1. Információ, informatikai rendszer és adat fogalmak. Számítástechnika és informatika kapcsolata.

Az **információ** a valóság leképezése. Az **adat** olyan információ, melyet a számítógép tárol. **Informatikai rendszer** az adatok, információk kezelésére használt eszközök (pl. gépek, internet), eljárások (szabályok, programok), valamint az ezeket kezelő személyek együttese.

A számítástechnika alá tartoznak a feladatok, amiket számítógéppel, programokkal oldunk meg. Az informatika a számítástechnika üzleti alkalmazása.

2. Programtervezés lépései. Algoritmus, program, kódolás, szintaktika és szemantika. Programozási nyelvek típusai. Programozást támogató eszközök.

Program: Egy feladat megoldására szolgáló, a számítógép számára értelmezhető utasítássorozat.

Algoritmus: Egy feladatot általában többféle számítógépes programmal lehet megoldani. Azt az eljárást, képletet, vagy módszert, amelyet a program a feladat megoldásához használ, algoritmusnak nevezik. Az algoritmus véges számú lépésből áll.

Kódolás: Több dolgot is jelenthet, először is az információ formájának valamilyen átalakítását. Ekkor a kódolás magába foglalja az ún. forráskódolást, melynek célja az információ tömörítése, a titkosítást, melynek célja, hogy illetéktelen ne tudja visszafejteni az információt, és a csatorna kódolást, melynek célja a digitális információ hibátlan átvitele. Másik jelentése, hogy a kiválasztott nyelven megvalósítjuk a megtervezett algoritmusokat és adatszerkezeteket.

Programkészítés lépései

1. **Specifikáció**
    a. **Funkcionális követelmények**: A létrehozott algoritmusnak / programnak hogyan kell működnie? Minek kell megfelelnie annak a szoftvernek, amit elő akarok állítani? → logikai megfogalmazás (adott bemenetre működik, és adott kimenetnek kell lennie)
    b. **Nem funkcionális követelmények** (általános követelmények a programmal szemben): pl. hogy milyen gyorsan kell lefutnia (lefutási idő) / milyen szoftverkörnyezetben kell működnie (pl. Windows, Mac, stb.)?
2. **Implementáció**: tervezési lépés, specifikációnak megfelelő előállítás.
3. **Tesztelés (+hibajavítás)**: Jól működik-e a program?
    a. **Verifikáció**: A specifikációnak megfelel-e az adott implementáció?
    b. **Validáció**: Alkalmazható-e az adott program arra, amire készült, jól ellátja-e a feladatát, nincsenek-e hibák?

Visszalépések lehetnek: pl. ha implementálunk, lehet, hogy lesz, ami kihat a specifikációra, ezért azt változtatni kell, de alapvetően nincsenek nagy lépések oda-vissza.

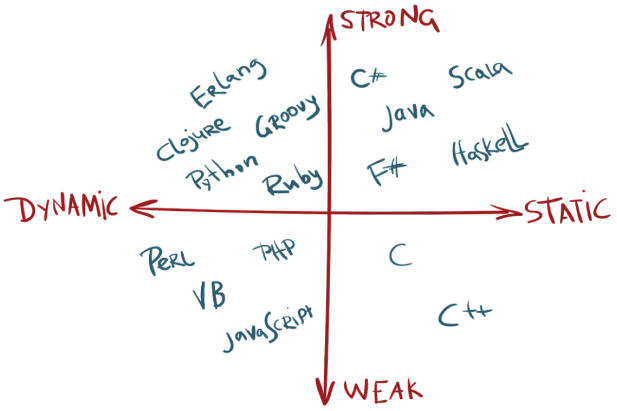
Szintaktika: Minden programozási nyelvnek van egy szintaktikája (minden programozási nyelvben mások a szabályok, amelyeket meg kell tanulni az adott nyelv használata érdekében): hogy hogyan kell pl. egy feltételt megfogalmazni, hogyan kell változót deklarálni, hova kerül pontosvessző, mik a kulcsszavak, stb. Ezeket a hibákat a fordító

mindenképp kijelzi és nem enged fordítani, de egy jó fejlesztői környezet a kód írása közben is aláhúzza a rossz részt. A szintaktika a program helyességének szükséges, de nem elégséges feltétele. Amennyiben a szintaktika jó, a program lefordul, de ez nem jelenti azt, hogy jól is működik.

Szemantika: Szemantikai hibának nevezzük, amikor a fordítóprogram nem talál hibát, mert a program formailag jó, csak nem azt csinálja, amit szeretnénk, hanem azt, amire utasítottuk. Ilyen például az, ha azt írjuk, hogy `if (a = b)`. Össze akartuk hasonlítani a-t és b-t, de ez igazából úgy lenne helyes, hogy `if (a == b)`. Amit írtunk, az annak felel meg, hogy a átveszi b értékét, és utána igazából egy `if (a)`, ami a C-ben azt jelenti, hogy ha a nem egyenlő 0-val, akkor igaz lesz. Ez nem az, amit le szerettünk volna írni.

Programozási nyelvek típusai

1. **Értelmezés szerint**
   a. **Fordított (compiled) nyelv**: A szöveges kódot a fordító átalakítja a gép által értelmezhető gépi kóddá, ilyen pl. a C vagy a C++. Nehezebb vele fejleszteni, könyvtárakra építeni, mindennek egységben kell működnie, cserébe nagyon gyors. A lefordított alkalmazást binárisnak (binary) is hívják. A fordítóprogram az, ami átalakítja a programot egy vele ekvivalens formára, de nem feltétlenül közvetlenül gépi kódra, lehet egy másik, alacsonyabb szintű nyelv is (például a C# fordító IL-re, azaz köztes nyelvre alakít).
   b. **Értelmezett (interpretált) nyelv**: Egy program valós időben “olvassa” és azonnal végre is hajtja a kódot, soronként haladva azon. Beolvassa a parancsot és meg is csinálja, rögtön látszik az eredmény. Ilyen pl. a Matlab vagy a Python. A program úgy is elindul, ha van benne hibás sor, majd ott megáll, és jelzi, hogy ez a sor értelmetlen. Könnyebb és kényelmesebb velük dolgozni, nincsenek túl erős megkötések, de olykor nagyon lassúak tudnak lenni.
   c. Összefoglalva: fordított nyelvnél az egész programot meg kell írni, ez lefordul, úgy futtatható, míg egy interpretált nyelv egyből futtatható, akár futás közben javítható. Fordított programok előnye még, hogy garantáltan nincs bennük szintaktikai hiba, mert a fordító nem engedi úgy a fordítást.
2. **Típus szerint**
   Az ábrának minden negyedéből elég egyet tudni, ez legyen a C, C#, Python, és a JS.



   Kétféle módszer szerint sorolhatók csoportba:
   a. **Statikus/dinamikus**

        i. **Statikusan típusos** egy nyelv, ha a változónak meg kell adni, milyen típusú adatot tárolhat (pl. `int`). Ettől eltérni nem lehet, ilyen a C.
        ii. **Dinamikusan típusos** egy nyelv, ha nem kell megadni konkrét típust egy változónak. Ettől még van típusa, például ha egy egész számot tartalmaz a változónk, az egy egész számként viselkedik, amikor használjuk éppen, de a tartalmát nyugodtan lecserélhetjük bármikor, mondjuk szövegre. Ilyen pl. a Python.
        iii. Összefoglalva: statikusan típusos nyelveknél a változónak van típusa, míg dinamikusan típusos nyelveknél az értékeknek van típusa.
    b. **Erős/gyenge**
        i. **Erősen típusos** egy nyelv, ha az értékek nem változnak váratlan módon, minden átalakítást explicit jelezni kell. Ilyen pl. a C#, ahol sokszor még pointeres varázslattal sem lehet a típustól eltérni, vagy a Python, ahol bár nincs a változóknak típusa, de minden típusnak megvan, hogy milyen műveletei lehetnek.
        ii. **Gyengén típusos** egy nyelv, ha megtörténhet benne, hogy a típusától eltérően kezelünk egy értéket, például JavaScript-ben szövegek összeadása egymás után fűzés (`"1" + "1" = "11"`), de a kivonásuk már számként kezeli őket (`"1" - "1" = 0`). A C is gyengén típusos, hiszen pl. kiírhatunk egy törtszámot `%d`-vel, ami az egészekhez való.
3. **Felhasználói közelség szerint**, azaz mennyire van elvonatkoztatva a gépi kódtól
    a. **Gépi kód/bináris**: Valójában nem nyelv, mivel a gép számára közvetlenül értelmezhető adatsort jelenti.
    b. **Alacsonyszintű (assembly) nyelvek**: A gépi kódhoz, vagyis a gép saját nyelvéhez legközelebb álló nyelvek. Mivel egy konkrét típusú architektúrához és esetleg operációs rendszerhez vannak kötve (pl. ARM procis Android), nem lehet őket hordozni vagy könnyedén lefordítani más rendszerre. Ezeken a legnehezebb programozni, hiszen az utasítások igazából a gépi kódok szöveges megfelelői. Előnye, hogy teljes hatalmunk van a gép felett, így maximális sebesség érhető el. Az assemblyből gépi kódot szintén egyfajta fordító (assembler) csinál, az cseréli le az utasításokat a nekik megfelelő számra. Emiatt fordított nyelv, de mégsem soroljuk annak, mert legtöbbször a fordított nyelvekből előbb assembly lesz, aztán gépi kód.
    c. **Magasszintű nyelvek**: Ember által könnyen olvasható nyelvek, használatukhoz nem szükséges az adott hardver ismerete, azt megoldja a fordító, nem kötődik a kód a hardverhez. Ilyen a C is. A processzor nem ért meg olyanokat, hogy `while` vagy hasonlók, azokat a fordító alakítja olyan formára, hogy megértse gépi kódból.

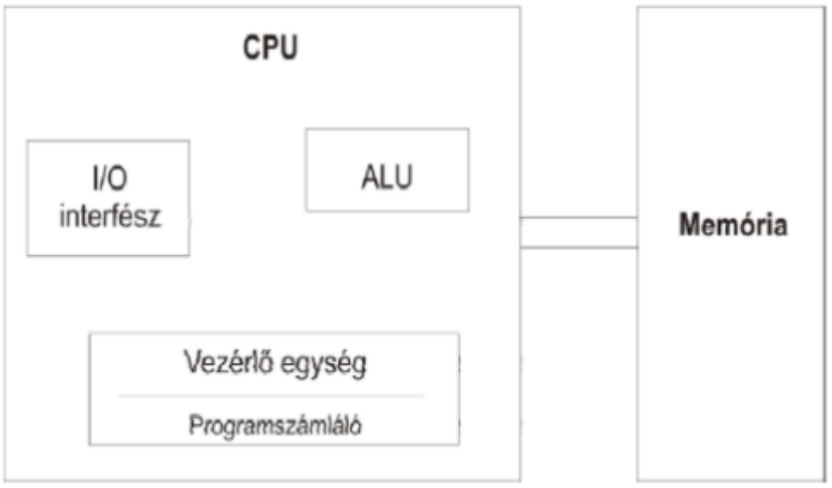
Programozást támogató eszközök

- **Szövegszerkesztő** (pl. Visual Studio Code (**nem a sima VS!**), Notepad++, jegyzettömb): írható benne kód, amihez a program ad megfelelő kiemelést, segíthet a kiegészítésben, illetve refaktorálhat (a kódot jobban érthetővé tevő átalakításokat csinál).
- **Fordító** (pl. GCC): a kódot kész programmá alakítja, közben optimalizálja és linkeli a fájlokat (megoldja az egymástól való függőségüket).

- **Debugger** (hibakereső): úgy futtatja a programot, hogy működés közben bele tudunk nyúlni. Megállíthatjuk bármelyik ponton, és ilyenkor megnézhetjük, hogy melyik változónak mi az értéke, illetve hogy milyen utat járt be a kód, hogy a megállítás pontjára eljusson (ezt hívják stack trace-nek, és azt mutatja, hogy melyik függvények hívták melyikeket).
- **Verziókezelő rendszer** (pl. Git): azt segíti elő, hogy naplózzuk a módosításokat, így bármikor vissza tudjunk állni korábbi állapotra, illetve láthatjuk, hogy mit módosítottunk. Elősegíti, hogy többen dolgozzanak ugyanazon a kódon, akár párhuzamosan, majd egyesítsék a módosításaikat. Mivel a kód nem a fejlesztő gépén van, biztonsági mentésként is szolgál.
- **Integrált fejlesztői környezet**: egy olyan programcsomag, amiben minden benne van a felsorolásból, például a Visual Studio.

3. Digitális számítógépek alapegységei, működési elvek. Központi egység, vezérlés, memória, perifériák. Aritmetikai egység. 2-es számrendszer, számok és karakterek ábrázolási módjai.

A számítógép működése szempontjából a 2 legfontosabb elem: a **CPU** (központi feldolgozó egység / Central Processing Unit, hétköznapi nyelven **processzor**) és az adatok tárolására alkalmas **memória**. Amit a számítógéphez kötünk (pl. egér, monitor, stb.), azt perifériának hívjuk.



A CPU három, különböző funkciót ellátó fő része:

1. **Regiszterek**
    a. Adattároló elemek a CPU-n belül: kevés adat tárolására alkalmasak, néhány bájt kapacitásúak.
    b. Tárolhat adatot, címet (később: pointert) és utasítást (gépi kódot).
2. **ALU (Arithmetic/Logic Unit = Aritmetikai/Logikai Egység)**

   a. Számítási (aritmetikai és logikai), illetve összehasonlító műveleteket hajt végre a tárolt adatokon. Aritmetikai műveletek: alapműveletek és maradékos osztás, logikai műveletek: összehasonlítások, egyenlőség, stb.
   b. Működése során a regiszterek tartalma változik a végrehajtott műveleteknek megfelelően.
3. **CU (Control Unit = Vezérlőegység)**
   a. Vezérli az utasítás-végrehajtást a CPU-n belül.
   b. A CPU regisztereiben az adatokkal végez műveleteket, mozgatja az adatokat, de jellemzően nem változtatja őket.
   c. Beolvassa a program utasításait és parancsokat ad az ALU-nak. A regiszterek majd ott változnak.
   d. Részei:
      i. **Memory Management Unit (Memória Vezérlő Egység)**: Az adatokat és utasításokat mozgatja a memória és a CPU között.
      ii. **I/O Interfész**: Külső eszközökkel beszélget, például egy billentyűzet és a CPU között teszi lehetővé a kommunikációt.

A **memória** a számítógépben az adatok és utasítások tárolására szolgáló elem. Mérete lényegesen befolyásolja a számítógép teljesítményét. Információ tárolására szolgáló rekeszeket tartalmaz. Minden rekesz egyedi címmel rendelkezik (az egyes rekeszekre ezzel a memóriacímmel hivatkozunk). A CPU mondja meg, hogy melyik rekeszben mit írunk/olvasunk. Felfogható úgy is a memória, mint egy nagyon hosszú táblázat, aminek minden cellájában 0 vagy 1 van:

0	0	0	1	1	1	1	0	1	0	1	1	1	0	0	1	1	1	1	0

Azért csak 0 és 1, mert a számítógépek kettes számrendszert használnak, és a kettes számrendszerben csak ez a két számjegy van. Azért kettes számrendszert használnak, és nem pl. tízest, mert így egyetlen kapcsoló elég a szám tárolásához. Ezeket a számokat hívjuk **bit**eknek. Mivel túl komplex áramkör lenne a memória, ha minden egyes bitnek memóriacíme lenne, ezért nyolcasával csoportosítjuk őket, ezt hívjuk **bájt**nak. Memóriacíme a bájtoknak van, és a memória méretét úgy adják meg, hogy ezekből mennyi van (pl. 8 gigabájt = 8 milliárd bájt).

Be	Ki
Igaz	Hamis
Igen	Nem
1	0

Két csoportra osztjuk a memóriákat:

1. Random Access Memory (**RAM**, véletlen elérésű / hozzáférésű memória)
   a. Tetszőleges pontján írható és olvasható memória, amit a programok fő munkaterületeként használunk.
   b. Nagyon gyors.
   c. Áram kell neki, hogy ne veszítse el a tartalmát, ezért kikapcsoláskor törlődik.
   d. Két típusa van:
      i. **Statikus RAM** (SRAM): olyan drága, hogy csak nagyon kevés (néhány MB) van belőle minden gépben, közvetlenül a CPU-ban, ahol gyorsítótár a szerepe. A gyorsítótár azt jelenti, hogy a memória helyett ideiglenesen itt dolgozik kisebb adatokkal, mert ez gyorsabb.

        ii. **Dinamikus RAM** (DRAM): ez az, amit a köznyelv memóriának hív, a többi memóriához képest olcsó, gyakran cserélhető elem. Jellemzően több GB méretű.
2. Read Only Memory (**ROM**, csak olvasható memória)
    a. Egyáltalán nem, vagy csak nagyon lassan írható.
    b. A program tárolására használjuk. Nem feltétlenül innen fut a program, lehet, hogy először átkerül a RAM-ba. Éppen ezért lehet ROM a merevlemez, egy SSD, de akár egy memóriakártya vagy CD is.
    c. Tartalma kikapcsolás után is megmarad.
    d. Típusai például:
        i. PROM (programozható ROM)
        ii. EPROM (törölhető és programozható ROM)
        iii. EEPROM (elektronikusan törölhető és programozható ROM)
        iv. flash memória, gyorsabb egy merevlemeznél is, cserébe drága

Perifériák típusai:

- Bemeneti: adatot ad a gépnek, pl. billentyűzet, egér, mikrofon…
- Kimeneti: a gép ad neki adatot, pl. monitor, hangszóró…
- Be- és kimeneti: mindkét irányba megy adat, pl. külső merevlemez, Wi-Fi…

2-es számrendszer

A számrendszer száma, hogy egy helyiértéken hányféle szám állhat. A 2-es számrendszerű számok (bináris számrendszer) egy számjegyét bit-nek nevezik, a binary digit (bináris számjegy) rövidítéseként. A számítógép a számokat és karaktereket binárisan ábrázolja (helyiértékes számábrázolás, 2 hatványain alapszik). Ahogy a mi számrendszerünkben vannak egyes, tízes, százas, … helyiértékek, a bináris ugyanez a logika, csak egyesek, kettesek, négyesek, nyolcasok, … vannak.

Áttérés binárisból 10-es számrendszerbe:

- Legyen a szám 1110, ekkor jobb oldalról (a legkisebb helyi értékről) indulunk, és összeadjuk, ahol 1-es van:
    - az egyesek helyén 0 van, tehát kihagyjuk
    - a kettesek helyén 1 van, tehát hozzáadunk 2-t
    - a négyesek helyén 1 van, tehát hozzáadunk 4-et
    - a nyolcasok helyén 1 van, tehát hozzáadunk 8-at
    - ezeket összeadva 14-et kapunk, vagyis 10-es számrendszerben ez a szám.

Próbáljuk meg visszafelé, mi a 14 binárisan:

- Addig osztjuk maradékosan a számot 2-vel, amíg nem lesz egyenlő 0-val, és most is jobbról balra haladunk:
    - $14 \div 2 = 7$, a maradék 0
    - $7 \div 2 = 3$, a maradék 1
    - $3 \div 2 = 1$, a maradék 1
    - $1 \div 2 = 0$, a maradék 1
    - mivel a maradékok jobbról balra állnak elő, ki is jött az 1110, ahol kezdtük.

Számok és karakterek ábrázolási módjai

Helyiértékes számábrázolás lehet:

- Decimális számrendszer: 10 hatványain alapszik
- Bináris számrendszer: 2 hatványain alapszik
- Oktális számrendszer: 8 hatványain alapszik
- Hexadecimális számrendszer: 16 hatványain alapszik
    - előnye, hogy benne egy bájt két számjegy
- Az informatikában általában binárist és hexadecimálist használunk

Számok ábrázolásához ezeket tárol(hat)juk:

- Bináris ábrázolás, előjel nélküli egész szám
- Előjel: az egyik bitet feláldozzuk, hogy azt is jelölhessük, ha egy szám negatív
- Lebegőpont: ha néhány biten azt tároljuk, hogy hány számjeggyel toljuk odébb a tizedespontot, akkor törteket vagy nagyon nagy számokat is tudunk tárolni

Karakterek ábrázolása:

- Általában egy karakter = egy bájt, és mivel egy bájt 8 bit, ezért $2^8 = 256$ különböző karaktert tudunk tárolni.
- Hogy melyik szám melyik karaktert jelenti, azt az ASCII tábla mondja meg.
- A rengeteg nyelv rengeteg karakteréhez különböző különleges kódolásokat vezettek be, például ilyen az Unicode. Azért volt erre szükség, mert az ASCII-ben nem fért el még az összes magyar betű sem, ezért fura néhány betűtípusban az ő és az ű.

4. Hardver és szoftver. Operációs rendszerek alapfeladatai. Operációs rendszerek típusai. Adjon példát egy operációs rendszerre, ismertesse főbb tulajdonságait.

Hardver: Elektronikus áramkörök, memória, be és kimeneti eszközök. Közvetlenül hajtja végre a gépi kódban írt programot.

Szoftver: Programok (gép által végrehajtható nyelven megfogalmazott algoritmusok) és a hozzá tartozó adatok.

Manapság azonban nagyon nehéz a megkülönböztetésük: (újra)programozható hardverek, minden szoftver megvalósítható elektronikus áramkörökben (FPGA), minden hardver szimulálható szoftver segítségével (pl. Android emulátor). A döntést az ár, a sebesség, megbízhatóság, fejlesztési idő, továbbfejlesztés bonyolultsága, stb dönti el.

A számítógép hardver elemeit az alábbi egyszerű módon lehet csoportosítani:

- Ki- vagy bemeneti eszköz (periféria)
- Vezérlőegység (CPU – Control Process Unit)
    - Aritmetikai-logikai egység (ALU)
    - Vezérlő egység (CU)
    - Interfész egység
- Elsődleges adattár (Memóriaegység)
- Másodlagos adattár (CD/DVD/lemez)

A szoftvereket két nagy csoportra oszthatjuk az általuk megvalósított szolgáltatások, funkciók alapján:

- **Operációs rendszer**: Egy olyan program, amely közvetlenül kezeli a hardvert, és egy egységes környezetet biztosít a számítógépen futtatandó alkalmazások számára. Alapvetően három részre bontható: a felhasználói felületre, az alacsony szintű segédprogramokra, és a kernelre (magra), amely közvetlenül a hardverrel áll kapcsolatban.
- **Alkalmazói szoftver**: A végfelhasználó számára tervezett alkalmazások, melyek 1-1 problémára vagy problémakörre nyújtanak megoldást.
- Néhány tankönyv a **fejlesztői szoftvereket** is külön kategóriának mondja.

Operációs rendszer: Intelligens (feladat-végrehajtásra alkalmas) gép a hardverre, mint erőforrásra támaszkodva. Számos olyan művelet van, amit sok program igényel, azonban nem érdemes hardveresen megvalósítani. Az operációs rendszer fő feladata, hogy biztosítsa ezeket a műveleteket a programok számára, miközben a hardvert hatékonyan kezeli, lényegében egységes környezetet biztosít a programok számára. A felhasználó és hardver közt helyezkedik el, olyan környezetet teremtve, amely lehetővé teszi a számítógép kényelmes használatát.

Az operációs rendszerek alapfeladatai az alábbiak:

- **Ütemezés**: Egy egyszerű számítógép egymás után hajtja végre a parancsokat. Hogy egy program miatt ne fagyjon le a rendszer, mert nem végez időben, minden program kap egy kis időt dolgozni, és az időszeletek kiosztását hívjuk ütemezésnek.

- **Megszakításkezelés**: Azt hívjuk megszakításnak, ha egy periféria olyan adatot küld, amit nagyon fontos kezelni, például mozgott az egér. Ekkor az oprendszer minden munkát félbehagy, kezeli a megszakítást, majd folytatja a dolgát.
- **Folyamatvezérlés**: Programok futási környezetének biztosítása, futtatás
- **Programok közötti kapcsolattartás**
- **Tárkezelés**
- **Működés nyilvántartás**: naplózás (mi okozott milyen hibát)
- **Kapcsolattartás a felhasználóval**
- **Szinkronizálás**: erőforrás igények sorba állítása
- **Memóriakezelés**
- **Perifériakezelés**

Az operációs rendszereket többféle szempont szerint is csoportokba foglalhatjuk, melyek:

- **Felhasználói felület szerint**
    - grafikus (Windows/Mac): egérrel vagy gombokkal navigálható, barátságosan elrendezett, akár animált felülete van
    - karakteres (Linux): a rendszerhez nem kell egér, különböző parancsokkal bírjuk feladatok megoldására
- **Folyamatkezelés szerint**:
    - kötegelt: utasítássorozatokat hajt végre, akár szövegfájlból, akár begépelve
    - multiprogramozott: egyszerre több programot futtat és tárol a memóriában
- **Kernel működése szerint**:
    - valós idejű (minden feladatot garantáltan és fix időn belül megold)
    - időosztásos (nem valós idejű)
- **Támogatott processzor architektúra szerint**: x86 (Windows), ARM (Android), PowerPC (régi Mac)
- **Jogállás szerint**: nyílt forráskódú (Linux) vagy szerzői jogvédelem alatt álló (Mac)

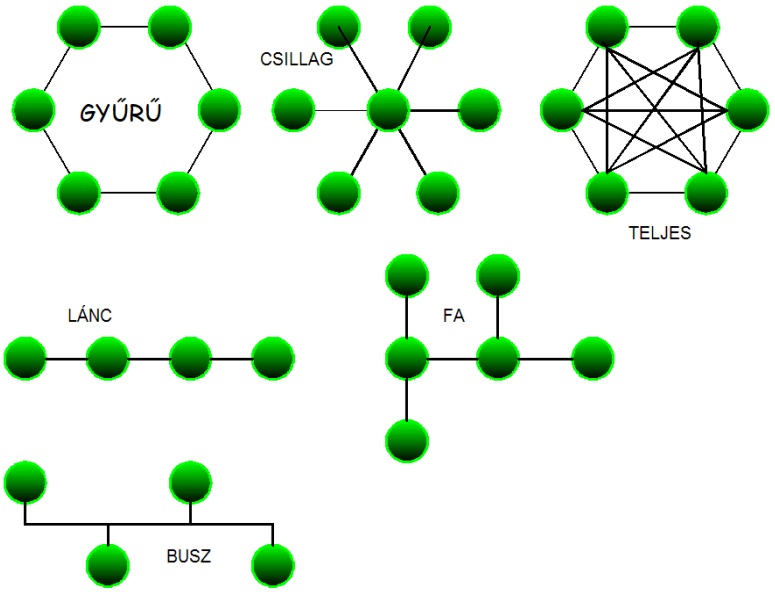
Egy példa bemutatása: A Windows egy grafikus, multiprogramozott, nem valós idejű operációs rendszer. Elsősorban x86 processzorokat támogat, de nyitnak az ARM felé is. Jogvédelem alatt áll, bár néhány rendszeralkalmazása (pl. számológép) és fő technológiája (.NET) nyílt forráskódú.

Ezzel ellentétben például a Debian egy Linux kernel alapú operációs rendszer, amely nyílt forráskódú, többfelhasználós, akár karakteres, akár grafikus felülettel is elérhető multiprogramozott, időosztásos rendszer, amely mind több architektúrán is elérhető. (Fontos megjegyezni, hogy az összes Linux alapú rendszer használható valós idejű folyamatkezeléssel is, viszont azt a hardvernek is támogatnia kell).

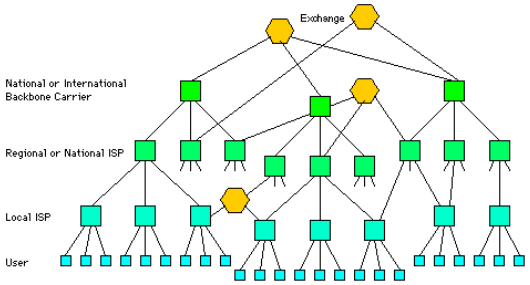
5. Hálózat fogalma, fontosabb tulajdonságai, topológiák, szolgáltatások, operációs rendszerek. Internet és fontosabb szolgáltatásai. Számítógépeket érő támadások problémái.

A **számítógép-hálózat** olyan speciális rendszer, amely a számítógépek egymás közötti kommunikációját biztosítja. A számítógép-hálózat lehet fix (kábeles) vagy ideiglenes (pl. betelefonálós). A vezeték nélküli internet általában vagy a cellás (mobil) szolgáltatásra, vagy a wifis megoldásra épül.

Topológiák:



- **gyűrű**: kört alkotnak, olcsó, egyetlen szakadást kibír
- **csillag**: mindenki egy központhoz van kötve (például egy házban minden a routerhez), de ha a csomópont kiesik, a hálózat leáll
- **teljes**: mindenki mindenkivel össze van kötve, a leginkább hibatűrő, de nagyon drága
- **lánc**: legolcsóbb, de egy hibát sem tűr
- **fa**: ilyen maga az internet is, az elemek közt szülő-gyermek viszony van:

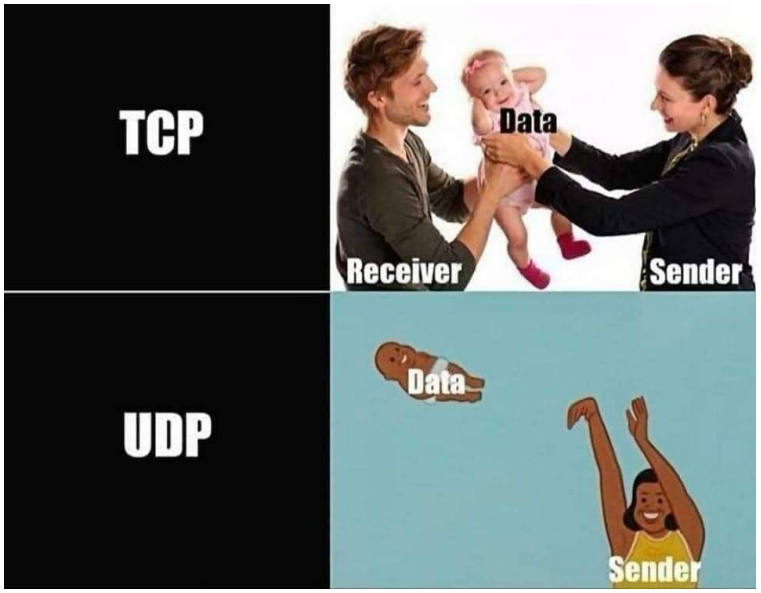


- **busz**: hasonló a lánchoz, de egy vezetéken van mindenki

Szolgáltatások:

- **IP**: Címeket rendel a hálózati szereplőkhöz, akik ezek által címzettként tudják megjelölni egymást - az interneten apró információkat tartalmazó csomagokat küldünk, a nagyobb adatokat több csomagra szedünk szét.

- **DNS**: Egy központi szolgáltatás, ami szöveges címeket (pl. www.google.com) fordít le IP-címekre, hogy felhasználóbarátabb módon találhassunk meg más számítógépeket, és ne számokat kelljen írogatni, ha el akarunk érni egy másik gépet.
- **TCP**: Egy szabványos módszer, amikor két gép megbeszéli, hogy hogyan kommunikálnak egymással, majd küldik az adatot - ez segít a csomagok sorrendjének megőrzésében (pl. nem keveredik össze a szöveg egy weboldalon), illetve abban is, ha egy adat nem érkezik meg, mert ekkor újraküldik.
- **UDP**: Olyan adatküldési forma, ahol nem számít, hogy a csomag megérkezik-e vagy hogy ez milyen sorrendben történik, a küldő elküldi, és nem érdekli más. Tipikusan játékok szokták használni, amikor fontosabb, hogy az adat gyorsan megérkezzen, mint az, hogy pontosan.



A **hálózati operációs rendszer** olyan szoftver, amely a szerveren fut, és lehetővé teszi a szervernek az adatok, felhasználók, csoportok, alkalmazások, a hálózati biztonság és egyéb hálózati funkciók kezelését. A hálózati operációs rendszert úgy tervezték, hogy engedélyezze fájlok megosztását, biztosítsa nyomtatók elérését több számítógép számára, általában egy helyi hálózaton (LAN) vagy magánhálózaton. Ilyen például a Windows Server operációs rendszer.

Internet és fontosabb szolgáltatásai

Az **internet** olyan globális számítógépes hálózat, amelyen a számítógépek az internetprotokoll (IP) segítségével kommunikálnak. Felhasználók milliárdjait kapcsolja össze és lehetővé teszi rengeteg elosztott rendszer működését:

- **Világháló** (World Wide Web, WWW, röviden web)**:** Milliónyi weboldalt tartalmaz, amiken rengeteg információ érhető el. Ezek közt a keresőmotorok (pl. Google, Bing, stb.) segíthetnek megtalálni, amit keresünk. A weboldalak elérhetőségét az URL-címek adják meg, és a DNS szolgáltatás alakítja ezeket a szöveges címeket IP-címekké, amik segítségével a gépek megtalálják egymást az interneten.
- **Elektronikus levelezés** (e-mail)**:** Szöveg, képek, és kisebb fájlok átvitelét segíti felhasználók között az interneten keresztül. Egy levelet többen is megkaphatnak, választ lehet úgy is írni, hogy mindenkihez elérjen, illetve titkos másolatot is lehet kiküldeni. Ha tartozik nyilvános webes felület a szolgáltatáshoz (pl. gmail.com), akkor

a világon bárhonnan elérhető. A **levelezési listák** olyan e-mail-re alapuló szolgáltatások, ahol egy központi címre beküldött levelet a lista minden regisztrált tagja megkap, és egyetlen címre írva lehet csoportos beszélgetéseket összehozni.
- **Instant üzenetküldés** (Instant Messaging, IM)**:** Ezek a chatszolgáltatások, mint a Facebook Messenger, WhatsApp, Viber...
- **Internetes telefon** (Voice over IP, VoIP)**:** Ez egy interneten keresztüli hanghívásos protokoll, de amúgy videotelefonálni is lehet. Ezek a hívások akár konferenciában is történhetnek.
- **Fájlátvitel** (File Transfer Protocol, FTP)**:** Távoli mappák böngészésére, az ottani fájlok kezelésére, oda feltöltésre, és onnan letöltésre van.
- **Fájlcsere:** Majdnem olyan, mint az FTP, de itt felhasználók egymástól töltik le a megosztott fájlokat. Ilyen technológia például a Torrent, amit leginkább illegális filmmegosztáshoz használnak.
- **Távoli asztal** (Remote Desktop Protocol, RDP)**:** Egy másik gépet tudsz távolról használni, a saját egereddel és billentyűzeteddel, az a gép pedig a saját képét és hangját küldi át.

Számítógépeket érő támadások problémái: A támadásoknak megannyi formája van:

- **Trójai vírusok**: Hasznos programnak, levélmellékletnek, vagy hasonlónak álcázzák magukat, és amint futtatjuk őket, bajt okoznak.
- **Férgek**: Saját magukat terjesztik más gépekre, hálózati biztonsági rést kihasználva.
- **Külső támadások**, például az elosztott szolgáltatásmegtagadásos támadás (distributed denial of service, DDoS): Ezt sokszor túlterheléses támadásnak is hívják, amikor több gép egyszerre kezd el valakit rengeteg üzenettel bombázni az interneten, aki ezzel nem tud mit kezdeni, ezért leáll.

Egy konkrét vírus vagy féreg a felhasználó gépén futó szoftver, és megannyi káros tevékenysége lehet: rögzítheti a billentyűleütéseket, összegyűjtheti a jelszavakat és banki adatokat, vagy akár elküldheti a mikrofon vagy webkamera felvételeit is. Kimondottan szerencsés esetnek mondható, ha egy vírus “csak” a helyben tárolt adatainkat teszi tönkre, főleg, ha van biztonsági mentés. Egy relatíve új dolog a zsarolóvírus, ami minden fájlt titkosít a gépen, és a feloldásukért pénzt követel. A fizetés nem ajánlott (simán lehet, hogy nem oldja fel a fájlokat vagy nem fertőz újra), csak az újratelepítés megoldás, illetve a megelőzés, gyakori biztonsági mentéssel.

Ami ellen legtöbbször alig tehetünk valamit, az a man-in-the-middle, vagyis középső emberes támadás. Ez azt jelenti, hogy próbálunk beszélgetni egy másik géppel, de a két gép közé beékeli magát egy kártékony harmadik fél: ő ekkor elolvashatja és módosíthatja az üzeneteket. Ha a számítógépünk titkosítatlan üzeneteket küld (például HTTP), sosem tudhatjuk meg, hogy ez történik, ezért is volt nagyon sokáig probléma, ha nyilvános internetet használtunk. Ma már elterjedt a HTTPS, ami bevezette a végponti titkosítást, vagyis a két fél olyan titkosított üzeneteket küld egymásnak, amit csak a fogadó tud feloldani, senki más nem.

Egy olyan támadás, ami bár számítógépekhez köthető, de nem informatikai, az a **phishing**. Így hívjuk, amikor egy e-mail vagy weboldal különböző trükkökkel (például azzal, hogy nagyon hivatalosnak tűnik) próbál személyes információkat, de akár pénzt vagy jelszavakat kicsalni. Nagyon sokszor a spamek célja ez, illetve az elgépelt webcímek is ilyen helyekre vezethetnek (például ha PayPal helyett PayPol-t írunk).

6. Alkalmazói és felhasználói programok osztályozása. Programozási nyelvek fordítói, segédprogramok, szimulációs programok. Office szolgáltatásai. C fordítók jellegzetes tulajdonságai. Hordozhatóság.

Szoftverek osztályozása:

- **Alkalmazói szoftverek**
  - Fájlkezelő alkalmazások (pl. Total Commander, de akár hálózatiak, pl. Google Drive), ide tartoznak a tömörítők is (pl. WinRAR)
  - Vírusvédelem (pl. Windows Defender, Norton, tűzfalak, stb.)
  - Médialejátszók (pl. VLC) és képnézegetők (pl. IrfanView)
- **Irodai szoftverek**
  - Szövegszerkesztők, pl. Word
  - Táblázatkezelők, pl. Excel
  - Prezentációkészítők, pl. PowerPoint
  - Adatbáziskezelők, pl. Access
- **Szervező szoftverek** (naptárak, teendők, cetlik, stb.)
- **Grafikus alkalmazások** (képszerkesztők, videóvágók, CAD/CAM programok. stb.)
- **Böngészők** (Chrome, Firefox, stb.)
- **Üzleti alkalmazások** (pl. SAP)
- **Játékok**
- **Fejlesztőeszközök** (kódszerkesztők, fordítók, debuggek)

Programozási nyelvek fordítói: az ember által olvasható kódból a processzor által értelmezhető gépi kódot, ezzel együtt az operációs rendszer által futtatható alkalmazást állítanak elő. A kód hibáinak egy kis részét felismerik, erről visszajelzést adnak, és többféle optimalizálásra (sebesség, méret) adnak lehetőséget.

A **segédprogram** lényegében egy szoftver, amely segít ellátni a felhasználó, vagy a rendszergazda egy konkrét számítógéppel kapcsolatos (gyakori esetben rendszerközeli) feladatait. Ilyen például a Gépház (rendszer beállítása), Eszközkezelő, Lemezkezelő, Töredezettségmentesítő, stb. Szinte minden segédprogram, amit a Start menüben a Windows mappákban találsz:

Szimulációs programok: Ez két dolgot jelenthet attól függően, hogy mire gondolt a költő. Egyik értelmezése, hogy fizikai szimulációkat futtat, például anyagtudományi kísérletekhez (pl. autó törésteszt), vagy biológiai kutatásokhoz (pl. Folding@home), akár az időjárás előrejelzéséhez, és megannyi más célhoz. A másik lehetőség áramkörök szimulációja, akár egyszerű villamosmérnöki feladatokhoz, de olyan komoly szimulációk is létezhetnek, amik egész másik hardvert (például egy Androidos eszközt vagy egy PlayStation-t) szimulálnak. Ezt a szimulációt nem szabad keverni az emulációval, ami ugyanazt jelenti (az Androidos példát), csak sokkal gyorsabb, mert okosan átalakítja úgy a szimulált rendszert, mintha a miénkre tervezték volna.

Office szolgáltatásai:

- **Word:** szövegszerkesztés
- **Excel:** táblázatkezelés
- **PowerPoint:** prezentációkészítés
- **Access:** adatbáziskezelés, azaz többféle táblából programozott lekérdezésekre használt szoftver
- **Outlook:** levelezés
- **OneNote:** jegyzetelés
- **Publisher:** kiadványszerkesztő, újságokhoz, magazinokhoz, stb.

C fordítók jellegzetes tulajdonságai:

A lépésein keresztül bemutatva:

1. Előfeldolgozás (nincs fájl kimenete)

Az előfeldolgozó (preprocesszor) a kód takarítását (megjegyzések, felesleges szóközök törlése, stb.) és a # kezdetű sorok feldolgozását végzi, konstansokat és makrókat helyettesít. Lényegében előkészíti a szöveget fordításra. Lehet neki utasításokat is adni, például olyat, hogy Windows-on egyik kód forduljon, míg Linuxon valami más, így szoktak egyszerre több rendszerrel kompatibilis C kódokat írni.

2. Fordító (.obj vagy .o fájlokat csinál, ezek gépi kódok)

A fordító a C nyelven megírt kódból a számítógép számára emészthető gépi kódot készít, melyet úgynevezett object fájlba ment. A gépi kód olyan elemi utasításokból álló sorozat, amely már kellően kicsi, egyszerű műveleteket tartalmaz ahhoz, hogy azt a számítógép processzora fel tudja dolgozni. Beállítható, hogy assembly nyelvű eredményt is produkáljon. Az assembly nyelv a gépi kód emberek számára jobban fogyasztható változata; az assembly kódban a processzor minden utasításához rendelnek egy jól megjegyezhető nevet (pl. mov = adatmozgatás, add = összeadás).

A fordítóban olyan dolgokat is beállíthatunk, hogy a kódot optimalizálja különböző sebességi szinteken (O1, O2, O3), esetleg arra, hogy minél kevesebb helyet foglaljon (Os). Azt is megmondhatjuk neki, hogy minden apróságra mutasson figyelmeztetést, de akár a figyelmeztetéseket is kezelhetünk hibaként, amitől már leáll a fordítás. Szintén beállítás kérdése, hogy a C nyelv melyik verziójának a szabályait és eszköztárát használja, például C99, C11, stb.

3. Linker (futtatható programot, pl. .exe fájlt csinál)

Egy projekt több .c fájt is tartalmazhat, mindegyikből létrejön egy-egy object fájl. A linker ezeket egyesíti, továbbá hozzászerkeszti a szabványos függvények (pl. `printf`) gépi kódját. Így létrejön a futtatható állomány (Windows-on .exe).

Hordozhatóság: Az a szoftver hordozható, amit egyszerűen átmásolhatunk egy másik gépre, és az telepítés nélkül működni fog. Egy nem hordozható program nagyon mélyen befészkeli magát a rendszerbe: több helyre menthet (és ha ezeket nem találja, nem biztos, hogy indul), több külön telepítendő programra épülhet (például játékok DirectX-re, de a Java is ilyen függőség lehet), a registry-be (Windows beállításjegyzék) is menthet, ezeket pedig nagyon nehéz összeszedni és másik gépre vinni. Ezzel szemben a hordozható szoftverek önmagukban is működőképesek (például mellé van téve a Java környezet, ha az kellene neki), és nagyon sokszor ugyanabba a mappába mentenek, ahonnan futnak, hogy az adatok hordozása is könnyű legyen. Kódból azt hívjuk hordozhatónak, ha bármelyik rendszeren változtatás nélkül le tudjuk fordítani.

7. C nyelv jellemzői. C nyelv egyszerű adattípusai. Példák adatok definiálására. Inicializálás.

A C nyelv jellemzői

- **Általános célú**: Sok eltérő célra használható, általános eszközöket kínál, szemben a speciális feladatok elvégzésére optimalizált programozási nyelvekkel (pl. R).
- **Rendszerprogramozási nyelv**: A C nyelv tervezésénél fontos szempont volt, hogy hatékony gépi kód generálódjon a fejlesztő által írt kódból, így a C-t gyakran használják rendszerprogramok (pl. operációs rendszer, illesztőprogram) írásához, így az alacsony szintű nyelvek (pl. assembly) kényelmes alternatívájává válhatott ezen a területen. Emellett persze egyéb alkalmazások is készíthetők vele. A C nyelv népszerűségének egyik oka éppen az, hogy mind alacsony, mind magas szintű elemeket tartalmaz.
- **Hordozható**: A C nyelv népszerűségének hála rengeteg különböző rendszerhez létezik fordító programja, ami miatt jól hordozható, sokféle rendszeren használható kódot írhatunk vele.
- **Strukturált**: A C nyelv biztosítja a strukturált programozáshoz szükséges eszközöket (pl. összetartozó utasítások külön fájlban, döntési szerkezetek, ciklusok). A strukturált programozás lényege, hogy a program által elvégzendő feladatot kisebb, egymáshoz jól meghatározott módon kapcsolódó részfeladatokra bontjuk, melyeket azután további alfeladatokra bonthatunk és így tovább egészen addig, míg a programozási nyelv által biztosított eszközökkel jól megoldható problémához nem jutunk.
- **Imperatív**: Nem a megoldandó problémát mondjuk meg a gépnek, hanem a megoldás módját, lépésről lépésre, ami lépéseket egymás után hajt végre.
- **Statikusan, gyengén típusos**: Minden változónak fix típusa van, amitől eltérni nem lehet (statikusan típusos), de különböző helyzetekben viselkedhetnek más típusként, például egy rosszul megírt `printf`-nél (gyengén típusos).
- **Fordított nyelv**: A fordítóprogram az ember számára olvasható kódból gépi kódot generál, ami nagyon gyors, cserébe csak olyan architektúrájú gépen fut, amire fordul.

C nyelv egyszerű adattípusai

Egyszerű adattípusnak mondjuk a legkisebb építőelemeket, amikből összetett szerkezeteket alkothatunk (például karakterekből szöveget, vagy két számból egy vektort). A program fontos részei a változók és az állandók (konstansok). A különböző utasítások ezeket kezelik és kombinálják. A változók adatok tárolására szolgálnak, minden változóhoz (és állandóhoz is) tartozik valamilyen típus. A típus meghatározza, hogy a változó milyen értékeket vehet fel és milyen műveletek végezhetők vele. A C nyelv alapvető adattípusai:

- `char`: A karakterkészlet egy eleme, mérete egyetlen byte. Példaérték: `'a'`.
- `int`: Egész szám. Mérete gépenként változó, jellemzően 2 (pl. Arduino-n) vagy 4 (pl. Windows-on) byte. Példaérték: `9`.
    - Különböző módosítókat lehet elé írni, hogy a méretét befolyásolják. Ekkor az `int`-et nem kötelező kiírni, tehát a `short int` és `short` ugyanazt jelenti.
    - `short`: Rövidebb, minimum 2 byte-os egész szám.
    - `long`: Hosszabb, minimum 4 byte-os egész szám.
    - `long long`: Garantáltan hosszabb, minimum 8 byte-os egész szám.

- `float`: Lebegőpontos szám. Azt jelenti, hogy a tizedespont helyét is tárolja, így tört értéke is lehet. Általában 4 byte, ennek egy része az előjel, másik kis része a tizedespont helye, a maradék pedig maga a szám. Egy példán szemléltetve:
    - Legyen a szám a 45.32.
    - Ekkor az előjel pozitív, tehát az első bit 0.
    - A mantissza (maga a szám) 4532.
    - A kitevő (tizedespont helye) -2, mivel 2 helyiértékkel csökkentettük a számot.
    - Ez a példa 10-es számrendszerben volt, a `float` ugyanezt 2-esben csinálja.
- `double`: Dupla pontosságú lebegőpontos szám, mindig kétszer akkora, mint a `float`.
    - Csak ennek a lebegőpontos számnak van módosítója, és az kizárólag a `long`. A `long double` a `double`-nél is kétszer nagyobb.
- Nem adattípus, de fontos megjegyezni a `void`-ot, ami a típusmentességet jelöli, például ha memóriát foglalunk, akkor `void` tömböt kapunk vissza, mert nem a foglaló függvény dönti el, hogy milyen típusú legyen, hanem a mi kódunk foglalás után. A `void` függvények azok, amik nem számolnak ki semmit, nem térnek vissza adattal, csak megoldanak egy feladatot.

Minden egész számnál eldönthetjük, hogy legyen-e előjele vagy sem, ezek a `signed` (legyen) és `unsigned` (ne legyen) módosítók, a típus elé kell őket írni. Alapból minden `int` előjeles, nem kell kiírni, hogy az legyen, `char`-nál pedig gépe válogatja, de asztali Windows-on az is előjeles. Az előjel az első bit, és ha kikapcsoljuk az előjelet, a szám hasznos része lesz, tehát kétszer akkora pozitív számot tud tárolni egy előjel nélküli típus, mint az előjeles. A `char` 1 bájt, vagyis 8 bit, ez azt jelenti, hogy $2^8 = 256$ darab különböző számot tud tárolni. Ha előjeles, akkor ezek -128-tól 127-ig tartanak, ha pedig előjel nélküli, akkor 0-tól 255-ig.

Adatok definiálása, inicializálás

A C nyelv statikusan típusos, ezért minden változót előre kell deklarálnunk, vagyis létrehoznunk. A változók mellett léteznek még állandók, avagy konstansok, ezek értékét nem lehet megváltoztatni, jelük a `const` módosító. Egy változót mindig úgy deklarálunk, hogy a típust a neve követi, például

```
int valtozo;
```

Ékezetes betűk nem érvényesek a változók nevében. Miután a változó létrejött, bármikor meg lehet adni neki új értéket:

```
valtozo = 5;
```

Ezt létrehozáskor is lehet, ekkor az ott megadott értéket kezdeti értéknek hívjuk. Amikor egy változót kezdeti értékkel hozunk létre, azt inicializálásnak hívjuk.

```
int valtozo = 5;
```

Egyszerre több változót is létre lehet hozni:

```
int valtozo1, valtozo2;
```

Egy változónév egy blokkon belül csak egyszer létezhet, így például két különböző függvénynek lehet ugyanolyan nevű változója, de egy függvényen belül egy név csak egyszer fordulhat elő.

Miután változóinkat deklaráltuk, már felhasználhatjuk őket a programunk későbbi részeiben. Ezt jellemzően a változók neveivel tehetjük meg. A változók célja, hogy bennük értékeket tárolhassunk. A változókhoz tehát értékeket rendelhetünk, ezt megtehetjük a deklarációval egyidőben vagy később is.

8. C nyelv kifejezés, utasítás fogalmai. Kiértékelés. Logikai adattípus.

Kifejezés: operátorokból, azonosítók, és zárójelek építik fel.

Utasítás: Egy feladatot meghatározó kifejezés, a programkód egy lépése. Amikor rákerül a vezérlés, következik a kiértékelés.

Kiértékelés: az utasításban megadott műveletek elvégzése megfelelő (precedenciájuk szerinti) sorrendben.

Az operátorok műveleteket végeznek el az azonosítókhoz rendelt értékekkel (operandusokkal), a zárójelek a műveletek sorrendjét képesek megváltoztatni. ha el akarunk térni a precedenciától (a kifejezésben szereplő műveletek elvégzésének szabályos sorrendjétől). A zárójellel közrefogott művelet előbb végződik el. Rengeteg operátor van:

- `+`, `-`, `*`, `/`: alapműveletek
- `%`: osztási maradék, például `7 % 3` eredménye 1 lesz
- `&&`: és kapcsolat - az eredmény akkor igaz, ha mindkét oldala igaz
- `||`: megengedő vagy kapcsolat - az eredmény akkor igaz, ha az egyik oldala vagy mindkét oldala igaz
- `<<`: balra shiftelés - kettes számrendszerben adott darab helyiértékkel balra lépteti a számot, például vegyük azt, hogy 5 << 1:
    - Az 5 binárisan 0101.
    - Ez egy számjeggyel balra léptetve 1010.
    - A 1010 értéke decimálisan 10.
    - 5 << 1 tehát egyenlő 10-zel, vagyis egy 2-vel való szorzás.
    - Alaposan belegondolva annyiszor szoroz 2-vel, amennyivel léptetünk.
- `>>`: jobbra shiftelés - ugyanaz, csak a másik irányba, annyiszor oszt 2-vel
- `++`, `--`: adott változó növelése vagy csökkentése 1-gyel
    - Ha a változó előtt van (pl. `++i`), akkor előbb növel, aztán használja, például:
      ```
      int x = 3;
      printf(“%d”, ++x);
      ```
      Ez azt írja ki, hogy 4.
    - Ha a változó után van (pl. `i++`), akkor előbb használja, aztán növel, például:
      ```
      int x = 3;
      printf(“%d”, x++);
      ```
      Ez azt írja ki, hogy 3.
- `=`: értékadás - felülírja, hogy mi van az adott változóban
- `==`: összehasonlítás - egyenlőséget vizsgál, és ha a két oldala egyenlő, igazat ad

- `<`, `>`, `<=`, `>=`: még több összehasonlítás - kisebb, nagyobb, stb., ezek is csak két értéket tudnak összehasonlítani, akár az `==` operátor. Olyan nincs, hogy `2 < 3 < 4`, az ilyeneket fel kell bontani úgy, hogy `2 < 3 && 3 < 4`.
- **Feltételes kiértékelés**: a `?:` operátor. Ha a `?` előtti rész igaz, a `:` előtti értéket választja, ha viszont hamis, akkor a `:` utánit. Egy példa:
  - ```
    if (a > b) {
        max = a;
    } else {
        max = b;
    }
    ```
  - Ugyanez rövidebben:
    ```
    max = a > b ? a : b;
    ```

Az alapműveletek és shiftelés rövidíthető, ha egy változót akarunk módosítani. Vegyük azt a példát, hogy `x` értékét 2-vel akarjuk növelni, ilyenkor `x = x + 2` helyett írhatunk olyat, hogy `x += 2`. Ugyanez érvényes a többire is, pl. `x -= 3`, `x *= 4`, `x /= 5`, `x <<= 8`, stb.

Szintén operátornak minősül az indexelés, vagyis a `[` és `]` karakterek, egy számmal köztük. Ezzel mondod meg, hogy egy tömb hányadik elemére szeretnél hivatkozni. A zárójelezés is operátor, ahogy a pont is (amivel egy struktúra tagjaira hivatkozol).

Utolsó operátor a `sizeof`, vagyis ami megmondja, hogy hány bájtos egy adott típus. Vegyük azt a példát, hogy `int x = sizeof(int)`, ilyenkor `x` értéke Windows-on 4 lesz, ahol egy `int` 4 bájtos. Ez majd a dinamikus memóriakezelésnél lesz hasznos, ahol például úgy foglalunk le 5 `int`-nek elég helyet, hogy `5 * sizeof(int)` bájtot kérünk a rendszertől.

C-ben igen gazdag operátorkészlet található, így a legtöbb dolog megoldható kifejezés utasítással. Például:

- `c = a + b;`: ez a kifejezés utasítás összeadja `a`-t és `b`-t, az összeget pedig `c`-be helyezi.
- `a - b;`: ez is egy kifejezés, bár nincs semmi értelme. Kiszámítja `a` és `b` különbségét, de az eredménnyel nem csinál semmit. A C-ben egy kifejezés értéke "eldobható".

Kiértékelés az, amikor egy kifejezés eredményét számítjuk ki, például a `8 + 2`-ből 10 lesz. A műveletek kiértékelésének sorrendje van, ahogy a matematikában is. A szorzás magasabbrendű, mint az összeadás, ezért először a szorzást számoljuk ki, majd az összeadást. A zárójel (szintén a matekkal összhangban) felülírja a műveleti sorrendet, tehát a `2 * (3 + 1)` kiértékelése 8 lesz. Ha egyenlő rangú műveletek maradtak, balról jobbra haladunk. A `8 + 2 + 4` először 10 + 4 lesz, majd végül 14. Fontos megjegyezni, hogy a precedenciák legutolsó helyén a vessző áll, még az értékadás is előbb következik nála. Ez lehetővé teszi például az ilyen tömör sorokat, vagyis a számolásokat a paraméterek helyén:

```
printf("A 3 ennyiedik hatványa: %f", powf(3, x));
```

Logikai adattípus

A számítástechnikában központi fogalom egy állítás **igaz** vagy **hamis** volta, ezt tárolja a logikai adattípus. Régen a C-ben nem volt ilyen, ezért az igazságértékek kiértékelés után egy `int`-ként álltak elő, 1 jelentette az igazat, 0 a hamisat. Ha például azt írtuk, hogy
```

```
int igaz = 3 < 5;
```

akkor ebbe a változóba egy 1-es került.

Komplex igazságértékeket is előállíthatunk, akár zárójelezhetünk is, tetszőlegesen sok feltételt megadva, például:

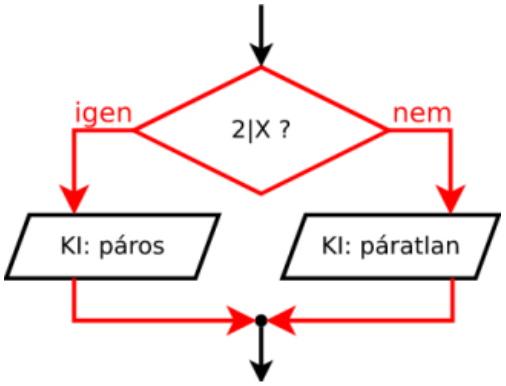
```
int szja_kedvezmeny = (kor >= 18 && kor <= 25)
                      && kereset <= 433000 && !lemondta;
```

A felkiáltójel egy kifejezés előtt az ellentétére változtatja az igazságértéket, igazból hamis lesz, és fordítva. Mivel C-ben nem volt mindig külön `bool` típus (1999 óta van), mint a legtöbb programnyelvben, ezért a feltételes elágazás, vagyis az `if`, elfogad egész számot is. Ha ez a szám 0, akkor hamisnak veszi, ha bármi más, akkor igaznak.

9. C nyelv vezérlő szerkezetei. Feltételes utasítás és ismétlő szerkezetek, egyszerű példákkal történő bemutatása.

Vezérlő szerkezetek: szekvenciális, feltételes elágazó és feltételes ismétlő szerkezetek (ciklusok). Egy kifejezés igazságtartalma alapján határozzák meg a következő végrehajtandó utasítást.

Feltételes szerkezetek



Feltételes szerkezetből kettő is van, az `if` és a `switch-case`. Az `if` egy feltétel szerint dönti el, hogy melyik ága fusson le. Igaz esetén az őt követő blokk, hamis esetén pedig az `else` ág, amennyiben van ilyen. Például:

```
int x;
scanf("%d", &x);
if (x < 0) {
      printf("Negatív számot írtál be.");
} else {
      printf("Nemnegatív számot írtál be.");
}
```

Az `else` után lehet másik if-et tenni, ebből lehet többirányú elágazást csinálni több feltétel alapján, például az előző kódot ki lehet úgy egészíteni, hogy kezelje a 0 esetet is:

```
if (x == 0) {
	printf("0-t írtál be.");
} else if (x < 0) {
	printf("Negatív számot írtál be.");
} else {
	printf("Pozitív számot írtál be.");
}
```

A `switch` egy olyan forma, ahol egy egész szám konkrét értékeire tudsz különböző kódrészeket megadni, például mi fusson le, ha a szám értéke 1, vagy 2, vagy 3… A `switch`-en belül a `case` adja meg, hogy melyik esetben melyik ponton induljon a kód, és a `break` lép ki belőle. Lehet egy `default` ágat is csinálni, ami akkor fut, ha egyik `case` sem illik az értékre. Például:

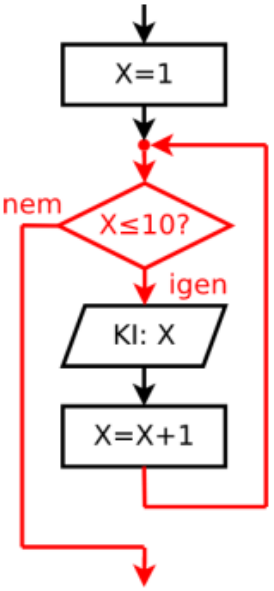
```
int x;
scanf("%d", &x);
switch (x) {
	case 1:
		printf("Megérett a meggy.");
		break;
	case 2:
		printf("Csipkebokor vessző.");
		break;
	default:
		printf("Nem tudom tovább.");
		break;
}
```

Lehetséges olyat is írni, hogy több esethez (`case`) ugyanaz a kód tartozik.

```
switch (x) {
	case 1:
	case 3:
	case 5:
		printf("Balra lépj!.");
		break;
	case 2:
	case 4:
	case 6:
		printf("Jobbra lépj!");
		break;
	default:
		printf("Állj!");
		break;
}
```

Ciklusok

Ismétlő szerkezetből három is van: a `for`, a `while`, és a `do while`. A `for` ciklushoz három részt lehet kitölteni, de egyik sem kötelező: értékadás, feltétel, és változtatás. Az értékadás részben hozhatjuk létre a ciklusváltozókat, például amikkel elszámolunk valameddig. Ilyen például, hogy `int i = 0`. A régi C nem engedte `for`-ban létrehozni a változót, először a ciklus előtt kellett egy `int i`, majd a cikluson belül szabadott az `i = 0`. A feltétel az, hogy meddig fusson a ciklus. Ezt minden futás előtt ellenőrzi, és ha igaz, akkor lefut egyszer a blokkban található kód. Ha a futás után is igaz a feltétel, akkor lefut mégegyszer, ez pedig egészen addig ismétlődik, amíg hamis nem lesz a feltétel, akkor áll le a ciklus ismétlődése. Például adjuk meg az előző után, hogy `i < 2`, ekkor amíg `i` nem éri el a 2-t, a ciklus futni fog. Ha a változtatás `i++`, az azt jelenti, hogy `i` minden futás után 1-gyel növekszik, vagyis `i` először 0, majd 1, majd 2, és ezen a ponton már nem fut le a ciklus, mert már nem igaz a feltétel. Egy példa, ami elszámol 1-től 10-ig (ugyanez van a képen):




```
int i;
for (i = 1; i <= 10; i++) {
	printf("%d\n", i);
}
```

A `while` a `for`-nak egy egyszerűbb változata, egyedül a feltételt lehet megadni. Amíg a feltétel igaz, addig újra és újra ismétlődik. Vegyük az előző példát `while`-ként megírva:

```
int i = 1;
while (i <= 10) {
	printf("%d\n", i);
	i++;
}
```

Az előbb bemutatott két ciklust elöltesztelő ciklusnak hívjuk, mert mielőtt futna a kód, megvizsgálja a feltételt, és ha igaz, lefut a kód. Van azonban egy olyan ciklus, ahol először mindenképp fut a kód, legyen akár igaz, akár hamis a feltétel. A futás után vizsgálja meg, hogy igaz-e a feltétel, és ha igen, újra futni fog, ezt hívjuk hátultesztelő ciklusnak. Ez a különleges tulajdonság, hogy egyszer mindenképp fut, nagyon kevés helyzetre teszi alkalmassá, de ahol szükséges, ott nagyon sokat könnyít, például bemenetet tudunk vele ellenőrizni, ilyen módon működik egy sor beolvasása is:

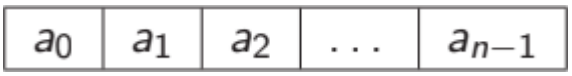
```
char c;
do {
	scanf("%c", &c);
	// valahogy tárolja el a betűt
} while (c != '\n');
```

Nézzük át egy kicsit ezt a kódot: beolvas egy karaktert, majd beolvasás után megnézi, hogy enter volt-e. Ha nem, akkor fut tovább, ha igen, akkor végzett, beolvastuk a sort. Ez sima `while`-ban bonyolultabb és csúnyább kód lenne.

10. C nyelv összetett adattípusai. Tömb és elemeire való hivatkozás. Pointer. Sztring kezelés.

Összetett adattípus a tömb és a struktúra, de ide vehető karakterek tömbje, vagyis a sztring.

Tömb: Egyforma típusú, szomszédosan elhelyezkedő elemek fix méretű tárolója a tömb. Egy tömböt elemeinek típusa és elemeinek száma határoz meg. A tömb elemei a sorszámuk segítségével érhetők el, a kezdőelem sorszáma 0. Elemei származtatott illetve összetett típusúak is lehetnek. Elérésük indexeléssel bármilyen sorrendben lehetséges, **n** méretű tömbnél az **n-1**. elemig:



Tömb létrehozása (“deklarációja”):

```
<típus> <név>[<darabszám>];
```

Például `int data[5];`

Tömb indexelése: `<azonosító> [<elem index>]`, például:

Egy tömböt kezdeti értékekre beállítani (inicializálni) több módon lehet. Ha nem csinálsz semmit, csak a korábbi `int data[5];` módon deklarálod, nem tudni, mi lesz benne. Ilyenkor a C úgy működik, hogy kijelöli a memória egy részét, ahol a tömb lesz, de nem üríti ki, nem állítja be 0-ra (mint mondjuk a C#), hanem ami korábban oda volt írva a memóriába, vagyis random memóriaszemét lesz a kezdeti érték. Ezért ajánlott megadni valamit, ha tudunk. Erre több mód is van. Először is megadhatjuk az összes konkrét értéket:

```
int data[5] = { 4, 2, 0, 6, 9 };
```

Ilyenkor a tömbben a leírt adatok lesznek kezdetben. Ha nem írjuk le az összeset, azt kiegészíti 0-val, például van ez a tömb:

```
int data[5] = { 4, 2 };
```

Itt két számot adtunk meg az 5 elemű tömbből, amit a fordító úgy vesz, hogy a maradék 0, vagyis a tömb elemei: 4, 2, 0, 0, 0. El lehet hagyni, hogy hány elemű a tömb, ilyenkor viszont pontosan akkora lesz, ahány elemet megadunk. Ez például egy kételemű tömb:

```
int data[] = { 4, 2 };
```

Tömböt általában `for` ciklussal járunk be, 0-tól a tömb méretéig, és feldolgozzuk az adott elemet. Mivel a tömb méretéig megyünk, de annyiadik eleme már nem létezik, ezért nem <=, hanem csak < összehasonlítást nézünk az elem számára:

```
for (int i = 0; i < 5; i++) {
    printf("%d\n", data[i]);
}
```

Struktúra: Tetszőleges, akár különböző típusú, szomszédosan elhelyezkedő elemek halmaza a struktúra. A struktúrának legalább egy eleme kell legyen. Az elemek (más szavakkal mezők vagy tagváltozók) az egyedi nevük segítségével érhetők el. Egy név csak egyszer szerepelhet. Elemei származtatott típusúak is lehetnek. Ezt a formát követik:

```
struct <név> {
	<típus> <adattag neve>;
	<típus> <adattag neve>;
	<típus> <adattag neve>;
	...
}
```

Lényegében több változót csomagol egybe, ha közösen kell kezelnünk őket. Erre egy nagyon egyszerű példa egy RGB szín, aminek három összetevője a piros, a zöld, és a kék szín fényerőssége:

```
struct Color {
	int red;
	int green;
	int blue;
}
```

Hogy egy struktúra egy változó típusa is lehessen, típusként kell definiálni és nevet kell neki adni. Ehhez elé kell írni a typedef kulcsszót, majd a struktúra lezárása után a nevét. Ilyenkor a nyitó { elé nem kötelező nevet írni. Így néz ki tehát végleges kódban:

```
typedef struct Color {
	int red;
	int green;
	int blue;
} Color;
```

Természetesen lehetne több különböző típusú adattagja is (például valakinek a neve és TAJ-száma), de itt nincs rá szükség. Így ennek a struktúrának a segítségével egyetlen változóként át tudunk adni egy színt, amitől sokkal átláthatóbb programot fogunk írni. Egy ilyen színt így hozunk létre:

```
Color valami;
```

Így a változóban tárolt struktúránk neve `valami`, a mezőire pontokkal hivatkozunk, vagyis ilyen módon írhatjuk át a tagváltozóit:

```
valami.red = 71;
valami.green = 65;
valami.blue = 89;
```

Egy struktúrát simán át lehet másolni egy másik változóba, egyetlen lépésben, ilyenkor az értékadás minden mezőt másol:

```
Color copy = valami;
```

Függvény is átvehet paraméterként struktúrát, ahogy visszatérési értéke is lehet:

```
Color kontraszt(Color eredeti, float szint) { ... }
```

Pointer: A pointer, magyarul mutató, egy olyan változó, ami memóriacímet tartalmaz, és azt az információt, hogy ott milyen típusú adat található. Alapból a memóriát úgy kell elképzelni, mint egy hatalmas, több GB-os tömböt, amiben mindenféle adat van random pontokon. A pointer megmondja, hogy ebben a hatalmas halmazban hol kezdődik egy adat, például ha egy 4 bájtos `int` a 25., 26., 27., és 28. bájtot foglalja el, egy rá mutató pointer értéke 25 lesz, onnantól pedig tudjuk, hogy 4 bájtot foglal, mert `int`. Így néz ki egy int pointer deklarálása:

```
int *ptr;
```

Itt `ptr` néven létrehozunk egy olyan pointert, ami egy `int` típusú változóra mutat, csak még nem állítottuk be, hogy melyik memóriacímre. Ha sehova nem mutat, azt hívjuk `NULL`-nak. A csillag a pointer jele, és nem a típushoz tartozik, hanem a változóhoz, hiába engedi a C úgy is leírni, hogy `int* ptr;`. Erre az a bizonyíték, hogy ha egyszerre több változót hozunk létre, például `int *ptr1, ptr2;`, akkor kizárólag `ptr1` lesz pointer, `ptr2` csak sima `int`. Helyesen úgy kell két pointert egyszerre létrehozni, hogy `int *ptr1, *ptr2;`. Nézzük, hogyan adunk neki értéket. Lehet kézzel is beírni számot, de ennek nem sok értelme van, inkább le tudjuk kérni más változók memóriacímét az & karakterrel. Ez a kód csinál ilyet:

```
int x = 3;
int *ptr = &x;
```

Az & jel neve címkéző operátor, és egy változó memóriacímét mondja meg. A példakódban `ptr` arra a memóriacímre mutat, ahol `x` értéke található, tehát ha megnézzük, milyen `int` érték található ott, akkor az 3 lesz. Amikor megnézzük, milyen érték van azon a helyen, ahová a pointer mutat, esetleg dolgozunk vele, azt dereferálásnak hívjuk, és a jele szintén csillag. Például így írjuk át a 3-ast 8-ra:

```
int x = 3;
int *ptr = &x;
*ptr = 8;
```

Ha csak azt írnánk, hogy `ptr = 8;`, az tönkretenné a programot, és ha kiírnánk, valami random szám lenne, ugyanis a memóriacímet írná át, nem azt az értéket, ami a memóriában az adott helyen van. Pointert ugyanúgy át lehet adni függvényben, mint bármelyik más típust, és másolható is. A tömbök valójában pointerek, és azt mondják meg, hogy az első elem hol kezdődik. Nézzük például ezt:

```
int nums[] = { 1, 2, 3, 4, 5 };
int *ptr = nums + 1;
printf("%d", *ptr));
```

Itt egy jellegzetességét figyelhetjük meg a pointereknek, mégpedig ha hozzájuk adunk valamennyit, akkor annyi elemmel léptetjük őket. A tömb elejére mutató pointerhez (a tömb változója) adtunk egyet, tehát a `ptr` már a második elemre fog mutatni. Ezt a kiírással már látjuk is, ez a kód futtatva 2-t ír ki. Sok nyelvvel ellentétben a tömb hosszát semmi nem tárolja, ez minden esetben a mi feladatunk, ha szükség van rá.

String: C-ben nincs olyan natívan kezelt `string`, mint pl. Python-ban vagy C#-ban. String alatt karakterek 0-val lezárt tömbjét értjük, nincs string típus. Éppen ezért egyik módja a string deklarálásának, ha leírjuk pontosan a karaktertömböt:

```
char str[] = { 'h', 'e', 'l', 'l', 'o', 0 };
```

A lezáró nulla a legfontosabb ebből az egészből, amit egyszerűen beírhatunk számként (elvégre a karakter technikailag egész szám), vagy úgy is, hogy `'\0'`, ez jelenti a nulla karaktert. Azért kell minden szöveg végére, mert ahogy a tömböknél vettük, a C nem tudja, hogy milyen hosszú egy tömb. Ezért kell neki egy jel, hogy itt hagyja abba a szöveg kezelését, és például eddig írja ki. Ha a nulla után teszünk még karaktereket, nem fog velük dolgozni, és ugyanúgy annyi marad a string hossza, ahány karakter a 0 előtt van.

Nem kötelező egyébként tömbként kiírni, lehet a más nyelvekből ismert módon, csak a jó típust kell a változóhoz adni, pointert vagy tömböt:

```
char *str1 = "hello";
char str2[] = "hello";
```

Ha így írjuk be a szöveget, automatikusan egy 6 elemű tömb jön létre, az 5 betűvel és a lezáró nullával. A stringekről érdemes még tudni, hogy a `%s` írja ki őket, ha printről van szó, és ugyanezzel olvashatjuk be őket:

```
printf("%s\n", str);
```

A stringes függvényekhez include-olni a `string.h` fájlt kell. Ezek a fontosabb függvények:

- `strcmp(char *a, char *b)` - Összehasonlítja a két stringet, egy int-et ad vissza. Ha 0-t ad vissza, akkor a két string egyenlő, ha mást, az arra utal, hogy melyik van előrébb az ábécében. Negatív visszatérési érték esetben az első, pozitív esetben a második.
- `strcpy(char *to, char *from)` - Stringet másol a második paraméterként kapott tömbből az első paraméterként kapott tömbbe.
- `strlen(char *str)` - Megmondja egy string hosszát.
- `strcat(char *str, char *extra)` - Az első paraméterben kapott string végéhez hozzáfűzi a második paraméterben kapott stringet, pl:
  ```
  char str[80] = "";
  strcpy(str, "Ezek a ");
  strcat(str, "stringek ");
  strcat(str, "össze vannak fűzve ");
  strcat(str, "az str-ben.");
  ```
- `strstr(char *haystack, char *needle)` - Megkeresi a második szöveget az elsőben, és visszaadja annak helyét pointerként. Programozásban a keresést úgy szoktuk jellemezni, hogy amit keresünk, az a tű, és ahol keressük, az a szénakazal.

11. Függvény tulajdonságai. Függvény definiálás, elhelyezése a kódban, függvény hívása. Könyvtári függvények alkalmazási szabályai.

A **függvény**ek olyan kis programrészek, amelyek egy jól elhatárolt részfeladatot hajtanak végre. Használatukkal egyszerűbbé és karbantarthatóbbá tehetjük a kódot. Dokumentáláskor pontosan meg kell határozni, hogy milyen feladatot hajt végre. Van be- és kimenete. Rögzíteni kell, hogy milyen bemenetet vár, és milyen kimenetet állít elő. Hasonló a matematikai függvényekhez. A függvénynek neve van, akár több helyen is hivatkozhatunk rá. Egy bonyolult részfeladat elemi lépésként kezelhető, a részfeladatokat külön függvényben írhatjuk meg. A függvényeknek paraméterei is lehetnek: a részfeladatokat tudjuk így konfigurálni. Például a `printf` is egy függvény, és mindenhol használjuk. Függvényt a következő formában definiálunk:

```
<visszatérési típus> <név>(<paraméterek>) {
	<kód>
	return <visszatérési érték>;
}
```

A függvény első sorát (típust, nevet, paramétereket) úgy hívjuk, hogy a függvény fejléce. Magát a kódot úgy hívjuk, hogy a függvény törzse. A paraméterek vesszővel vannak elválasztva, és a típusuk nyugodtan lehet pointer vagy struktúra is a primitíveken (vagyis az alapvető C-s típusokon) túl. Amennyiben a visszatérési érték típusa nem `void`, a kiszámolt eredményt a `return` szóval kell jeleznünk, ilyenkor a függvény eredményét a hívási helyen elmenthetjük változóba vagy használhatjuk helyben. Egy egész számokat összeadó függvény például így néz ki:

```
int add(int a, int b) {
	return a + b;
}
```

Egy függvény törzsén belül lehetnek úgynevezett lokális változók, amik csak itt léteznek, a függvényen belül vannak definiálva. A függvénybe lépéskor jönnek létre, és a végén megszűnnek, értékük elveszik. Más függvényeknek lehetnek ugyanolyan nevű lokális változói, minden függvény csak a sajátjait látja.

C-ben csak olyan függvényt lehet használni, amit már definiáltunk, ezért csak akkor fordul a kód, ha a függvény a használati helye előtt lett megírva. Ez lehetetlenné tenné, hogy két függvény egymást hívja, amire néha szükségünk van, ezért kitalálták a prototípust, amivel leírjuk, hogy majd lesz egy ilyen függvény, nyugodtan fordítsa le a kódot. Ilyenkor a függvény fejlécét írjuk a használati hely elé, például így írjuk meg az összeadást a használat után:

```
int add(int a, int b);

int main() {
	printf("2 + 3 = %d", add(2, 3));
	return 0;
}
```

```
int add(int a, int b) {
	return a + b;
}
```

Függvényeket úgy hívunk, hogy leírjuk a nevüket, és megadjuk, hogy a paramétereik milyen értéket kapjanak, például láttuk a fenti példán, hogy `add(2, 3);`. Ilyenkor ugyanúgy behelyettesíthetők, mintha változók lennének, például elmenthető az értékük, vagy kezelhető bármi olyan helyen, ahová adott típusú adat kell, lásd a kiírást az előbbi példán, ahol egy számot írunk ki, és a paraméter helyén hívtuk a függvényt.

Könyvtári függvények alkalmazási szabályai: Abszolút fogalmam sincs, hogy ez mit akar jelenteni, mert nincsenek rájuk szabályok. Ezek a függvények bármikor használhatóak, ha a hozzájuk tartozó megfelelő fájl include-olva van. Az amúgy így néz ki, például legyen a `printf` a példa, ami az `stdio.h` fájl része. Ha saját fájlokat akarunk importálni, az így néz ki:

```
#include "valami.h"
```

De ha könyvtári fájlokat, akkor idézőjel helyett kacsacsőrök vannak:

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

Néhány példa:

- `stdio.h` - kiírás és beolvasás, fájlkezelés
- `stdlib.h` - memóriakezelés, rendezés
- `string.h` - szövegkezelés
- `math.h` - matematikai függvények (szinusz, gyök…)

12. Állományok kezelése. Bináris és szöveges állományok alapműveletei.

A C nyelv kétféle magas szintű fájlkezelést támogat: egy általános fájlt és egy speciális fájlkezelést. Az általános fájlkezelést bináris fájlkezelésnek nevezzük. Ha egy fájlt binárisan kezelünk, akkor a fájlkezelő függvénynek egy memóriacímet adunk (ahová ír vagy ahonnan olvas), és egy egész számot, amely megmondja, hogy hány bájtot szeretnénk beolvasni vagy kiírni.

A speciális fájlkezelés alatt azt értjük, hogy a nyelv ismeri az adott fájltípus belső felépítését, azt tehát a programozónak nem kell. A C nyelvben támogatott speciális fájl a szöveges fájl. Más programnyelvek (pl. C#) támogatják például a JPG, PNG, és egyéb grafikus állományokat, a ZIP-archívumokat, stb. A C sajnos nem ilyen bőkezű, bár léteznek külső fejlesztésű könyvtárak ezen fájltípusok támogatására.

Fájlok kezelése az `stdio.h`-ban (ezt kell include-olni) megadott `FILE*` típusú pointerrel és függvényekkel lehetséges. A megnyitás függvénye:

```
FILE* fopen(const char *filename, const char *mode);
```

Kap tehát egy fájlnevet, ahol megadhatunk teljes elérési útvonalat vagy csak egy fájlnevet, ekkor abban a mappában fogja keresni, ahonnan elindult a program. A visszatérési érték a fájl kezelésére használt pointer, vagy ha nem sikerült a megnyitás, akkor `NULL`. A mód helyére két betűt kell írni stringként:

- Első betű: **megnyitás módja**
  - `"r"`: read/**olvasás**: A fájlt olvasásra kérjük el a rendszertől, egyszerre több program tud olvasni.
  - `"w"`: write/**írás**: A fájlt írásra kérjük el a rendszertől, egyszerre csak egy program tud írni. Ha a fájl nem létezik, létrehozza, ha viszont létezik, töröl belőle mindent, vagyis megnyitás után garantáltan üres fájllal dolgozunk.
  - `"a"`: append/**hozzáfűzés**: Ugyanaz, mint az írás, de ha a fájl létezik, nem töröl belőle semmit, hanem a végétől folytatja az írást. Naplózáshoz jól jön.
- Második betű: tartalom típusa
  - `"t"`: text/**szöveg**: A fájlban szöveg van, egyszerű szövegkezelési függvényekkel is lehet használni. Ez a betű elhagyható, ha csak a megnyitás módját írjuk le, akkor alapból szövegfájlnak veszi.
  - `"b"`: binary/**bináris**: Nyers bájtokat, így bármilyen adattípust tudunk írni vagy olvasni.
- Lehetséges egyszerre írásra és olvasásra is megnyitni egy fájlt. Ekkor ugyanúgy ki kell választani egy megnyitási módot (olvasás, írás, vagy hozzáfűzés), amitől az függ, hogy a fájlt töröljük vagy nem, illetve hogy az elejéről vagy végéről kezdjük. Az írás-olvasás módot egy `"+"` karakter jelöli a két betű után.

A bináris fájl és a szövegfájl közötti különbség leginkább úgy érthető meg, ha a kettőt összehasonlítjuk. Tegyük fel, hogy adott egy `int` változónk, legyen a benne tárolt érték 1000000. Ezt leírni 1 darab egyes, 6 nulla, illetve egy sor vége vagy bármi más, ami a szám végét jelöli, ez összesen 8 karakter. Szövegesen tehát ez a szám 8 bájtot foglal. Ha binárisan kezeljük a fájlt, akkor annyi bájt kell egy `int`-nek, amennyit előír a típus, tehát a legtöbb gépen 4.

Bináris fájl

A bináris fájlok azok, amelyek nem szöveget tartalmaznak. Legtöbbször bájtról bájtra kiírunk valamilyen memóriaterületet, ezekhez az `fread` és `fwrite` függvények használhatóak. A két függvény paraméterezése egyforma:

- `fread(pointer, méret, darab, fájlpointer)`
- `fwrite(pointer, méret, darab, fájlpointer)`

Ezek a `pointer` által mutatott memóriaterületet olvassák a fájlból vagy írják a fájlba. A `fájlpointer` egy `FILE*` típusú pointer, amit a fájl megnyitásakor kapunk. A `méret` mondja meg, hogy egyetlen adat hány bájt, a `darab` pedig azt, hogy mennyit írjunk vagy olvasunk belőle. Így lehet tömböt írni vagy olvasni, egyszerű változóknál ez a szám természetesen 1. Nincs meghatározva, hogy milyen típusú elemekre mutat a pointer, ezt `void*`-nak hívjuk. Ezek a függvények tehát nem foglalkoznak az adatok értelmével, egyszerűen elvégzik a fájlműveletet, adatot mozgatnak a memória és egy fájl közt. Mindkét függvény egy egész számot ad vissza, mégpedig azt, hogy hány darab elemre sikerült elvégezni a műveletet.

Szövegfájl

A szövegfájlok azokat a karaktereket tartalmazzák, amelyeket a `printf` a képernyőre is írna. A szövegfájlok kezeléséhez a `printf` és `scanf` fájlokhoz készült változatát, az `fprintf` és `fscanf` függvényeket használjuk. Ezek első paramétere a megnyitott fájl, a többi pedig ugyanaz, ami az eredeti függvényeké lenne.

```
FILE *f = fopen("random.txt", "w"); fprintf(f, "Hello, fájl!"); fclose(f);
```

A szövegfájlokat lineárisan kezeljük, nem ugrálunk bennük ide-oda. Elméletileg lehetséges, de ki kellene számolni hozzá, hogy pontosan hol kezdődik egy adott rész. Ezt meg nem ismerjük, amíg nem olvastuk be a többi sort, hiszen nem ismerjük a hosszukat.

A szövegfájlok jól hordozhatók, hiszen bennük egyedül a sor végének jelölése változhat két rendszer között, azt pedig könnyen tudják kezelni. Binárisoknál már probléma, ha két architektúra kezelheti máshogy a bájtok sorrendjét, ugyanis előfordulhat, hogy egyik gépen a nagyobb helyiértékek vannak balra (mint ahogy mi is ábrázoljuk a számainkat), de máshol már jobbra.

További függvények

- `fseek` - Adott sorszámú bájtra ugorhatunk vele, abszolút vagy relatív helyet megadva. Első paramétere a fájlpointer, második a bájtok száma, harmadik, hogy a fájl elejétől/végétől vagy a jelenlegi pozíciótól számoljon annyi bájtot.
- `fputs` - Ez egyszerűen kiírja az első paraméterként kapott szöveget, nem formázza meg úgy, mint az `fprintf`. Éppen ezért lehetséges az is, hogy előbb jön a kiírandó string, aztán a fájlpointer a paraméterek között. Ennek a konzolos verziója a `puts`, ami ír új sort a sor végére, de az `fputs` nem.
- `fputc` - Ugyanaz, mint az `fputs`, csak nem stringgel, hanem egyetlen karakterrel. A konzolos verziója ennek is f nélkül van, `putc`.
- `fgetc` - Egyetlen karaktert olvas, amivel vissza is tér. Ha nem tudott mit olvasni, mert vége a fájlnak, egy EOF karaktert ad vissza, ami a -1-es számnak felel meg. Egyetlen paramétere a fájlpointer.
- `fgets` - Adott hosszúságú szöveget olvas. Három paramétere a karaktertömb pointere, amibe bele fog olvasni, az olvasandó karakterek száma, és a fájlpointer.

13. Dinamikus adatszerkezet. Adatterület foglalás és felszabadítás futási időben. Saját típus definiálása és pointer.

Eddig, ha nagyobb mennyiségű adatot tároltunk, tömböket használtunk. Igen gyakran azonban fordítási időben nem lehet előre tudni, hogy mennyi adatot kell majd feldolgozni. Két eset van:

- Az egyik, ha azt tudjuk mondani, hogy „ennyi hely biztosan elég lesz”. Például sztringek beolvasásánál mást nem is nagyon tudunk csinálni, legfeljebb ha rendkívül megbonyolítjuk a programunkat.
- A másik eset, ha saccolni sem tudunk.

Mondhatnánk erre, hogy akkor foglaljunk pl. 100 millió elemű tömböt. Ez egyrészt nem gazdaságos, hiszen mi van akkor, ha az esetek túlnyomó többségében ennél sokkal kevesebb hely kell? Feleslegesen foglalná a programunk a rendszer memóriáját. Másrészt viszont mi van, ha nem elegendő a 100 millió sem? (Előre ismeretlen mennyiségű adatnak nagyméretű tömböt lefoglalni igen súlyos hiba.) Ha alábecsüljük egy tömb méretét, könnyen túlindexelhetjük, amitől a program instabillá válhat.

Másik probléma a tömbökkel, hogy nem tudjuk kontrollálni az élettartamot. Vagy létrehozunk a függvényeken kívül egy úgynevezett globális változóként tömböt, ami a program teljes futása közben foglalja a memóriát, vagy függvényen belül hozzuk létre, de ekkor a függvény végén megszűnik. Ezek közül egyik sem a helyes gyakorlat, mindkettő hibák melegágya.

A megoldás, ha a memóriát nem előre kérjük, hanem a program futása közben, az aktuális igényeinknek megfelelően (dinamikusan) foglaljuk le, és ha már nincs rá szükség, felszabadítjuk. Ilyenkor mi dönthetjük el, hogy mennyi memóriát foglalunk, mikor foglaljuk le, és mikor szabadítjuk azt fel. Ezekért az előnyökért cserébe a mi felelősségünk a foglalások felszabadítása, ugyanis ha elfelejtjük, a program szépen lassan elkezdheti felzabálni az összes memóriát, ezt hívjuk úgy, hogy a program szivárog.

Az igényelt memóriát az operációs rendszer biztosítja a rendelkezésre álló szabad memória terhére. Ehhez a C nyelvben két memóriafoglaló függvény áll rendelkezésre: a `malloc` és a `calloc`. A két függvény közötti különbségek a következők:

- A `malloc`-nak egy paramétere van: hány bájtot szeretnénk lefoglalni. Ezzel szemben a `calloc`-nak két paramétere van, és a kettő szorzata adja a kívánt bájtszámot.
- A `malloc` által lefoglalt memóriaterület „memóriaszemetet” tartalmaz, azaz a dinamikusan lefoglalt változó kezdeti értéke bármi lehet, ahogyan ezt az egyszerű, nem dinamikus változóknál is megszoktuk. A `calloc` ezzel szemben a lefoglalt terület minden bájtját kinullázza.

Más különbség nincs, a használat és a felszabadítás ugyanaz mindkét esetben. **A függvények visszatérési értéke egy `void*` típusú pointer** (amit majd át kell cast-olni a kívánt típusra, vagyis a `malloc` elé kell írni zárójelben, milyen típusú adatot foglaltunk) a lefoglalt memóriaterületre, vagy ha nem sikerült lefoglalni (nincs a kívánt nagyságú összefüggő memóriaterület az operációs rendszer birtokában), akkor `NULL` pointer, ezt érdemes minden foglalás után ellenőrizni. A dinamikus memória kezeléséhez szükséges függvények használatához szerkesszük a programunkhoz (include) az `stdlib.h` fejlécállományt!

Ha a lefoglalt tömböt valami miatt át kell méretezni, a `realloc` függvényt használjuk: első paramétere a korábbi foglalás pointere, második az új méret, amit igénylünk. Ez foglalni fog egy új területet az új mérettel, annak az elejére átmásolja a korábbi adatokat, ez viszont lassú lehet. Ha kisebb méretet választunk, az utolsó néhány elemet törli, ha viszont nagyobbat, a tömb végéből memóriaszemét lesz. Az újrafoglalás miatt az új tömb természetesen más helyen is lesz a memóriában, ezért egy új pointerrel tér vissza, amire le kell cserélni az előzőt.

A lefoglalt dinamikus memóriát kötelező felszabadítani a `free` függvénnyel! Egyetlen paramétere egy pointer, ami dinamikus tömbre mutat. Ha ezt elfelejtjük, a memória lefoglalva marad használat után is, így a végtelenségig nőhet, mennyit használ a program, vagyis szivárog. Gyakori infós mondás, hogy “ahány `malloc`, annyi `free` legyen”, egyszerűbb programoknál ez jó irányelv. A `malloc` által adott pointer nagyon fontos! Ha elveszítjük, nem tudjuk felszabadítani a lefoglalt területet. Olyan memóriaterület, ami nincs lefoglalva (pl. korábban felszabadítottuk, de még maradt rá pointer), nem használható!

Az utolsó fontos összetevője a dinamikus adatfoglalásnak, hogy ismerjük a `sizeof` operátort: egyszerűen annyit csinál, hogy megmondja, hány bájt kell egy típushoz. Például tanultuk, hogy egy `float` 32 bites, vagyis 4 bájtos, így a `sizeof(float)` helyre a fordító azt helyettesíti be, hogy 4. Erre azért van szükségünk, mert a `malloc` bájtokat foglal, és ha csak annyit írnánk be, hogy 6 egységet foglaljon, majd `float` tömböt csinálnánk belőle, akkor nem 6 eleme lenne, hanem másfél. Éppen ezért egy `malloc` így néz ki például egy 6 elemű `float` tömb foglalására:

```
float *nums = (float*)malloc(6 * sizeof(float));
```

Random kérdések

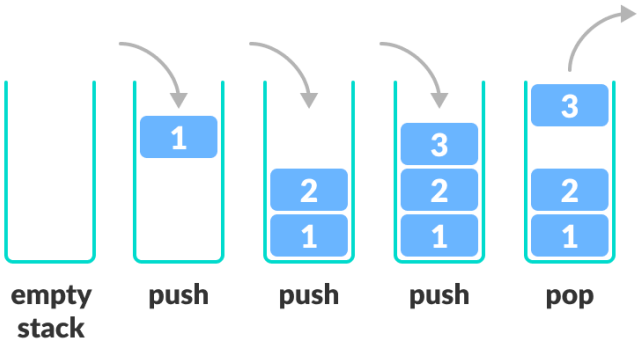
- Hogyan lehet lekérdezni, hogy hány elemű tömböt foglaltunk? – Sehogyan, nekünk kell tudni.
- Hogyan lehet ellenőrizni, adott terület le van-e foglalva? – Sehogyan, nekünk kell tudni.
- A `sizeof` nem tudja megmondani egy dinamikus tömb méretét? – Nem, mert ilyenkor a változó méretét nézi, ami egy pointer, vagyis általában 4 bájt. A méretet is nekünk kell tudni.
- Lehet nulla méretű területet foglalni? – Lehet, de fel is kell szabadítani.
- Szabad `free(NULL)` hívást csinálni, vagyis a semmit felszabadítani? – Szabad.

A memóriakezelés áttekintése

A három memóriaterület:

- Egy változó definíciójának módjától függ, hol jön létre. A definíció módja főként annak a helyét jelenti, ami az élettartamot is meghatározza:
    - Függvény belsejében: lokális változó, függvény végén megszűnik
    - Függvényen kívül: globális változó, a program futása során végig létezik
- Ezeket a helyeket tárolási osztályoknak nevezzük:
    - Globális változók memóriaterülete = data segment
    - Lokális változók memóriaterülete = stack segment (verem)
    - Dinamikusan foglalt memóriaterület = heap (kupac)
- A globális területet kivéve a méretük változik is. A globális azért nem, mert már a

program elején tudjuk, hogy a programnak mennyi globális változója van, és azok mekkorák.
    - A veremben lévő adatok mennyisége függvényhíváskor nő (új függvények új változókat tesznek a tetejére), függvények végén pedig ezek törlődnek a tetejéről:



    - A kupac mérete is változik, attól függően, hogy mikor foglalunk vagy szabadítunk fel.
- Természetesen a számítógépben nincs többféle memória, hanem a memória különböző részeit kezeljük különböző módokon, és más célból teszünk oda adatokat. Éppen ezért a pointerek bármelyik memóriaterület változóira tudnak mutatni.
- Egy függvény változójából úgy csinálunk örökéletű (kvázi globális) változót, hogy eléírjuk a `static` kulcsszót. Ettől nem szűnik meg a függvény végén, hanem a következő futáskor megmarad a korábban felvett értéke.

Ajánlott irodalom

Van ez a doksi: Informatika - gépelt kidolgozás.pdf, ebben az 50. oldaltól van egy nagyon nagyon gyors összefoglalója mindannak, amit itt olvashattál (info szigó mini), erősen ajánlott átnézni.

A konzolra írás és bemenet beolvasása C-ben többféleképp történhet. Egy sornyi szöveget a `puts` ír ki, egy sort pedig a `gets` olvas be, viszont C-ben ezek karaktertömbökkel dolgoznak, és ezen a nyelven nem divat alakítgatni számból szöveggé vagy vissza. Helyette a formázott kiírás és beolvasás van, ezek a `printf` és `scanf` függvények, amik nevében az `f` azt jelenti, hogy formatted. Mindkettő ugyanúgy működik: az első paraméter a forma, majd következnek azok a változók, amiket a formázott helyekre akarunk írni vagy amikben el akarjuk menteni, amit beolvastunk. Például a kiírás:

```
int num = 1;
printf("Így írsz ki egészet: %d", num);
```

Látszik tehát, hogy egy százalékjel jelöli, hogy hova kell behelyettesíteni valamit, egy betű jelöli a típust (`d` az egész számot), és amiket be kell helyettesíteni a helyekre, azt felsoroljuk paraméterként. Fontos megjegyezni, hogy a `printf` nem ír új sort maga után, ezt a `\n` karakterrel lehet jelölni. Néhány típus jelölése:

- `%d` - egész szám
- `%f` - valós szám
- `%s` - szöveg
- Ha egy u betűt teszel ezek után, akkor előjel nélküli számnak veszi, pl. `%du`

A `scanf` esetében ugyanez a minta, ha mondjuk két egész számot olvasunk be, így néz ki:

```
int a, b;
scanf("%d %d", &a, &b);
```

Néhány dolgot azonban meg kell magyarázni: szóközzel van elválasztva a két beolvasott szám, de mégis működik úgy, ha mindkét számot enterrel adjuk a gépnek. Egyszerűen csak legyenek a formátumban felsorolva, és szinte bárhogy be lesznek olvasva. Itt már a változók elé oda kell írni az & karaktert, vagyis a memóriacímüket adjuk át a függvénynek. Ez azért fontos, mert ha csak bemásoljuk a változók értékét, akkor nem lehetne őket felülírni, így a pointerek használatára lesz szükség, hogy módosuljon a változóink értéke ott is, ahol hívjuk őket. Ez egy nagyon gyakori használati mód a pointerekre más kódban is.