszolútérték számító programot. Ezt a C egyetlen függvénnyel képes kiszámítani nekünk.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h> // Include math header!
4
5  int main()
6  {
7      int num;
8      printf("Give me a number: ");
9      scanf("%d", &num);
10     printf("Absolute value: %d\n", abs(num));
11     return 0;
12 }
```

A gyökvonás is működik.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<math.h>
4
5  int main()
6  {
7      printf("\nAdjon meg ket szamot es a program kiszamolja a szamtani es mertani kozepuket!\n");
8      double egy;
9      double ketto;
10     scanf("%lf,%lf", &egy, &ketto);
11     double szamtani = (egy+ketto)/2;
12     double mertani = sqrt(egy*ketto);
13     printf("A ket szam szamtani kozepe: %lf\n", szamtani);
14     printf("A ket szam mertani kozepe: %lf\n", mertani);
15     return 0;
16 }
```

Ezen kívül tud szinuszt, koszinuszt számolni, hatványra emelni, tehát egyszerűen rengeteg matematikai problémát tud kezelni ez a header állomány, az interneten az összes függvényét megtaláljátok.



## 2.8. Kommentek

Tudunk olyan szövegeket is hagyni a programunkban, amik nem jelennek meg a konzolon, hanem kvázi TODO-ként szolgálnak, vagy éppen magyarázatként. Ezek a kommentek. A legtöbb nyelvben a `//` illetve a `/* */` jelekkel lehet kommenteket hagyni. Míg az előbbi egysoros, az utóbbi többsoros kommenteket hagy a kódban.

## 3. fejezet

# Ellenőrzött beolvasás

Tanárnő kedvenc függvénye, avagy amit be kell tanulni.

```
C ellenbeolv.c > main()
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<stdbool.h>
4
5  int main()
6  {
7      int number;
8      bool ok;
9
10     do
11     {
12         ok = true;
13         printf("Give me a number: ");
14         if(scanf("%d", &number) != 1)
15         {
16             printf("Wrong input! Please retry!\n");
17             ok = false;
18             while(getchar() != '\n');
19         }
20     } while (!ok);
21
22     printf("Number = %d", number);
23
24     return 0;
25 }
```

Hogy megkönnyítsük a dolgunkat és ne 1-eseket meg 0-ásokat kelljen írni az igaz, hamis helyett, inkludáljuk be az `<stdbool.h>` headert. Ez egy új típust vezet be, a booleant, amit **bool**-ként fogunk definiálni. E mellé kell egy szám is, kezdetben most csak egész, amibe be akarunk olvasni. Indul a do-while ciklus, az ok boolean változót igazra állítjuk, mivel azt akarjuk, hogy addig fusson nekünk a ciklus, amíg az ok változó hamis. Bekérjük a számot, mivel megváltozik a memóriacím, elerakjuk az & jelet. Feltételként vizsgáljuk a bekért adatot. Hogyha nem szám, tehát nem egyezik meg 1-el, akkor kiíratjuk a konzolra, hogy hibás az input és az ok változót hamisra állítjuk, tehát ismétlődni fog a ciklus. Ezek után jön egy fontos lépés. Kiürítjük az input buffert. Erre azért van szükség, hogy a hibás adat kiürüljön a bufferből, amibe beleolvastunk, így újabb adatot tárolhatunk el benne ideiglenesen. Ha az adatunk megfelel a követelményeknek, akkor az a változóba kerül, ha nem, akkor újra kiürül az input bufferből. Ezeket jegyezzük meg. A félév során szükség lesz az ellenőrzött beolvasásra.

### 3.1. Intervallum ellenőrzött beolvasása

Egy számot egy adott intervallumon is be tudunk ellenőrzöten olvasni. Íme a példa:

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<stdbool.h>
4
5  int main()
6  {
7      double lower, upper;
8      bool ok;
9
10     do {
11         ok = true;
12         printf("\nGive me the two number of an interval: ");
13         scanf("%lf, %lf", &lower, &upper);
14         if (lower <= 1 || upper < lower) {
15             printf("\nWrong input! Please retry!\n");
16             ok = false;
17         } while ((getchar()) != '\n');
18     } while (!ok);
19
20     int darab = 0;
21     for (double i = lower; i <= upper; i++) {
22         for (double j = i + 1; j <= upper; j++) {
23             printf("%lf, %lf", i, j);
24             darab++;
25         }
26     }
27
28     printf("\nResult: %d\n", darab);
29     return 0;
30 }
```

A program megszámolja, hogy hány szám van az adott intervallumon.

### 3.2. Telefonszámla kalkulátor feladat ellenőrzött beolvasással

```
C phonebills.c > main()
1  ✓ #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4
5  ✓ enum type
6  {
7      IN = 1,
8      ON,
9      F
10 };
11
12 ✓ int main()
13 {
14     int n_calls;
15     double in_calls = 0, on_calls = 0, f_calls = 0, sum_mins = 0;
16     bool ok;
17
18     printf("Phone bill calculator\n");
19
20 ✓ do
21 {
22     ok = true;
23     printf("How many calls did you make?\n");
24 ✓ if (scanf("%d", &n_calls) != 1)
25 {
26     printf("Wrong input!");
27     while (getchar() != '\n');
28     ok = false;
29 }
30 } while (!ok || n_calls <= 0 || n_calls > 10);
31
32     printf("Number of calls made: %d\n", n_calls);
33 }
```

```
34 for (int i = 0; i < n_calls; i++)
35 {
36     enum type call_type;
37     double minutes;
38
39     // Call type input
40     do
41     {
42         ok = true;
43         printf("Type of call [1-foreign; 2-inner network; 3-out of network]\n");
44         if (scanf("%u", &call_type) != 1 || (call_type < 1 || call_type > 3))
45         {
46             printf("Bad input!\n");
47             while (getchar() != '\n');
48             ok = false;
49         }
50     } while (!ok);
51
52     // Minutes input
53     do
54     {
55         ok = 1;
56         printf("How many minutes did it take?\n");
57         if (scanf("%lf", &minutes) != 1 || minutes < 0)
58         {
59             printf("Bad input!\n");
60             while (getchar() != '\n');
61             ok = 0;
62         }
63     } while (!ok);
64
65     sum_mins += minutes;
66     switch (call_type)
67     {
68         // foreign
69         case F:
70             f_calls += minutes * 100.0;
71             break;
72         // inner-network
73         case IN:
74             in_calls += minutes * 40.0;
75             break;
76         // out of network
77         case ON:
78             on_calls += minutes * 60.0;
79             break;
80     }
81
82     printf("Your calls:\n");
83     printf("minutes\t\t%10.2lf\n", sum_mins);
84     printf("foreign\t\t%10.2lf$\n", f_calls);
85     printf("inter network\t%10.2lf$\n", in_calls);
86     printf("out of network\t%10.2lf$\n", on_calls);
87
88     return 0;
89 }
90 }
```

A feladat, hogy kiszámítsunk telefonszámlákat. Van hálózaton belüli, kívüli illetve külföldi tarifa. Ezeket egy ún. **enum**-ba tesszük bele. Az enum egy konténer. Adatokat képes tárolni, és az adatokra hivatkozni az enum nevén keresztül tudunk. Mi történik a mainen belül? Szükségünk lesz egy egész típusú változóra, amiben a hívásokat tároljuk el. Kellenek nekünk a tarifáinkhoz is külön változók, ezek double-ba kerülnek. Valamint szükség lesz egy összegváltozóra is. Ellenőrzötten beolvassuk a hívást, aminek a feltétele az, hogy szám legyen, ne legyen kisebb vagy egyenlő, mint 0, illetve 10-nél nagyobb sem. Végig iterálunk a számon. Létrehozzuk blokkon belül a hívás típusát, illetve a perceket. Ellenőrzötten be is olvassuk őket. A %u, az unsigned int-et jelöli, tehát az előjel nélküli egészet. Ebbe a változótípusban csak pozitív egész számokat tárolhatunk el, így sokkal több memóiahellyel rendelkezik. Ezek után összeadjuk a perceket, majd elágazunk. Megnézzük mi van a call type-ban, majd kiíratunk minden adatot.

A futás eredménye:

```
Phone bill calculator
How many calls did you make?
>3
Number of calls made: 3
Type of call [1-foreign; 2-inner network; 3-out of network]
3
How many minutes did it take?
43
Type of call [1-foreign; 2-inner network; 3-out of network]
2
How many minutes did it take?
12
Type of call [1-foreign; 2-inner network; 3-out of network]
1
How many minutes did it take?
33
Your calls:
minutes           88.00
foreign           4300.00$
inter network     1320.00$
out of network    720.00$
```

## 4. fejezet

# Véletlenszámgenerálás

Szintén egy fontos témakör, amit érdemes jól megjegyezni. Más programozási nyelvekben, pl. Java-ban, a véletlenszámgeneráláshoz van egy adott metódus, nincs szükségünk pl. időt lenullázni. Mind ezt azért kell, hogy kiszámítható eredményt kapjunk minden random szám generálásnál. A véletlenszámgenerálás képletét egyébként érdemes megtanulni kívülről, ugyanis OOP-ból is használni fogjuk.

Mindenek előtt inkludáljuk be a **time.h**-t, majd hívjuk meg az **srand(time(0))** függvényt. Ezek után vegyünk egy integer változót, majd töltsük fel véletlenszámmal. A képlet:

```
int number = (rand() \% (upper - lower + 1) + lower);
```

Azért kell egyet hozzáadni az alsóhatárhoz, mivel akkor azt a számot is belerakja a véletlenek közé.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<time.h>
4
5  int main()
6  {
7      srand(time(0));
8      int random = (rand() \% (100 - 11) + 10);
9      printf("%d", random);
10     return 0;
11 }
```

A program véletlenszámot generál 10 és 100 között.

## 5. fejezet

# Tömbök

Ha azonos típusú elemekből szeretnénk sokat eltárolni, akkor használjuk a tömböket. Egy tömb deklarálásakor mindig meg kell adnunk a méretet, különben a program error-t dob. Így célszerű rögtön egy konstans változóval rögzíteni azt.

```
const unsigned int SIZE = 10;
```

Ezt megváltoztatni már nem tudjuk a program futása közben, csak manuálisan. A tömb méretét egyébként a `#define` makróval is meg tudjuk adni. Ilyenkor a program header részében, az `include`-k alatt, felett kell használni a `#define` makrót. Egy tömb elemére indexelve hivatkozhatunk. Az indexelés 0-ától kezdődik.

```
int [] numbers = {3, 4, 5, 6};  
printf("\%d", numbers[0]);
```

A program kimenetele 3 lesz, mivel ez a szám található a 0. helyen.

Egy tömb elemeinek kiírására, értékadására általában `for` ciklust használunk, mivel ezzel az előltesztelő ciklussal a legkönnyebb dolgozni.

```
1  #include<stdio.h>  
2  #include<stdlib.h>  
3  #define N 10  
4  
5  int main()  
6  {  
7      int szamok[N];  
8      for (int i = 0; i < N; i++)  
9      {  
10         printf("%d. number: ", i + 1);  
11         scanf("%d", &szamok[i]);  
12     }  
13  
14     printf("\nNumbers:\n");  
15  
16     for (int i = 0; i < N; i++)  
17     {  
18         printf("%d. number: %d\n", (i + 1), szamok[i]);  
19     }  
20  
21     return 0;  
22 }
```



Itt cikluson belül töltjük fel számokkal a tömböt, amely számokat a felhasználótól kérünk be, egy másik ciklusban pedig ki is írjuk az eredményt.

### 5.1. Tömbelemek kiírása fordítva

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main()
5  {
6      double numbers[5] = {2.0, 4.0, 6.0, 8.0, 10.0};
7      int i = 4;
8      for (; i >= 0; i--) {
9          printf("\nd. number: %.11f\n", i + 1, numbers[i]);
10     }
11     return 0;
12 }
```

Láthatjuk, hogy nem okoz akkora fejtörést ez a művelet. Egyszerűen megfordítjuk a ciklust, hogy a ciklusváltozónk a **tömbméret - 1**-ről induljon egészen a 0-ás indexig csökkenő sorrendben haladva.

### 5.2. Tömbelemek összegzése

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main()
5  {
6      int array[5] = {2, 4, 6, 8, 10};
7      int sum, i;
8      sum = 0;
9      for (i = 0; i < 5; i++) {
10         sum += array[i];
11     }
12
13     printf("Sum = %d", sum);
14     return 0;
15 }
```

Itt a szokásos módon járunk el. Blokkon kívül inicializáljuk a sum változót, majd pedig a tömb-elemeket összeadjuk vele.

### 5.3. Tömbök feltöltése véletlenszámokkal

```

1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<time.h>
4  #define N 5
5
6  int main()
7  {
8      printf("\nLet's generate random numbers between 1 and 90!\n\n");
9      srand(time(0)); //<time.h>
10     int array[N];
11     for (int i = 0; i < N; i++) {
12         array[i] = rand() % 90 + 1;
13         printf("%d. number: %d\n", (i + 1), array[i]);
14     }
15     return 0;
16 }

```

A szokásos képlettel kigeneráljuk a random számokat, belerakva a tömbbe.

#### 5.3.1. Lottósorsolás feladat

Az alábbi lottósorsolási feladatot készítettem el:

```

1  #include<stdio.h>
2  #include<stdlib.h>
3  #define SIZE 10
4
5  int main()
6  {
7      printf("\n\n\tLottósorsolás! A program eloallítja az otos lotto heti nyeroszamaikat.\n\n");
8
9      int i, j;
10     int lotto[SIZE], jelolt;
11
12     srand(time(0)); //véletlenszám generátor inicializálása
13
14     for (i = 0; i < SIZE; ) {
15         jelolt = rand() % 90 + 1; //rand() % x : [0...x) intervallumból ad vissza egy egész számot
16         for (j = 0; j < i; j++) {
17             if (lotto[j] == jelolt) { //ismétlődés elkerülése
18                 break;
19             }
20         }
21         if (j == i) {
22             lotto[i] = jelolt;
23             i++;
24         }
25     }
26     //tömb kiírása
27     for (i = 0; i < SIZE; i++) {
28         printf("\n%d. szám: %d\n", (i + 1), lotto[i]);
29     }
30     return 0;
31 }

```

A kommentek tartalmaznak minden hasznos információt.

## 5.4. Karaktertömb

Egy tömb elemeinek száma lekérhető a **sizeof** kulcsszóval.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main()
5  {
6      char vowels[] = {'a', 'e', 'i', 'o', 'u'};
7      int size = sizeof(vowels);
8      printf("\n%c elemeinek szama: %d", vowels, size);
9      return 0;
10 }
```

Egy karaktertömb megfelel az Objektum Orientált nyelvekben használt string-nek.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main()
5  {
6      char string[10];
7      printf("Adjon meg egy szoveget: ");
8      scanf("%s", string);
9      printf("%s\n", string);
10     printf("\n");
11     return 0;
12 }
```

Láthatjuk, hogy a %s formázással stringet is be tudunk olvasni, illetve ki tudunk írni.

Ahhoz, hogy stringekkel tudjunk dolgozni, szükségünk lesz a **string.h** header állományra. Egy string méretét az **strlen(string)** függvény határozza meg. A **cctype.h** header-t beinkludálva pedig további műveletek végezhetőek a karaktereken, pl. **toupper(string)** vagy a **tolower(string)** függvények. Ezen kívül még sok sok függvényt tartalmaz a **string.h**, ilyen pl. string összehasonlító függvény, a **strcmp(str1, str2)** ami két függvény hasonlóságát vizsgálja tartalmuk szerint. Amennyiben hasonlít a tartalmuk igazat ad vissza, vagyis 1-et, ha pedig nem, akkor 0-át, azaz hamisat. A **string.h** tartalma is elérhető az interneten.

### 5.4.1. Keresés az ABC-ben

```

1  #include <ctype.h>
2  #include <math.h>
3  #include <stdbool.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  int main()
8  {
9      char abc[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
10                  'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r',
11                  's', 't', 'u', 'v', 'w', 'x', 'y', 'z'};
12      char key;
13      bool ok, found = false;
14
15      do
16      {
17          printf("Give me a character to search!\n");
18          ok = true;
19          if (scanf(" %c", &key) != 1)
20          {
21              printf("Bad input format!");
22              ok = false;
23              while (getchar() != '\n');
24          }
25      } while (!ok);
26
27      for (unsigned int i = 0; i < sizeof(abc) / sizeof(abc[0]); i++)
28      {
29          if (abc[i] == key || abc[i] == tolower(key))
30          {
31              printf("Your character is %u. in the ABC\n", i + 1);
32              found = true;
33              break;
34          }
35      }
36
37      if (!found)
38      {
39          printf("Character '%c' is not in the ABC!\n", key);
40      }
41
42      return 0;
43  }

```

Az ábrán látható karaktertömbbe beleírtuk az abc-t. Szeretnénk megkeresni egy betűt belőle. Nem kell mást tennünk, csak ellenőrzötten beolvasni a kulcsot, majd a keresési algoritmust (lásd hamarosan) használni.

## 6. fejezet

# Alapalgoritmusok

Keresés, egy tömb minimuma, maximuma. Ezek mind olyan feladatok egy programnak, melyeket algoritmusok nélkül nagyon nehezen tudnánk lefuttatni. Két alapalgoritmust már használtunk, egyet amikor összegeztünk a sum változóval, másodszor kerestünk karaktertömbben. Szerencsére ezeket az algoritmusokat nem nekünk kell megírni, nekünk csak be kell tanulni őket. Lássuk:

### 6.1. Összegzés

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main()
5  {
6      int N, sum = 0;
7      printf("Give me a number: ");
8      scanf("%d", &N);
9      for (int i = 1; i <= N; i++)
10     {
11         sum += i;
12     }
13     printf("Sum of numbers 1 to %d = %d\n", N, sum);
14     return 0;
15 }
```

## 6.2. Számlálás

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #define LENGTH 10
4
5  int main()
6  {
7      int array[LENGTH] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
8      int input = 4;
9      // Count of those smaller than the input is returned.
10
11     int count = 0;
12     for (int i = 0; i < LENGTH; i++)
13     {
14         if (input > array[i])
15         {
16             count++;
17         }
18     }
19     printf("Count: %d\n", count);
20     return 0;
21 }
```

## 6.3. Minimum kiválasztás

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #define size 5
4
5  int main()
6  {
7      const int arr[size] = {2, 4, 6, 8, 10};
8      int min = arr[0];
9      for (unsigned int i = 1; i < size; i++)
10     {
11         if (arr[i] < min)
12         {
13             min = arr[i];
14         }
15     }
16
17     printf("%d", min);
18     return 0;
19 }
```

## 6.4. Minimumhely kiválasztás

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #define size 5
4
5  int main()
6  {
7      const int arr[size] = {2, 4, 6, 8, 10};
8      unsigned int minindex = 0;
9      for (unsigned int i = 1; i < size; i++)
10     {
11         if (arr[i] < arr[minindex])
12         {
13             minindex = i;
14         }
15     }
16
17     printf("%d", minindex);
18     return 0;
19 }
```

## 6.5. Maximum kiválasztás

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #define size 5
4
5  int main()
6  {
7      const int arr[size] = {2, 4, 6, 8, 10};
8      int max = arr[0];
9      for (unsigned int i = 1; i < size; i++)
10     {
11         if (arr[i] > max)
12         {
13             max = arr[i];
14         }
15     }
16
17     printf("%d", max);
18     return 0;
19 }
```

## 6.6. Maximumhely kiválasztás

```
1  < #include<stdio.h>
2  < #include<stdlib.h>
3  < #define size 5
4
5  < int main()
6  {
7      const int arr[size] = {2, 4, 6, 8, 10};
8      unsigned int maxindex = 0;
9      for (unsigned int i = 1; i < size; i++)
10     {
11         if (arr[i] > arr[maxindex])
12         {
13             maxindex = i;
14         }
15     }
16
17     printf("%d", maxindex);
18     return 0;
19 }
```



## 6.7. Keresés

Két féle keresést különböztetünk meg. A lineáris keresést illetve a bináris keresést. Az utóbbi sokkal gyorsabban keres.

### 6.7.1. Linear search

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4
5 int main()
6 {
7     unsigned int array[] = {11260, 9850, 9125, 6630, 5890, 5385, 4650, 3030, 2860, 2665};
8     const unsigned int size = sizeof(array) / sizeof(array[0]);
9     const unsigned int search = 4650;
10
11     // Linear search.
12     // Worst case: O(n)
13     // Best case: O(1)
14     // Average case: O(n/2)
15     bool found = false;
16     for (unsigned int i = 0; i < size; i++)
17     {
18         if (search == array[i])
19         {
20             found = true;
21             printf("%u was found at index %u.\n", search, i);
22             break;
23         }
24     }
25     if (!found)
26     {
27         printf("%u is not in the array.\n", search);
28     }
29 }
```

### 6.7.2. Binary search

```
30 // Binary search.
31 // Worst case: O(log n)
32 // Best case: O(1)
33 // Average case: O(log n)
34 unsigned int first = 0;
35 unsigned int last = size - 1;
36 unsigned int middle = (first + last) / 2;
37
38 while (first <= last)
39 {
40     if (array[middle] < search)
41     {
42         last = middle - 1;
43     }
44     else if (array[middle] == search)
45     {
46         printf("%u was found at index %u.\n", search, middle);
47         break;
48     }
49     else
50     {
51         first = middle + 1;
52     }
53     middle = (first + last) / 2;
54 }
55 if (first > last)
56 {
57     printf("%u is not in the array.\n", search);
58 }
59
60 return 0;
61 }
```

## 6.8. Számcsere

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  void swap(int *x, int *y);
5
6  int main()
7  {
8      int a = 5, b = 10;
9      printf("Before swap(): a=%i, b=%i\n", a, b);
10     swap(&a, &b);
11     printf("After swap(): a=%i, b=%i\n", a, b);
12     return 0;
13 }
14
15 void swap(int *x, int *y)
16 {
17     int tmp = *x;
18     *x = *y;
19     *y = tmp;
20 }
```

## 6.9. Prímszám

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<stdbool.h>
4  #include<math.h>
5
6  bool is_prime(int input);
7
8  int main()
9  {
10     int number;
11     scanf("%d", &number);
12     printf("Is prime?: %s", is_prime(number) ? "yes" : "no");
13
14     return 0;
15 }
16
17 bool is_prime(int input)
18 {
19     // Basic algorithm for analysing whether the input number is prime or not
20     unsigned int prime = 2;
21     while (prime <= sqrt(input) && input % prime != 0)
22     {
23         prime++;
24     }
25     if (prime > sqrt(input))
26     {
27         return true;
28     }
29     return false;
30 }
```

## 6.10. Monotonitás

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<stdbool.h>
4  #define ARR_SIZE 6
5
6  int main()
7  {
8      const int array[ARR_SIZE] = {1, 2, 1, 6, 8, 12};
9
10     bool monotone = true;
11     for (int i = 1; i < ARR_SIZE; i++)
12     {
13         if (array[i - 1] > array[i])
14         {
15             monotone = false;
16             break;
17         }
18     }
19     printf("The array is %s.\n", monotone ? "monotone increasing" : "non monotone");
20 }
```

## 6.11. Rendezés

### 6.11.1. Minimum sort

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #define SIZE 5
4
5  int main()
6  {
7      int array[SIZE] = {2, 3, 55, 33, 23};
8      for (unsigned int i = 0; i < SIZE - 1; i++)
9      {
10         for (unsigned int j = i + 1; j < SIZE; j++)
11         {
12             if(array[i] > array[j])
13             {
14                 unsigned int tmp = array[i];
15                 array[i] = array[j];
16                 array[j] = tmp;
17             }
18         }
19     }
20     // Printing the result
21     for (unsigned int i = 0; i < SIZE; i++)
22     {
23         printf("%u ", array[i]);
24     }
25     return 0;
26 }
```

### 6.11.2. Maximum sort

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #define SIZE 5
4
5  int main()
6  {
7      int array[SIZE] = {2, 3, 55, 33, 23};
8      for (unsigned int i = 0; i < SIZE - 1; i++)
9      {
10         for (unsigned int j = i + 1; j < SIZE; j++)
11         {
12             if(array[i] < array[j])
13             {
14                 unsigned int tmp = array[i];
15                 array[i] = array[j];
16                 array[j] = tmp;
17             }
18         }
19     }
20     // Printing the result
21     for (unsigned int i = 0; i < SIZE; i++)
22     {
23         printf("%u ", array[i]);
24     }
25     return 0;
26 }
```

## 7. fejezet

# Függvények

Korábban már megismerkedtünk egy függvénnyel. Ez volt a program belépési pontja, a main, melynek nincs függvénydefiníciója és mindenképpen integer a visszatérési értéke.

Van egy fontos szabály a programozásban. Ne ismételjük magunkat! Erre is jók például a függvények. A függvények lényege tulajdonképpen, hogy ne a main-ben legyen minden összesűrítve "spagettikódként", hanem egyes programszakaszok külön legyenek particionálva. Íme egy példa:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4
5  unsigned int read_u();
6  unsigned int factorial(unsigned int num);
7
8  int main()
9  {
10     unsigned int num = read_u();
11     unsigned int fact = factorial(num);
12     if (fact > 0)
13     {
14         printf("Factorial of %u is %u\n", num, fact);
15     }
16     else
17     {
18         printf("%u is too large!\n", num);
19     }
20     return 0;
21 }
22
23 unsigned int read_u()
24 {
25     bool ok;
26     unsigned int num;
27     do
28     {
29         ok = true;
30         printf("Give me a positive integer: ");
31         if (scanf("%u", &num) != 1)
32         {
33             ok = false;
34             printf("Wrong input format!\n");
35             while (getchar() != '\n')
36             {
37             }
38         }
39     } while (!ok);
40     return num;
41 }
42
43 unsigned int factorial(unsigned int num)
44 {
45     unsigned int fact = 1;
46     for (unsigned int i = 2; i <= num; i++)
47     {
48         fact *= i;
49     }
50     return fact;
51 }
```

Az ábrán látható, hogy előbb definiálnunk kell a függvényeket. Erre a legjobb mód, ha a main előtt hozzuk létre a függvényprototípust, utána jöhet a deklarálás. Ezt célszerűen a main után tegyük meg. Egy függvénydefiníció a következőkből épül fel:

**visszatérés név(esetleges függvényparaméterek);**

A deklarálásban ugyan ezt a formulát kell felhasználnunk, ám itt már külön blokkja lesz a függvénynek, ide kerül a megvalósítás. A függvényeket a main-en belül hívhatjuk meg, vagy akár más függvényekben, de létezik olyan is, amikor egy függvény önmagát hívja meg, ezt rekurciónak nevezzük.

Na de mi is ez a visszatérési érték és mire jó? Ha egy függvénynek megmondjuk, hogy legyen visszatérési értéke (mert olyan is van, amikor nincsen), akkor ő a **return "típus"**; hívásra fog várni a blokk végén. Például az előző factorial függvény egy unsigned int-tel tér vissza. **Fontos, hogy a return funkcióra is ugyan azok a láthatósági szabályok érvényesek, mint a változókra, tehát általában blokkon kívül (ebben az esetben pedig csak a függvény blokkján belül) használjuk, kivéve, ha egy feltétel (pl. egy "if") megváltoztatja a függvény végkimenetelét.**

## 7.1. Függvényparaméter

Bármilyen típus lehet, primitív vagy pointer. Olyan paramétert kell megadni a függvénynek, amit fel is fogunk használni a belsejében, pl. a **factorial(unsigned int num)** függvénynek az a szám a paramétere, amelynek a faktoriálisát keressük.

## 7.2. Ellenőrzött beolvasás függvénnyel

```
16 void read_u(unsigned int *num)
17 {
18     bool ok;
19     do
20     {
21         ok = true;
22         printf("Give me a positive integer: ");
23         if (scanf("%u", num) != 1)
24         {
25             ok = false;
26             printf("Wrong input format!\n");
27             while (getchar() != '\n')
28             {
29             }
30         }
31     } while (!ok || *num < 1);
32 }
```

Ez a függvény **void** visszatérési értékkel rendelkezik, olyan, mint a `scanf()`, és nem csak olyan, de szinte teljesen ugyan az, annyi különbséggel, hogy nem adjuk meg ""-ök közt a szövegformázást, csak a változó memóriacímét. Ugyanis a mi függvényünk is pointert vár, akárcsak a `scanf()`. Ezek után már minden ugyan úgy működik, mint amikor a mainben írtuk meg az ellenőrzött beolvasást, ugyan az az algoritmus. Ha szeretnénk, hogy az adott számmal, vagy karakterrel térjen vissza a függvényünk, akkor nem kell megadni függvényparaméterként a számot, elég ha azt a függvényen belül hozzuk létre, és beolvasás után azzal térünk vissza. A mainben pedig az alábbi kód fog történni: **`unsigned int number = read_u();`** Esetleg paraméterben megadhatjuk azt az üzenetet is, amit a `printf`-be szeretnénk üzeni a felhasználónak: **`unsigned int number = read_u("Give me a number!");`**

### 7.3. Tökéletes szám keresése függvényekkel

Vegyük az alábbi feladatot:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4
5  void read_u(unsigned int *num);
6  bool is_perfect(unsigned int num);
7
8  int main()
9  {
10     unsigned int num;
11     read_u(&num);
12     printf("%u is %s\n", num, is_perfect(num) ? "perfect" : "not perfect");
13     return 0;
14 }
15
16 void read_u(unsigned int *num)
17 {
18     bool ok;
19     do
20     {
21         ok = true;
22         printf("Give me a positive integer: ");
23         if (scanf("%u", num) != 1)
24         {
25             ok = false;
26             printf("Wrong input format!\n");
27             while (getchar() != '\n')
28             {
29             }
30         }
31     } while (!ok || *num < 1);
32 }
33
34 bool is_perfect(unsigned int num)
35 {
36     unsigned int sum = 0;
37     for (unsigned int i = 1; i < num; i++)
38     {
39         if (num % i == 0)
40         {
41             sum += i;
42         }
43     }
44     return num == sum;
45 }
```

Külön függvényünk van az ellenőrzött beolvasásra, illetve külön függvényünk van arra, hogy megvizsgáljuk, hogy tökéletes e a szám. A második függvélynél érdemes megemlíteni, mivel bool-lal térünk vissza, itt logikai kifejezés is megadható a return utasítás után.



## 7.4. Tömbkezelés függvényekkel

A program az alábbi függvényeket fogja megvalósítani:

```

1  ✓ #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4
5  void read_farray(float *array, const size_t size);
6  void print_farray(const float *array, const size_t size);
7  bool is_monotone_increasing(const float *array, const size_t size);
8  float avgf(const float *array, const size_t size);
9  void print_avgf_diff(const float *array, const size_t size, const float avg);
10 size_t minindex(const float *array, const size_t size);
11 size_t maxindex(const float *array, const size_t size);
12

```

Beolvasunk a tömbbe, kiíratjuk a tömb elemeit a konzolra. Megvizsgáljuk, hogy monoton növekvő-e a sorozatunk, majd annak az eredményét is közöljük a felhasználóval. Ezen kívül átlagot számítunk, kiszámoljuk a különbséget a bekért elem és az átlag között, majd megkeressük a minimum és maximum helyét a tömbnek.

Megvalósítás:

```

13 int main()
14 {
15     const size_t array_size = 12;
16     float array[12];
17     read_farray(array, array_size);
18     print_farray(array, array_size);
19     if (is_monotone_increasing(array, array_size))
20     {
21         puts("The array is monotone increasing.");
22     }
23     else
24     {
25         puts("The array isn't monotone increasing.");
26     }
27     float avg = avgf(array, array_size);
28     printf("The average is %.2f\n", avg);
29     print_avgf_diff(array, array_size, avg);
30     size_t min_i = minindex(array, array_size);
31     printf("The index of the smallest element is %lu, with value %.2f.\n", min_i, array[min_i]);
32     size_t max_i = maxindex(array, array_size);
33     printf("The index of the largest element is %lu, with value %.2f.\n", max_i, array[max_i]);
34     return 0;
35 }
36
37 void read_farray(float *array, const size_t size)
38 {
39     bool ok;
40     for (size_t i = 0; i < size; i++)
41     {
42         do
43         {
44             ok = true;
45             printf("Element %lu: ", i + 1);
46             if (scanf("%f", &array[i]) != 1)
47             {
48                 printf("Bad input format!\n");
49                 ok = false;
50                 while (getchar() != '\n')
51                 {
52                     continue;
53                 }
54             } while (!ok);
55         } while (!ok);
56     }
57 }

```

```

58 void print_farray(const float *array, const size_t size)
59 {
60     printf("The array:\n");
61     for (size_t i = 0; i < size; i++)
62     {
63         printf("%.2f ", array[i]);
64     }
65     putchar('\n');
66 }
67
68 bool is_monotone_increasing(const float *array, const size_t size)
69 {
70     if (size <= 1)
71     {
72         return true;
73     }
74     for (size_t i = 1; i < size; i++)
75     {
76         if (array[i - 1] > array[i])
77         {
78             return false;
79         }
80     }
81     return true;
82 }
83
84 float avgf(const float *array, const size_t size)
85 {
86     float avg = 0;
87     for (size_t i = 0; i < size; i++)
88     {
89         avg += array[i];
90     }
91     return avg / (float)size;
92 }
93
94 void print_avgf_diff(const float *array, const size_t size, const float avg)
95 {
96     printf("Difference from average (%.2f):\n", avg);
97     printf("%s%12s%12s\n", "index", "value", "difference");
98     for (size_t i = 0; i < size; i++)
99     {
100         printf("%6lu%12.2f%12.2f\n", i, array[i], array[i] - avg);
101     }
102 }
103
104 size_t minindex(const float *array, const size_t size)
105 {
106     size_t minindex = 0;
107     for (size_t i = 1; i < size; i++)
108     {
109         if (array[i] < array[minindex])
110         {
111             minindex = i;
112         }
113     }
114     return minindex;
115 }
116
117 size_t maxindex(const float *array, const size_t size)
118 {
119     size_t maxindex = 0;
120     for (size_t i = 1; i < size; i++)
121     {
122         if (array[i] > array[maxindex])
123         {
124             maxindex = i;
125         }
126     }
127     return maxindex;
128 }

```

Itt is érvényesül a Top-down programszervezés módszere, miszerint egyes programfunkciókat részekre bontunk, így jobban átlátható a kód.

## 7.5. Stringkezelés függvényekkel

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <ctype.h>
5
6  #define STR_BUF (1 << 8) // 2 to the power of 8 (2^8 = 256)
7
8  void read_string(char *string);
9  void copy_reverse(const char *string, char *reverse);
10 void copy_capitalize(const char *string, char *capital);
11 unsigned int count_letter(const char *string, const char letter);
12

```

Definiáljuk az alábbi függvényeket:

- String beolvasása konzolról,
- String fordított kiírása,
- String nagybetűs kiírása,
- String betűinek megszámlálása

```

13 int main()
14 {
15     char string[STR_BUF];
16     read_string(string);
17     printf("%s\n", string);
18     puts(string); // same as above
19
20     char reverse[STR_BUF];
21     copy_reverse(string, reverse);
22     puts("Your thoughts backwards:");
23     puts(reverse);
24
25     char capital[STR_BUF];
26     copy_capitalize(string, capital);
27     puts("Your thoughts, but angry:");
28     puts(capital);
29
30     printf("Number of '%c' in your thoughts: %u\n", 'e', count_letter(string, 'e'));
31     return 0;
32 }
33
34 void read_string(char *string)
35 {
36     printf("Give me your thoughts: ");
37     // read everything until the '\n', aka new line character
38     scanf("%[^\n]", string);
39     // If we read a line, that is more than STR_BUF-1 characters long,
40     // we run into buffer overflow
41     // a safer version (when string is 256 long):
42     // scanf("%255[^\n]", string);
43 }
44
45 void copy_reverse(const char *string, char *reverse)
46 {
47     const size_t size = strlen(string);
48     for (size_t i = 0; i < size; i++)
49     {
50         reverse[i] = string[size - i - 1];
51     }
52     reverse[size] = '\0';
53 }
54
55 void copy_capitalize(const char *string, char *capital)
56 {
57     const size_t size = strlen(string);
58     for (size_t i = 0; i < size; i++)
59     {
60         capital[i] = (char)toupper(string[i]);
61     }
62 }
63

```

```
64 unsigned int count_letter(const char *string, const char letter)
65 {
66     unsigned int cnt = 0;
67     const size_t size = strlen(string);
68     for (size_t i = 0; i < size; i++)
69     {
70         if (tolower(string[i]) == tolower(letter))
71         {
72             cnt++;
73         }
74     }
75     return cnt;
76 }
```

A korábban tanult, beépített függvények használatával saját függvényeket kreálunk.

## 8. fejezet

# Pointerezés

A pointerek vagy mutatók, rámutatnak egy változó által befoglalt memóriaterületre. Miért is jó ez nekünk? Emlékezzünk vissza a számcsere algoritmusra, amikor megnéztük, hogy az adott változók hol helyezkednek el, s mivel megváltoztattuk az értéküket ezért a memóriacímük is megváltozik.

Pointert definiálni a következőképpen lehet:

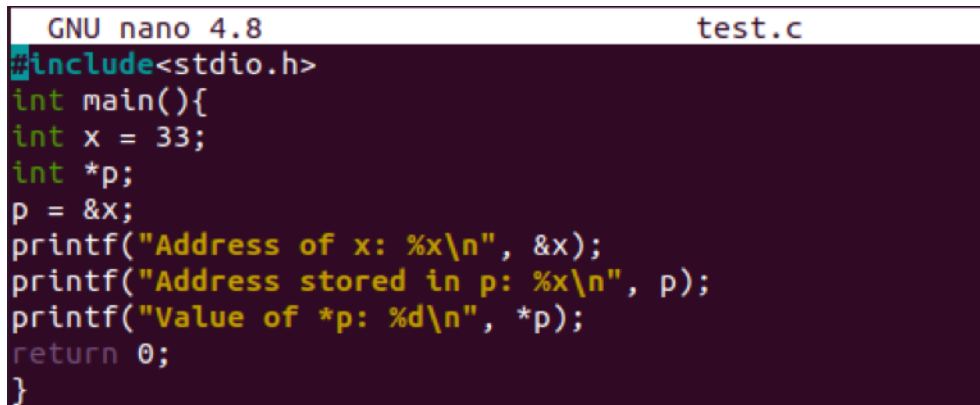
**típus\* pointer = &valtozo;**

Ahol a \* a pointert jelöli, a & pedig egy változó memóriacímét, amire a pointerünk mutatni fog.

**Jegyezzük meg: A \* az értéket jelöli, a pointer neve, pedig a memóriacímét.**

Egy másik példa az ellenőrzött beolvasás. Bekérjük a változó memóriacímét, mivel írni szeretnénk bele. Az eddig üres (null) változóba tartalom kerül, így a memóriacím is más lesz.

Létezik pointerre mutató pointer is, ilyenkor **típus\*\*** a jelölés.



```
GNU nano 4.8 test.c
#include<stdio.h>
int main(){
int x = 33;
int *p;
p = &x;
printf("Address of x: %x\n", &x);
printf("Address stored in p: %x\n", p);
printf("Value of *p: %d\n", *p);
return 0;
}
```

Itt **p**, az **x**-re mutató pointer(memory address). Helye hexadecimális alakban megadható. **&x**, **x** memóriacímét adja vissza. **\*p** pedig nem más, mint maga az **x(value)**. Így lehet feloldani a jelöléseket.

## 9. fejezet

# Struktúrák

Ha szeretnénk logikailag összetartozó elemeket egy blokkba helyezni, akkor struktúrát használhatunk. A struktúrát a **struct** kulcsszóval vezetjük be és névvel látjuk el. Bármely típus definiálható benne, és akár értéket is adhatunk meg, de legtöbbször a struktúrákat is függvényekkel töltjük fel.

```
14 typedef struct employee
15 {
16     unsigned int id;
17     char name[STR_BUFF];
18     Date birth_date;
19     double salary; // celery
20     unsigned int hours;
21 } Employee;
```

Az alábbi példában egy **typedef** kulcsszót is bevezetünk, hogy könnyebb legyen a mainen belül definiálni a struktúrát. Láthatjuk, hogy jelen struktúra egy munkavállaló adatait foglalja magába. A **Date** szintén egy struktúra típus.

```
7 typedef struct date
8 {
9     unsigned int year;
10    unsigned int month;
11    unsigned int day;
12 } Date;
```

Most már nincs más hátra, mint a mainen belül is definiálni a struktúránkat, majd pedig feltölteni értékekkel, különböző műveleteket végezni rajta.

```

23 unsigned int readu(const char *msg);
24 double readd(const char *msg);
25 void reads(const char *msg, char *str);
26 Employee read_emp(const char *msg);
27 Date read_date(const char *msg);
28 void print_emp(const Employee emp);
29 void print_date(const Date date);
30 void double_salary(Employee *emp);
31 double calc_hourly_rate(const Employee);
32 size_t max_hourly_rate(const Employee *emps, const size_t size);

```

Az alábbi függvényeket fogjuk használni:

- Ellenőrzött beolvasás unsigned int-re,
- Ellenőrzött beolvasás double-re,
- Ellenőrzött beolvasás stringre,
- Az Employee elemeinek beolvasása,
- A Date elemeinek beolvasása,
- Az Employee elemeinek kiírása,
- A Date elemeinek kiírása,
- A fizetések megduplázása,
- Órabér kiszámítása,
- A legtöbb órabér megkeresése

Lássuk a megvalósítást:

```

54  unsigned int readu(const char *msg)
55  {
56      bool ok;
57      unsigned int input;
58      do
59      {
60          ok = true;
61          puts(msg);
62          if (scanf("%u", &input) != 1)
63          {
64              printf("Bad input format!");
65              while (getchar() != '\n')
66                  ;
67              ok = false;
68          }
69      } while (!ok || input < 1);
70      return input;
71  }
72

```

```

73  double readd(const char *msg)
74  {
75      bool ok;
76      double input;
77      do
78      {
79          ok = true;
80          printf("%s\n", msg);
81          if (scanf("%lf", &input) != 1)
82          {
83              printf("Bad input format!");
84              while (getchar() != '\n')
85                  ;
86              ok = false;
87          }
88      } while (!ok || input < 1);
89      return input;
90  }

92  void reads(const char *msg, char *str)
93  {
94      puts(msg);
95      scanf(" %[^\\n]", str);
96  }

97
98  Employee read_emp(const char *msg)
99  {
100     Employee emp;
101     printf("%s\n", msg);
102     emp.id = readu("Give me the id!");
103     reads("Give me the name!", emp.name);
104     emp.birth_date = read_date("Give me the birth date!");
105     emp.hours = readu("Give me the weekly hours!");
106     emp.salary = readd("Give me the weekly salary!");
107     return emp;
108 }

110 Date read_date(const char *msg)
111 {
112     puts(msg);
113     Date date;
114     bool ok;
115     do
116     {
117         ok = true;
118         // 10 chars for date + 1 for '\\0'.
119         char buffer[11];
120         scanf(" %10[^\\n]", buffer);
121         if (sscanf(buffer, "%4u:%2u:%2u", &date.year, &date.month, &date.day) != 3)
122         {
123             printf("Bad date format! (YYYY:MM:DD)\\n");
124             ok = false;
125         }
126         else if (date.year < 1900 || date.month > 12 || date.month < 1 || date.day < 1 || date.day > 31)
127         {
128             printf("Bad date format! (YYYY:MM:DD)\\n");
129             ok = false;
130         }
131     } while (!ok);
132     return date;
133 }

```



```
135 void print_emp(const Employee emp)
136 {
137     printf("id: %u\n", emp.id);
138     printf("name: %s\n", emp.name);
139     printf("birth date: ");
140     print_date(emp.birth_date);
141     printf("salary: %.2lf$\n", emp.salary);
142     printf("weekly hours: %u\n", emp.hours);
143     printf("hourly rate: %.2lf$\n", calc_hourly_rate(emp));
144 }
145
146 void print_date(const Date date)
147 {
148     printf("%u:%u:%u\n", date.year, date.month, date.day);
149 }
150
151 void double_salary(Employee *emp)
152 {
153     emp->salary *= 2;
154 }
155
156 double calc_hourly_rate(const Employee emp)
157 {
158     return emp.salary / (double)emp.hours;
159 }
160
161 size_t max_hourly_rate(const Employee *emps, const size_t size)
162 {
163     size_t maxindex = 0;
164     for (size_t i = 1; i < size; i++)
165     {
166         if (calc_hourly_rate(emps[maxindex]) < calc_hourly_rate(emps[i]))
167         {
168             maxindex = i;
169         }
170     }
171     return maxindex;
172 }
```

És végül a main:

```

34  ✓ int main()
35  {
36      printf("Employeeeeeeeeeeeeeeeeeees!\n");
37      Employee emps[2];
38  ✓  for (unsigned int i = 0; i < 2; i++)
39      {
40          emps[i] = read_emp("Employee:");
41          double_salary(&emps[i]);
42      }
43
44  ✓  for (unsigned int i = 0; i < 2; i++)
45      {
46          print_emp(emps[i]);
47      }
48
49      printf("Employee with the highest hourly rate:\n");
50      print_emp(emps[max_hourly_rate(emps, 2)]);
51      return 0;
52  }

```

Láthatjuk, hogy nem bonyolult struktúrába olvasni, a dátum ellenőrzött beolvasása pedig kifejezetten hasznos.

## 9.1. Struktúra elemére mutató pointer

Van itt nekünk ez a függvény:

```

150
151  ✓ void double_salary(Employee *emp)
152  {
153      |   emp->salary *= 2;
154  }

```

Szeretnénk a kétszeresére növelni egy Employee fizetését. Ehhez hiába érjük el a függvényparaméteren keresztül az Employee egyik elemét, módosítani, memóriacímet megváltoztatni nem fogunk tudni pointer nélkül.

Erre találták ki a `->` operátort. Itt az **emp** változóból rámutatunk a **salary** elemre, amit meg szeretnénk szorozni 2-vel.

## 9.2. Dinamikus memória foglалás

Változóink eddig a statikus memóriaterületen (a Stacken) foglaltak helyet. Ha egy változónak értéket adunk, az a Stack-re kerül. Ennek elég kicsi a mérete. Annyira, hogy pár ezernél több számot nem is tudsz tárolni rajta. Ha nagy mennyiségű adatot akarsz tárolni, akkor a heapet szokás használni. Ez úgy működik, hogy csak pointert tárolsz a stacken, és a pointer mutat a heapre. Íme pár példa:

- `int szam=5;` //ez egy sima változó, a stacken tárolódik
- `int *p=&szam;` //ez egy pointer, de stacken levő adatra mutat
- `int *tomb= new int[32];` //ez egy másik pointer, de ez egy olyan tömbre mutat, amit a heap-en tárolunk.
- Heapre általában a `malloc()` függvényt használjuk.
- Fontos, hogy amit heap-en lefoglalsz memóriát, azt kötelességed neked felszabadítani. Erre való a `free()` függvény.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main()
5  {
6      // Malloc függvény
7      printf("\n");
8      printf("Let's see how does malloc function works!\n");
9
10     int* x = (int*) malloc(sizeof(int));
11     readi(x);
12     printf("The number: %d", *x);
13     free(x);
14
15     // Dinamikus tömb
16
17     printf("\n");
18     const int size = 10;
19     int* tomb = (int*) malloc(sizeof(int)*size);
20     for (unsigned int i = 0; i < size; i++)
21     {
22         read_array_i(&tomb[i]);
23     }
24     printf("The array:\n");
25     for (unsigned int i = 0; i < size; i++)
26     {
27         printf("%d ", tomb[i]);
28     }
29     free(tomb);
30     return 0;
31 }
```

## 10. fejezet

# Fájlkezelés

C-ben nagyon egyszerű a szövegfájlok kezelése. A szabványos bemenetet és kimenetet kezelő `scanf()` és `printf()` függvényeknek is van olyan változata, amelyik fájlból és fájlba dolgozik. Ezek az `fscanf()` és az `fprintf()`. Ezek első paraméterként megkapják a fájlt, amellyel dolgozniuk kell, amúgy pedig a használatuk teljesen megegyezik az előbb említett függvényekkel.

Fájlt megnyitni, létrehozni az `fopen()` nevű függvénnyel lehet. Ennek visszatérési értéke egy ún. file handle, amellyel a megnyitott fájlra hivatkozunk (mert egyszerre több fájlal is dolgozhatunk). A használat nagyon röviden az alábbi programban látszik. Ez a klasszikus „helló, világ” program, azzal a különbséggel, hogy nem a képernyőre, hanem a `hello_world.txt` fájlba írja az üzenetet.

```
1  #include <stdio.h>
2
3  int main(void) {
4      /* Az fp változóval hivatkozunk majd a nyitott fájlra. */
5      FILE* fp;
6
7      /* Létrehozzuk a fájlt, w = write = írni fogunk bele. */
8      fp = fopen("hello_world.txt", "w");
9
10     /* Beleírjuk a "Helló, világ!" szöveget. */
11     fprintf(fp, "Helló, világ!\n");
12
13     /* Végeztünk, bezárjuk a fájlt. */
14     fclose(fp);
15
16     return 0;
17 }
```

A megnyitás sikerességét egyébként ellenőrizni kell, mert előfordulhat, hogy nem sikerül valamilyen okból létrehozni a fájlt (pl. rossz helyen próbáljuk, nincs oda írási jogunk, és így tovább). A hibát úgy látjuk, hogy az `fopen()` függvény `NULL` értéket ad vissza. Ilyenkor a `perror()`-ral szokás hibaüzenetet kiírni, mert az egyből kiírja a sikertelenség okát is. És természetesen ilyenkor a fájlműveleteket (írás, zárás) nem végezhetjük el, mert nincs értelme.

```
1  #include <stdio.h>
2
3  int main(void) {
4      FILE* fp;
5      fp = fopen("szamok.txt", "w"); /* file-open, w = write */
6      if (fp != NULL) {
7          for (int i = 1; i <= 10; ++i)
8              fprintf(fp, "%d\n", i); /* file-printf */
9          fclose(fp);                /* file-close */
10     }
11     else {
12         perror("Nem sikerült megnyitni a fájlt");
13     }
14
15     return 0;
16 }
```

Az olvasás ugyanez; `w` helyett `r` (mert `read`), és `fprintf()` helyett `fscanf()`. Beolvasás közben a fájlból sorban kapjuk az adatokat, az elejétől végéig; mintha a fájl tartalmát a felhasználó folyamatosan gépelné be.

Fájl beolvasásánál gyakori az, hogy nem közvetlenül a fájlból `fscanf()`-elünk, hanem komplett sorokat olvasunk be, és utána a beolvasott sorokból, sztringekből vesszük ki az adatokat. Ez azért előnyös, mert így könnyebb kezelni a hibás fájlokat: tudjuk, hogy mekkora egységeket olvasunk be a fájlból, nem a sor közepén akad el hiba esetén a beolvasás. A beolvasott sor tartalma alapján pedig bonyolultabb esetszétválasztásokat is meg tudunk csinálni. A beolvasott soron akár `sscanf()`, `strtok()` vagy más sztringkezelő függvények is használhatók.