10. Informatikai alapok

Írta: Pintér Olivér, Proksa Gábor

Lektorálta: Szabó Ádám

Ebben a tananyagrészben megismerkedünk az informatika alapjaival, amelyet egy kis ismétléssel kezdünk, majd a C nyelv alapjainak megismerésével folytatjuk. A tananyagrész végére már képesek leszünk egyszerű programokat készíteni. Ezeket egyelőre nem mikrokontrolleres környezetben, hanem az egyszerűség kedvéért PC-n futtatjuk.

SZÁMRENDSZEREK (ISMÉTLÉS)

Először kicsit ismételjük át, amit a számrendszerekről tanultunk. A hétköznapi életben a 10-es számrendszer az elterjedt, ezt tanuljuk már az általános iskolától kezdve. Ennek a hátterében az húzódik meg, hogy tíz ujjunk van, így egyszerűen tudunk vele számolni, ábrázolni.

Az informatikában viszont a kettes számrendszer terjedt el, mivel ott ezt egyszerű megvalósítani.

10-ES SZÁMRENDSZER

A 10-es számrendszer számjegyei a 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 és a helyiértékei a 1, 10, 100, 1000, 10000, … vagyis a 10 hatványai. A matematikában általában nincs rá külön jelölés. C nyelvben ugyancsak sincs rá külön prefix (azaz előtag), így a megszokott formában tudjuk őket használni, ezt majd később látni is fogjuk.

2-ES SZÁMRENDSZER

A 2-es vagy más néven bináris számrendszer esetén használt számjegyek a 0 és az 1, a helyiértékei pedig az 1, 2, 4, 8, 16, 32, 64, … vagyis a kettő hatványai. A matematikában általában az alsó indexbe írt 2-essel jelöljük, hogy a leírt számot kettes számrendszerben kell érteni. A C nyelvben a *"0b"* prefix szolgál erre.

Nézzünk most meg egy-egy példán keresztül, hogy hogyan is kell leírnunk egy bináris számot. Matematikában a következőképp néz ki egy bináris szám 10111011001_2, míg C nyelven a `0b10111011001` fogja jelenteni ugyanezt.

16-OS SZÁMRENDSZER

A 16-os vagy hexadecimális számrendszer esetén a 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A (=10), B (=11), C (=12), D (=13), E (=14), F (=15) számjegyeket használjuk, a helyiértékek pedig az 1, 16, 256, 4096, 65536, … vagyis a 16 hatványai. A 16-os számrendszer esetén a számjegynek kell tekinteni a [A-F] betűket is, mivel így tudjuk egyszerűen és nem összetéveszthetően ábrázolni őket.

A matematikában az alsó indexbe írt 16-os szám jelöli azt, hogy az hexadecimális számrendszerbeli. A C nyelv esetén a *"0x"* prefix hivatott a hexadecimális számokat jelölni.

Itt is nézzünk meg egy-egy példát. Matematikában így írunk le egy hexadecimális számot $\texttt{f32ab6527a83c3}_{16}$, míg C nyelvben ugyanezt a `0xf32ab6527a83c3` leírás fogja jelenteni, a kis és nagybetű a szám leírásánál egyenértékű.

SZÁMRENDSZEREK KÖZÖTTI ÁTJÁRÁS

Olykor előfordulhat, hogy át szeretnénk térni egyik számrendszerből egy másikba. Az átváltás 10-es számrendszerből egy másik számrendszerbe az általános iskolában megtanult módszerekkel is történhet, azaz osztófával, ahol feljegyezzük, hogy mennyi az alap által adott egészrész és a maradék. Ezt addig iteráljuk, vagyis ismételjük, ameddig az egészrész 0 nem lesz. Ezt követően a kapott maradékokat lentről felfelé olvasva megkapjuk a számot, de már az új alapú számrendszerben. Nézzünk erre egy példát. A 37-et szeretnénk átírni 2-es számrendszerbe. Az eljárás a fent leírtak szerint a következő alakot ölti.

37| 1
18| 0
9| 1
4| 0
2| 0
1| 1
0|

A 37-t tehát kettes számrendszerben a 100101_2 alakban írható fel. Más alapú számrendszer esetén is hasonlóan kell eljárni. A 16-os számrendszerbe a 2-es számrendszerből könnyedén áttérhetünk, ha a kettes számrendszerbeli számot négyes csoportokra bontjuk szét és meghatározzuk azok értékét. Emlékszel még erre, hogyan is történik ez? Ha nem, nyugodtan nézz bele a “Matematika alapok ismétlése” című tananagyrészbe.

A visszaírás decimális számrendszerbe is egyszerűen elvégezhető. Ehhez a számjegyeket a nekik megfelelő helyiértékű hatvánnyal kell összeszoroznunk, majd a végén összeadnunk azokat. Például a 100101_2 bináris szám 10-es számrendszerben $1 \cdot 2^5 + 1 \cdot 2^2 + 1 \cdot 2^0 = 37$ vagy az $1B_{16}$ hexadecimális szám pedig 10-es számrendszerbe átírva $1 \cdot 16^1 + 11(= B) \cdot 16^0 = 27$ lesz.

LOGIKAI MŰVELETEK (ISMÉTLÉS)

Röviden elevenítsük fel, amit a logikai műveletekről tanultunk. A logikai változókkal műveleteket tudunk végezni, a legegyszerűbb ilyen művelet a negálás, amely pont az ellentétét rendeli hozzá a logikai változónkhoz. Például $A = 1$-nek a negáltja, vagyis az $\underline{A} = 0$ lesz.

Két logikai változóval többfajta művelet végezhető el. Például az *“or”* vagy *“vagy”* művelet, ami akkor fog eredményül 1-t hozzárendelni a logikai változónkhoz, ha valamelyik változó tartalmazott 1-est. Például $A = 1, B = 0, ekkor\ A|B = 1$.

Az *“and”* vagy *“és”* művelet ezzel szemben, csak akkor rendel eredményül 1-et a változónkhoz, ha mindkét változó értéke 1 volt. Például $A = 1, B = 1, ekkor\ A\&B = 1$.

Ezeknek a műveleteknek az ellentétes megfelelőjüket vagy negáltjukat is használjuk, ezt úgy kell érteni, hogy amikor például az *"és"* művelet eredménye 0-át adna a *"nem és" vagy "nand"* esetén pont ennek a negáltját, vagyis 1-et fogunk kapni.

Fontos művelet még a *"XOR"* vagy *"kizáró vagy"* művelet, amely akkor rendel eredményül a változónkhoz 1-et, ha pontosan csak az egyik változó értéke volt 1. Például $A = 1, B = 0, ekkor\ A \otimes B = 1$.

A logikai műveletek felírhatóak igazságtábla formájában is, amelyeket megtekinthetsz a "Matematikai alapok ismétlése" című tananyagrészben.

SZÁMÁBRÁZOLÁS

Az informatikában a számok ábrázolása tipikusan binárisan történik, vagyis egy szám leírásában 0-ák és 1-esek szerepelnek. A **természetes számok**at könnyen le tudjuk írni, ha felírjuk magát a számot bináris alakban. De gondoljunk bele, hogy ha már egy egész számot szeretnék felírni, például a -3-at akkor ezt, hogyan is tudnánk megtenni, mivel nincsen "-" előjelünk csak 0-ás és 1-es számjegyeink?

Mielőtt ezt megválaszolnánk vezessünk be pár az informatikában használt fogalmat. Az informatikában az adat legkisebb, vagyis elemi egysége a **bit**, neve az angol **bi**nary digi**t** kifejezésből származik. Két értéket vehet fel 0-át vagy 1-et, de ehhez többfajta jelentést is társíthatunk. Például jelenthet logikai esetben igazat vagy hamisat, igent vagy nemet, de gondolhatunk rá úgy, mint egy előjelre ekkor "+"-t vagy "-"-t takar. Bárminek megfeleltethető, ami két állapottal leírható. A következő fontos fogalom a byte vagy bájt. Azt már biztos sokszor hallottátok, hogy 8 bit jelent 1 byte-ot, de miért is olyan fontos ez? Azért, mert általában a digitális információ alapegységeként szoktuk használni, például tipikusan a memóriákat byte pontossággal lehet megcímezni, az adatátvitel során is azt szokás megadni, hogy hány byte-t tudunk küldeni vagy fogadni egy másodperc leforgása alatt vagy a karaktereinket is byte-ban szoktuk ábrázolni, vagyis 8 bit-en.

Kitekintés

Karaktereink is egész számokként vannak lekódolva, az úgynevezett ASCII táblában. Ez egy mozaikszó, amely az American Standard Code for Information Interchange (szabványos amerikai kód információcserére) kezdőbetűiből tevődik össze. Az ASCII tábla tartalmazza az angol abc betűit, számokat, írásjeleket és vezérlő karaktereket. Ezek 7 bit-en elhelyezkedő előjel nélküli egészek, tehát 128 karakterből álló jelkészlet. Például az 'a' karakter kódja decimálisan a 97.

A különböző nyelvek támogatásához, ahol speciális angol abc-től különböző karakterek is előfordulnak, ezt a jelkészletet kibővítik a 8. bittel, így vagyunk képesek a magyar nyelvben megtalálható betűk írására is.

Az **egész számok**nál tapasztalt előjel problémát, tehát megoldja az, ha használunk úgynevezett előjel bit-et is. Ez azt jelenti, hogy a bináris számunk legelső bit-je, ha 0 akkor a számunk pozitív, ha 1 akkor pedig negatív lesz. Nézzünk rá egy példát: a *0b00001001* szám áll egy előjel bitből és adatbitekből. Az előjel bit-je a 0-ás, tehát pozitív szám lesz, majd ezt követi a *0001001* bitsorozat, ami nem más mint a 9, tehát a *0b00001001* bitsorozat a 9-et jelöli. Nézzük meg a *0b10001001* számot, az előzőhöz hasonlóan eljárva, rájöhetünk, hogy ez a -9-et fogja jelenteni.

Kitekintés

Előző leírási módnál az értékkészlet tagja lesz az úgynevezett negatív zérus is, illetve nem különül el a negatív és a pozitív tartomány teljesen. Kicsit bonyolultabb leírási mód a kettes komplemens. Ha a számunk pozitív egész, akkor az megeggyezik az előzőekben bemutatottal. Ha azonban negatív, az kettes komplemensbe úgy fog kinézni, hogy vesszük a pozitív megfelelőjének bitjeinek ellentétét és hozzáadunk még egyet. Ezáltal kiküszöböltük a negatív zérust, illetve teljesen elválasztottuk a pozitív és negatív tartományt egymástól. Ezzel a módszerrel továbbá a kivonást összeadásra vezettük vissza, így kevésbbé bonyolult ALU-kat (aritmetikai logikai egység) kell tervezni. Ezért is máig ez a legalapvetőbb ábrázolási mód. Nézzük meg hogyan néz ki a -9 kettes komplemensben. A 9-et a 0b00001001 bitsorozat írta le, vegyük minden bitnek az ellentétét, ekkor a 0b11110110 bitsorozatot kapjuk. Ehhez adjunk hozzá még 1-et és meg is kaptuk a kettes komplemensben ábrázolt -9-et, ami nem lesz más, mint a 0b11110111.

Mi van akkor, ha egy törtszámot szeretnénk leírni, azt hogyan tudnánk megtenni? A törtszámok az úgynevezett **fix pontos számábrázolás**sal írhatóak le. Ez azt jelenti, hogy előre definiálva, rögzítve van az ábrázolásnál a tizedes (vagy kettedes, mivel bináris számrendszer) pont helye. Például szeretnénk leírni fix pontos számábrázolással a 9,5-öt úgy, hogy 1 bit törtrész és 4 bit egész részből álljon. Ezt úgy tudjuk megtenni, hogy vesszük az egészrészt, tehát a 9-et. Azt láttuk, hogy az 1001-ként jeleníthető meg. Ezt követően a törtrészt is megnézem ez 0,5, ami 2-nek pont a -1-dik hatványa. Így a 9,5-öt *0b10011*-ként tudom ábrázolni, ahol tudjuk, hogy az utolsó bit a törtrészt jelenti, mivel előre rögzítettük azt. Ha ugyanezt a számot szeretnénk 2 bit törtrésszel leírni mit kaptunk volna? A helyes megoldás a *0b100110*, mivel az utolsó két bit jelenti a törtrészt, de nekünk 2-nek csak a -1-dik hatványára van szükségünk a -2-dik hatványára nem, így annak helyére 0 kerül.

Továbblépve még egyet, mi van akkor ha azt a számot, amit ábrázolni szeretnék nem tudjuk leírni törtszámként, csak valós számként. A **valós szám**ok ábrázolását a **lebegőpontos számábrázolás** teszi lehetővé. Így az előzőekhez képest egy jóval szélesebb tartományt vagyunk képesek lefedni. Az ábrázolásmód a nevét onnan kapta, hogy a tizedes pont nem fix helyen szerepel, hanem “lebeg”, bárhova kerülhet. A következő formában tároljuk el a számunkat:

$$\textit{értékes számjegy} \times \textit{alap}^{\textit{exponens}}$$

Ez azt jelenti, hogy az értékes számjegyünk skálázva van egy exponens, vagyis kitevő segítségével, ahol az alap informatikábban tipikusan a 2. Nézzünk erre is egy példát, most a jobb érthetőség érdekében alapnak a 10-et válasszuk és írjuk fel a 300-at különböző exponensekkel. Ha az exponenst 1-nek választom akkor a következőképpen nézne ki: $30 \cdot 10^1$, ha 2-nek akkor pedig $3 \cdot 10^2$-vel írhatjuk le.

BYTE SORREND

Megismertük, hogy adataink byte-okban tárolódnak el a memóriában. Felvetődhet a kérdés, hogyan tárolunk el több byte-ból álló adatokat. A byte sorrend vagy az angol kifejezés szerint endianness fogja meghatározni, hogy hová fog kerülni a legnagyobb helyiértékű byte (angolul Most Significant Byte, rövidítve MSB). Alapvetően két elterjedt reprezentáció létezik, a big-endian és a little-endian. A big-endian esetén a nagy elől elv érvényesül, vagyis a legalacsonyabb memóriacímen fog tárolódni az MSB, míg little-endian esetén pont fordítva, az MSB fog a legmagasabb memória címre kerülni.

Például a hexadecimálisan felírt 16 bites számunk legyen a következő: *0x1A2B*. Ezt a számot a memóriában a *100-as* címtől kezdve szeretnénk eltárolni, mivel a memóriánkat byte-onként tudjuk címezni ennek a tárolásához két címre lesz szükségünk. Big-endian esetén a *100-as* címen fog eltárolódni a *0x1A* és a *101-*

es címre pedig a *0x2B* byte fog kerülni. Little-endian esetén pont fordítva történik, a *100-as* címen lesz a *0x2B* és az MSB a magasabb címre, vagyis a *101-es* címre fog kerülni. Az endianness jelentőségével például olyankor lehet találkozni, amikor a mikrokontroller kommunikál egy külső eszközzel, hiszen olyan bytesorrendben kell küldeni majd az adatokat, ahogyan azt az aktuális kommunikációs protokoll megkívánja.

Kitekintés

Megjegyzendő, hogy nem igazán lehet eldönteni melyik a jobb, nincs jelentős előny amely az egyik mellett szólna. A világon talán a little-endian van túlsúlyban, mivel az Intel x86 alapú processzorai ezt a byte sorrendet preferálják.

BEVEZETÉS A PROGRAMOZÁSBA

Amikor kapunk egy feladatot, általában csak annyit tudunk, hogy mire mit kellene csinálnia a gépnek. Azt, hogy a gép ezt hogyan hajtsa végre azt nekünk kell kitalálnunk, megterveznünk és mindezt megmondanunk neki. De hogyan is mondjuk meg, hogy mikor mit tegyen?

A gépek rendelkeznek úgynevezett utasításkészlettel, ez azoknak a műveleteknek összessége, amelyeket ő ismer. Mi utasításokat adunk neki a saját gépi nyelvén keresztül. Mivel a gépek binárisan látják a világot ez 1-esek és 0-sok sorozatát jelenti. Egy programot megvalósító 0-ák és 1-esek sorozatát nevezzük gépi kódnak. Ez emberi szemnek elég bonyolult lenne és nehezen tudnánk benne hatékonyan dolgozni (nem mindenki képes arra, mint ami a Mátrix című filmben látható).

Az emberi szemnek már egy kicsit olvashatóbb forma, ha az utasításokat szövegesen adnánk meg. Ilyen formában tudjuk megadni az utasításokat az “Assembly nyelv”-ben. Például ilyen az “ADD” utasítás, aminek a jelentése, hogy adj össze valamit. Assemblyben írt kódokban szereplő szavakat a fordító (az a szoftver elem, amely a kódunkból létrehozza a már gépek által is értelmezhető kódot) lényegében csupán lecseréli ezeket az utasításnak megfelelő bit-sorozatra. Belátható, hogy ez sem lenne túl kényelmes még számunkra, mert egyszerűbb feladatokat is csak nehezen átlátható, sok kis lépésből álló, ezért hosszú programokkal tudnánk csak megvalósítani.

Az előzőek helyett a C nyelvet fogjuk megismerni, amelyet széles körben használnak, annak általánosan használható jellege miatt, illetve mert ez a nyelv még kellően alacsony szintű (hardverhez közeli), hogy számunkra is megfelelő legyen. A C nyelven megírt programok Assembly nyelvre fordulnak le, majd pedig abból készülnek el a gépi kódok. A gépi kódról és a fordítás folyamatáról kicsit részletesebben majd egy későbbi tananyagrészben olvashatsz, most elégedjünk meg ennyivel.

Mielőtt elkezdenénk a C nyelvvel megismerkedni, fontos tisztázni néhány fogalmat.

Az **algoritmus** jelenti magát a megoldást egy feladatra, ez magában foglalja az egymás után elvégzendő utasításokat. Például a szendvics készítés algoritmusa lehetne például ez:

1. Vedd elő a kenyeret!
2. Vágj le belőle egy szeletet!
3. Majd vajazd is meg!

4. Tegyél rá szalámit!
5. Tegyél rá sajtot!
6. Kész is vagy!

Ezen az egyszerű példán keresztül látható, hogy az utasítások sorozata alkotja az algoritmust, illetve hogy meghatározott lépést követően el tudunk készülni a szendvicsünkkel.

Kitekintés

A fent leírtak szerint egy feladatot úgy oldunk meg, hogy annak megoldását mi találjuk ki és lépésről lépésre adjuk át azt a gépnek, amely végig is fogja hajtani azt. Ez az úgynevezett imperatív programozás. A C nyelv is egy ilyen imperatív nyelv.

Léteznek deklaratív programozás nyelvek is. Az előzőtől eltérően ebben az esetben azt kell megadnunk, hogy milyenek a helyes megoldások, de a megoldás módját majd a gép fogja saját magának kitalálni. Így egy feladatot akár könnyebben is le lehet programozni, de a felhasználhatóság köre is sokkal szűkebb, mint az imperatív nyelveknek. Ilyen deklaratív nyelv például tipikusan az adatbázisoknál használt nyelvek, ahol például elég csak azt megmondanunk milyen adatokat keresünk és majd a program maga fogja kitalálni, hogyan oldja majd ezt meg.

Következő fontos fogalom a **változó**. Változókat el tudjuk látni névvel, amellyel később könnyen meg tudjuk őket találni, hivatkozni tudunk rájuk és bennük értékeket tudunk tárolni. Használatukkor az eltárolt értéket ki lehet olvasni vagy új értékre cserélni a régit.

Általában a változóink rendelkeznek egy **típus**sal, amely az értékkészletük és a rajtuk végezhető műveletek együttesét jelenti. Típus például az egész számok, ahol az értékkészletben helyet foglalnak például a -1, 2, 6 számok és a velük elvégezhető műveletek, mint például az összeadás, kivonás.

Mint minden nyelv a programozási nyelvek is rendelkeznek nyelvi elemekkel és szabályokkal vagy szintaktikával (szintaxissal), amelyeket meg kell tanulnunk ugyanúgy, mint mikor például angolul tanulunk. Szerencsére a fejlesztőkörnyezetek, amelyekben a kódokat általában készítjük a segítségünkre vannak és jelzik, ha szintaktikai hibát vétettünk és addig amíg hiba van a kódunkban nem tudjuk a gép által érthető nyelvre lefordítani.

A C NYELV

Elérkeztünk, ahhoz a ponthoz amikor elkezdünk megismerni egy programnyelvet. A C nyelv egy erősen típusos programozási nyelv, ez azt jelenti, hogy minden változóról meg kell mondanunk, hogy milyen típusú.

Egy változó létrehozásakor tehát meg kell mondanunk annak a típusát, a nevét, majd pedig kezdeti értéket is adhatunk neki, más szóval inicializálhatjuk. A kezdeti értékadás nem feltétlenül szükséges. A példában egy **int** (integer röviden, később lesz szó részletesen a különböző típusokról) típusú változót hozunk létre *“a”* néven és egy másikat *“alma”* néven, majd kezdőértéket is adunk nekik. Fontos a sor végén található pontosvessző, ami az utasítás végét jelzi.

```
int a = 3;
```

```
int alma = -1;
```

A példában az “int” típus egyben egy **kulcsszó** is. Kulcsszavakkal tudunk hivatkozni a különböző nyelvi elemekre. Változónévnek érdemes olyan nevet választani, ami utal arra, hogy milyen célra hoztuk azt létre. Például a kör sugarának hosszát tartalmazó változónkat célszerű például `int sugar` vagy `int R` néven létrehoznunk, így később is biztosan fogjuk tudni mit jelent a benne tárolt érték.

ALAPTÍPUSOK

Nézzük most meg milyen alaptípusok vannak a C nyelvben. Az előző példában láttuk már az **int** típust. Ennek a mérete tipikusan 32 bit, ez azt jelenti, hogy az értékkészlete -2^{31}-től egészen $2^{31}-1$-ig terjed. A változóink alapértelmezetten előjelesek (**signed** angolul, ez egyben a kulcsszó is hozzá), de ha mi előjel nélkül szeretnénk létrehozni őket akkor az **unsigned** kulcsszót kell a típus elé helyeznünk. Ekkor az értékkészlete 0-től egészen $2^{32}-1$-ig fog terjedni. Létrehozása a következőképp történik.

```
unsigned int a = 102;
```

Az integer-ből létezik rövidebb, 16 bites és hosszabb, 64 bites változat is, előbbire a kulcsszó a **short**, míg utóbbira a **long**.

A következő fontos típus a **char**, amely karakterek tárolására szolgál. Ez egy 8 biten tárolt változó, tehát az értékkészlete előjeles esetben -128-től egészen 127-ig terjed, míg előjel nélküli esetben (**unsigned char**) 0-től egészen 255-ig terjed. A karaktereket **‘ ‘** jelek között adhatjuk meg, például:

```
char betu = 'a';
```

Ahogy láthattuk, ezek a típusok egészpontosságúak. A lebegőpontos ábrázolásra (így hívják számítástechnikában a törteket) a **float** és a **double** típusok szolgálnak. A float 6 tizedes pontossággal, míg a double (dupla pontosság) 15 tizedes pontossággal rendelkezik. Természetesen a double tárhely igénye ennek megfelelően magasabb 64 bit, míg a float típusú változó 32 bit-en tárolódik el. Létrehozásuk a következőképpen történik:

```
float kerulet = 25.7;
double terulet = 12.342;
```

A következő táblázat összefoglalja a nyelvben lévő alaptípusokat.

Adattípus:	Értékkészlet:	Méret:	Pontosság (tizedesjegy):
char	$-128 \dots 127$	8 bit	
unsigned char	$0 \dots 255$	8 bit	
int	$-2^{31} \dots 2^{31}-1$	32 bit	
unsigned int	$0 \dots 2^{32}-1$	32 bit	
short int	$-32\,768 \dots 32\,767$	16 bit	

unsigned short int	$0 \dots 65\,535$	16 bit	
long int	$-2^{63} \dots -2^{63} - 1$	64 bit	
unsigned long int	$0 \dots 2^{64} - 1$	64 bit	
float	$-3.4 \cdot 10^{38} \dots 3.4 \cdot 10^{38}$	32 bit	6
double	$-1.7 \cdot 10^{308} \dots 1.7 \cdot 10^{308}$	64 bit	15

Természetesen léteznek másmilyen típusok is, ezek használatához azonban hozzá kell rendelnünk egy külső fájlt a programunkhoz. Ilyen például a **bool** típus, amelynek az értékkészletében a *true* és a *false* szavak szerepelnek, magyarul ezek igazat és hamisat jelentenek. Használatához az *stdbool.h* fájlt kell a programunkhoz rendelnünk. Logikai változóknál célszerű használnunk például ezt a típust.

A későbbiekben olyan típusokkal is találkozhatsz, mint például **uint8_t**. Ettől ne ijedj meg, ez csupán azt fogja jelenti, hogy előjel nélküli (u = unsigned) 8 bites integer változóról van szó. Sok esetben nincs szükségünk a 32 bit által lefedett számtartományra, elég ennek csak egy töredéke, így gazdaságosan bánhatunk a memóriánkkal, amely egy mikrokontroller esetén még akár jól is jöhet.

Mi a helyzet akkor, ha például több egymáshoz kapcsolódó értéket szeretnék eltárolni? Mindegyiknek hozzunk létre külön nevű változót? Ez elég kényelmetlen lenne, ilyenkor vannak segítségünkre az úgynevezett tömbök. Egy tömbbel több azonos típusú értéket el tudunk tárolni. Létrehozásukhoz meg kell mondanunk hány elemű legyen, például int terulet[3]; így a terulet nevű változónk 3 integer értéket fog tudni eltárolni. Hivatkozni az egyes elemekre azok indexével, vagyis sorszámával tudunk. A tömbünk első eleme a 0-ás indexű elem (általában a legelső elem az informatikában 0-ás indexel van ellátva), míg a 3. elemhez tartozik a 2-es index.

```
int szam[3];
szam[0] = 12;
szam[1] = -2;
szam[2] = 4;
```

Vagy így is megadhatjuk a kezdőértékeket:

```
int szam[3] = {12, -2, 4};
```

Láttuk, hogyan tudunk karaktereket eltárolni, azonban egy szöveg esetén már kényelmetlen lenne azt karakterenként megadnunk. Ezért szövegeket **string**ekben tudunk könnyen eltárolni. A stringek hasonlítanak a tömbökhöz, lényegében a karakterek egy tömbben tárolódnak el azzal a különbséggel, hogy a stringek utolsó eleme minden esetben a **'\0'** karakter lesz, ez fogja jelezni a string végét. A következőképp tudunk létrehozni egy stringet.

```
char szoveg[] = "Hello";
```

Ez egyenértékű a következővel

```
char szoveg[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Előfordulhat, hogy egy változó értékét létrehozása után már nem szeretnénk módosítani, állandóvá, **konstans**á szeretnénk tenni. Erre is van lehetőségünk, a **const** kulcsszóval tudjuk ezt elérni változó létrehozáskor, például:

```
const int PI = 3.14;
```

Fontos, hogy ilyen esetben a konstans létrehozásakor neki értéket is adjunk, mivel később már erre nincs lehetőségünk.

A C fordítók rendelkeznek optimalizációs funkcióval is. Tehát a kódunkat ahol tudják optimalizálják, egyszerűsítik, hatékonyabbá próbálják tenni. Az optimalizálást fokát mi magunk tudjuk kiválasztani, akár ki is tudjuk azt kapcsolni.

Az optimalizáció néha ellenünk dolgozik, például azt látjuk, hogy egy optimalizáció nélkül addig jó működő kód az optimalizációt követően már nem az elvártak szerint viselkedik. A fordító, és ezzel az optimalizáció egyszerre csak egy fájlban dolgozik. Erre az egyik legegyszerűbb példa, ha egy változó az adott fájlban nem módosulhat, akkor a fordító egy állandóval helyettesíti, tehát így optimalizálja a kódot. Ez viszont helytelen működést eredményez, ha a változó értéke egy másik fájlban (/fordítási egységben) módosulhat. Ilyen esetekben nekünk kell a fordító felé ezt jeleznünk, amit a változó típusa elé tett **volatile** kulcsszóval tudunk megtenni.

Operátorok

Az operátorok vagy más néven műveleti jelek segítségével tudunk egy kifejezést (amelyet operátor(ok) és operandus(ok) alkotnak) leírni. A következő táblázat mutat pár példát kifejezésekre.

Kifejezés:	Érték:	Magyarázat:
3+2	5	A 3 és a 2 az operandusok az operátor maga az összeadás
2*(3+1)	8	Az előzőhöz hasonlóan az operandusok a 2,3 és az 1, míg az operátorok a *,+ és a (). Összeadás, majd ezt követően összeszorzás
2<5	Igaz	Logikai kifejezés, összehasonlító operátorral. A kettő tényleg kisebb, mint az 5, igazra értékelődik ki

Sokfajta operátor létezik, ezek között úgy mint matematikában van műveleti sorrend, idegen szóval precedencia. A következő táblázat a C nyelv operandusait mutatja precedencia sorrendben. Ne ijedj meg, ha elsőre nem világos, idővel majd amikor elkezded jobban megismerni és használni a C nyelvet, meg fogod érteni, hogy mi mit jelent. A későbbi tananyagban használt és fontos operátorokat kiemelve láthatod.

Precedencia	Operátor	Leírás	Asszociativitás
1	**++ --**	**Suffix/postfix inkrementálás dekrementálás**	Balról-jobbra
	()	**Függvény hívás**	
	`[]`	Tömb indexelés	
	`.`	Struktúra és unió adattag elérés	
	`->`	Struktúra és unió adattag elérés pointeren keresztül	
	(*típus*){*lista*}	Nyílt lista (Compound literal) (C99)	
2	**++ --**	**Prefix inkrementálás dekrementálás**	Jobbról-barra
	+ -	**Unáris plusz és mínusz**	
	! ~	**Logikai NEM és bitenkénti NEM**	
	(*típus*)	Típus kasztolás	
	`*`	Indirection (dereference)	
	`&`	Címe (Address-of)	
	`sizeof`	Mérete (Size-of)	
	`_Alignof`	Illesztés (Alignment requirement)(C11)	
3	*** / %**	**Szorzás, osztás, és maradék**	Balról-jobbra
4	**+ -**	**Összeadás és kivonás**	
5	**<< >>**	**Bitenkénti balra és jobbra shiftelés(tolás)**	
6	**< <=**	**Relációs műveletekhez < és ≤ operátor**	
	> >=	**Relációs műveletekhez > és ≥ operátor**	
7	**== !=**	**Relációs műveletekhez = és ≠ operátor**	
8	**&**	**Bitenkénti ÉS**	
9	**^**	**Bitenkénti XOR (kizáró VAGY)**	
10	**|**	**Bitenkénti VAGY**	
11	**&&**	**Logikai ÉS**	
12	**||**	**Logikai VAGY**	
13	`?:`	Feltételes operátor	Jobbról-barra

		`kifejezes ? igaz : hamis`
		
		`if (kifejezes)`
		`  igaz;`
		`else`
		`  hamis;`	

14	**=**	**Értékadás**	
	+= -=	**Értékadás összeadással és kivonással**	
	`*= /= %=`	Értékadás szorzás, osztás és maradékrész művelettel	
	`<<= >>=`	Értékadás bitenkénti balra vagy jobbra shifteléssel	
	`&= ^= |=`	Értékadás bitenkénti ÉS, XOR, vagy VAGY művelettel	
15	**,**	**Vessző**	Balról-jobbra

A C NYELV UTASÍTÁSAI

Már az eddigiek során is megismertünk sokféle utasítást, gondoljunk csak arra amikor létrehoztunk egy változót az is egyfajta utasítás volt vagy a kifejezésekre amiket megnéztünk. Az utasítások lezárására a **";"**, vagyis a pontosvessző szolgált, azonban ha ez csak egymagában áll akkor az egy üres utasítást jelöl.

Nézzünk meg most további elemeket is. Több utasítást úgynevezett utasítás blokkban helyezünk el, amelyet kapcsos zárójelek **{}** határolnak. Ez az utasítás blokk így lényegében egy összetett utasítást fog jelenteni és a következőképpen néz ki.

```
{
  utasítás1;
  utasítás2;
  ...
  utasításN;
}
```

Nézzünk meg egy konkrét példát, amelyben létrehozunk három változót, majd ezeknek értéket adunk.

```
{ //utasításblokk kezdete
  int a,b,c; //három integer változó létrehozása
  a = 2; //értéket adunk az "a" változónknak
  b = 3; //értéket adunk a "b" változónknak
  c = a*b; //összeszorozzuk "a"-t és "b"-t,
           amit "c"-ben fogunk eltárolni
  //"c" változó ezáltal 6-ot fogja tartalmazni, ez is egy értékadás
} //utasításblokk vége
```

A kódunkat elláthatjuk kommentekkel, amelyek segítik a kód megértését. Kétféleképpen jelölhetjük a kommenteket. A **//** jelet követően csak abban a sorban tudunk elhelyezni kommentet, míg a **/*** és ***/** jelek közé írt szövegek akár több soron keresztül is átívelhetnek. A kommentek nem kerülnek be a gépi kódba, ezek csak számunkra nyújtanak információt. A fordító ezeket a kommenteket nem veszi figyelembe, egyszerűen csak kihagyja a fordítási folyamat során.

Elágazások

Az eddig megismertek alapján az utasítások megadott sorrendben egymás után hajtódnak végre, ezt nevezzük szekvenciális végrehajtásnak.

De mi van akkor, ha nekünk valamilyen feltételhez kellene kötnünk egy vagy több utasítás lefutását? Erre nyújtanak megoldást az **elágazás**ok vagy feltételes elágazások. Több ilyen szerkezet is rendelkezésünkre áll, most megnézzük ezeket.

Az első szerkezet az úgynevezett **if()** elágazás (magyarul azt jelenti, hogy **ha**). Akkor használjuk, ha feltételhez szeretnénk kötni egy utasítás vagy utasításblokk lefutását. A szintaxis a következőképp néz ki.

```
if(feltétel)
{
  utasítás;
}
```

Ez azt jelenti, hogy az utasítás csak akkor fog végrehajtódni, ha az *if()* hasában lévő feltétel teljesül, máskülönben ez az egész blokk kimarad, nem hajtódik végre. Egy C-ben megvalósított példát mutat a következő, amelyben az *if()* elágazásba be fogunk lépni, mivel a feltételünk teljesülni fog.

```
int szam = 2;    //a példa kedvéért létrehozott változó

// if() elágazás bemutatása
if(szam < 3) //ha a "szam" értéke kisebb mint 3,
             belépünk az elágazásba (2 < 3 ez igaz)
{
  szam = szam + 5; //"szam"-hoz hozzáadunk 5-öt
}
```

A *szam* változóban tárolt érték kisebb mint 3. Így az elágazásunkban található utasítás végre fog hajtódni és a *szam* változónkhoz hozzá fogunk adni 5-öt. Annak értéke az elágazás végén 7 lesz. Ha a *szam* változónk értéke kezdetben például 4 lenne, akkor a feltételünk már nem teljesülne és az elágazást átugornánk, így nem növelnénk meg a változót 5-el. A matematikában jártasoknak bizonyára feltűnt, hogy a feltételes blokkban található sornak matematikai szempontból nincs értelme, hiszen valami nem lehet egyenlő valami + 5-el. Programozás során itt kicsit félre kell tennünk a matematikát, és értékadó utasításként kell értelmeznünk az egyenlőségjelet. Az egyenlőségjel bal oldalán az áll, hogy a jobb oldalra írt műveletsor eredményét hol szeretnénk tárolni - ez most a *szam* változó, míg a jobb oldalon szereplő *szam* a változó aktuális értékére utal. Az utasítást szövegesen úgy lehet elképzelni, hogy “vedd a *szam* változó aktuális értékét és adj hozzá ötöt (jobb oldal), majd az így kapott eredményt mentsd el a *szam* változóba (bal oldal)”. Így az utasítás végeredménye az, hogy a változó értékét megnöveltük öttel.

Az if-else szerkezet lényegében az előzőleg bemutatottnak a kiegészítése. Az *if* ág mellett bekerült most egy *else* ág is (magyarul azt jelenti, hogy különben). Az általános felépítése a következő.

```
if(feltétel)
{
  utasítás;
}
else
{
  utasítás;
}
```

Az előzőhöz hasonlóan, ha az `if()` hasában lévő feltétel teljesül, akkor az `if` ágon lévő utasítások fognak végrehajtódni. Egyéb esetben a másik ág fog szerepet kapni, vagyis az `else` ág. Az ott szereplő utasításblokk fog ekkor érvényre jutni és végrehajtódni. A következő C-ben megírt példa az előzőnek a kibővítését mutatja.

```
int szam = 6;  //a példa kedvéért létrehozott változó

// if-else elágazás bemutatása
if(szam < 3)    //ha a "szam" értéke kisebb, mint 3 belépünk
                (6 < 3 ez hamis)
{
  szam = szam + 5; //"szam"-hoz hozzáadunk 5-öt
}
else   //egyébként, ha "szam" nem volt kisebb, mint 3 ide lépünk
{
  szam = szam - 2; //"szam" változónkból kivonunk 2-őt
}
```

A `szam` változónk értékét most 6-ra állítottuk, így mikor megvizsgáljuk a feltételünket hamisat kapunk, nem teljesül, mivel 6 nagyobb, mint a 3. Emiatt az `else` ágban lévő utasítás fog végrehajtódni. A `szam` változónkból elveszünk 2-őt, végeredményben a `szam` változónkban az elágazást követően 6-2, azaz 4 fog szerepelni.

Előfordulhat, hogy több elágazásra lenne szükségünk, mert például egy változó értékétől függően kell majd valamit elvégeznünk. Erre használható a **switch-case** szerkezet. Általános szintaxisa a következő:

```
switch(egész típusú érték)
{
  case érték1: utasítások;
  case érték2: utasítások;
  case érték3: utasítások;
  ...
  case értékN: utasítások;
  default: utasítások;
}
```

Működéskor a *switch()* hasában elhelyezett egész típusú értéknek megfelelő *case* ágba fogunk lépni. Fontos, hogy az egyes *case* ágakat nem követik kapcsos zárójelek. Megadhatunk továbbá egy úgynevezett **default** ágat is, amelyet akkor szeretnénk, hogy lefusson, ha a *switch()* szerkezetnek átadott érték egyik ágban szereplő értéknek sem felel meg. A *default* ág használata nem kötelező.

Nézzünk meg a következő C-s példát a switch szerkezet működésének megértéséhez.

```
int szam = 0;  //a példa kedvéért létrehozott változó

//switch-case bemutatása
int ugrik = 2; //ennek a változó értékének megfelelő ágba fogunk ugrani

switch(ugrik)   //"ugrik" változó értékének megfelelő ágba ugrik
{
  case 1: szam = szam - 5; //ide lépünk, ha az "ugrik" értéke 1 volt
  case 2: szam = szam + 2; //ide lépünk, ha az "ugrik" értéke 2 volt
  case 3: szam = szam * 3; //ide lépünk, ha az "ugrik" értéke 3 volt
}
```

A *switch()* hasában az *ugrik* szó olvasható, ez annyit fog jelenteni, hogy az *ugrik* változó értékének megfelelő ágba fogunk lépni. Jelen esetben a *case 2* ágba, mivel az *ugrik* változónak a 2-es értéket adtuk. A *case 2* ágban a *szam* változónkhoz hozzá adunk 2-őt, így annak értéke 2-re módosul, ezt követően haladunk tovább a soron következő ágakkal.

A *case 3* ágban megszorozzuk a *szam* változónkat még 3-mal, így a *switch()* végére a *szam* változónk értéke már 6 lesz. Tehát lényegében a *case 1* ágat ugrottuk át a példánkban. Kicsit nehezen átlátható ez, hogy nem csak a *case 2* ág fut le, hanem az őt követő többi is. Gondoljunk bele ez egy nehezen átlátható kódot és bonyolult logikát eredményezne sok ág esetén. Általában csak azt szeretnénk, hogy egy bizonyos ág hajtódjon végre.

Ezt a **break** kulcsszó segítségével tehetjük meg. Amikor elérünk ehhez az úgynevezett **ugró utasítás**hoz, az elágazásunkat megszakítjuk és azt látjuk, mintha kiugrottunk volna belőle. Ezt a kulcsszót az előbb bemutatott és a későbbiekben bemutatásra kerülő szerkezeteknél is használhatjuk természetesen, bár megjegyzendő, hogy azoknál pont nehezebb átláthatóságot fog eredményezni a használata.

Nézzük meg az előző példát, de már a megismert *break* utasítással és *default* ággal ellátva.

```
int szam = 0;  //a példa kedvéért létrehozott változó

//switch-case bemutatása
int ugrik = 2; //ennek a változó értékének megfelelő ágba
               fogunk majd ugrani

switch(ugrik)  //"ugrik" változó értékének megfelelő ágba ugrik
{
  case 1: szam = szam - 5;  //ide ugrik, ha az "ugrik" értéke 1 volt
    break;
  case 2: szam = szam + 2;  //ide ugrik, ha az "ugrik" értéke 2 volt
```

```
    break;   //break miatt kiugrunk a switch szerkezetből
  case 3: szam = szam * 3;  //ide ugrik, ha az "ugrik" értéke 3 volt
    break;
  default: szam = 0;  //ide ugrik, ha az "ugrik" értéke az
                      előzőektől eltérő volt
    break;
}
```

Az előzőekhez hasonlóan a *case 2* ágba fogunk lépni és a *szam* változónkat megnöveljük 2-vel. Majd ezt követően a *break* utasítás hatására a switch szerkezetből kilépünk, így az előzőhöz képest nem fognak a következő ágak is végrehajtódni.

Ha az ugrik változó értékének kezdetben például 4-et választunk, akkor a default ágra lépünk, ahol is a szam változónk értékét kinullázzuk.

Ciklusok

Sokszor megesik, hogy egy utasítás blokkot többször szeretnénk végrehajtani. Lehet ez akár egy meghatározott számú ismétlődés vagy csak valamilyen feltétel teljesülése. Erre valók a **ciklusok**. Az első, amit megnézünk az a **while()** ciklus. Ennek a szintaktikája a következő:

```
while(feltétel)
{
   utasítás;
}
```

A ciklusunk addig fog futni és így a benne lévő utasítás(ok) annyiszor fognak újra és újra végrehajtódni, ameddig a *while()* hasában lévő feltétel teljesül. Nézzünk meg rá egy egyszerű példát C-ben megvalósítva.

```
int x = 0; //a példa kedvéért létrehozott változó

// while() ciklus bemutatása
while(x < 2) //Feltétel megvizsgálása, ameddig igaz újra és újra
belépünk
{ //while ciklus utasításblokkjának kezdete
  x++; //"x" változó növelése 1-el, x=x+1-et is írhattunk volna
} //while ciklus utasításblokkjának vége
```

Először létrehoztunk egy integer típusú változót *x* néven, amelynek 0 kezdőértéket adtunk. A *while()* ciklus a feltétel vizsgálattal kezdődik, ami ha igazra értékelődik ki, akkor végrehajtódik az utasításblokk. Az *x* kezdőértéke 0, tehát 0<2 kifejezés igaz, belépünk az utasításblokkba. Itt *x* értékét megnöveljük 1-el, más szóval inkrementáljuk, így *x* értéke már 1 lesz. Az utasításblokk végét követően visszaugrunk a *while* ciklus feltételének vizsgálatára. Emlékezzünk vissza, azt tanultuk, hogy annyiszor fog lefutni a ciklusunk, amíg igaz marad a feltétel. Most 1<2 még mindig igaz, ezért ismét belépünk az utasításblokkba, megnöveljük az *x-et*, amelynek az értéke így már 2 lesz. Ezt követően ismét visszaugrunk a *while* ciklus feltétel vizsgálatára. Az *x* értéke ekkor már 2, így a 2<2 kifejezés hamisra értékelődik ki, a programunk a

while ciklus utáni részre ugrik. Láthattuk, hogy ebben az egyszerű példában egymás után 2-szer hajtódott végre a *while()* ciklus függvénytörzse.

A **for()** ciklus, egy számlálós ciklus, amely működésében nem különbözik az előbb bemutatott *while()* ciklusétól, a kettő egymásba át is írható, meg is feleltethetőek egymásnak. A *for()* szerkezet szintaktikája a következő:

```
for(inicializálás; feltétel; léptetés)
{
  utasítás;
}
```

Az *inicializálás* alatt egy változónak kezdőértéket adunk, ezt fogjuk a ciklusunk számlálójaként használni. A változót elég itt létrehozni, nem szükséges a szerkezet előtt már léteznie. Középen található a *feltétel*, ami ugyanazt jelenti, mint az előző szerkezeteknél, amíg ez teljesül addig újra és újra be fogunk lépni a ciklusba. A harmadik tag a *léptetés*, itt valósítjuk meg a ciklus számláló léptetését. Nézzük meg a következő egyszerű példát a leírtak szemléltetése érdekében.

```
int x[10];   //a példa kedvéért létrehozott tömb

//for() ciklus bemutatása
for(int i = 0; i < 10; i++) //a ciklus számlálónk inicializálása,
                            feltétel, majd léptetés leírása
{
  x[i] = i * 7;  //"x" tömb i-edik elemének értékadása
} //for ciklus utasításblokkjának vége
```

A példa kedvéért létrehozunk egy *x* nevű tömböt, majd ezt követi a *for()* ciklus. A ciklusunk. A *for()* ciklus zárójelei között először a számlálót létrehozzuk, ezt *i-nek* nevezzük, kezdeti értékét 0-ra állítjuk, majd megadjuk a feltételt. Most a feltételünk az, hogy *i<10*, vagyis addig fusson a ciklusunk, amíg *i* kisebb, mint 10. A zárójel utolsó tagja a számlálónk léptetését írja le. Ez az i++ fogja jelenteni, hogy minden ciklus végén növekszik a számlálónk értéke 1-el.

Nézzük meg a működését: az inicializálást követően *i* értéke 0, ami teljesíti a feltételt, tehát az *x* tömbünk i-edik, azaz 0 indexű elemének értéket adunk. Ez az érték nem más, mint i*7, jelen esetben 0*7=0. A *for()* ciklus utasításblokkja végén, amikor már minden benne lévő utasítás végrehajtódott, de még a következő ciklus feltétel vizsgálata előtt fog megtörténni a számlálónk növelése 1-el (*i++*). Ezt követően ismét kiértékelődik a feltételünk, de már *i=1* értékkel. Ez még mindig kisebb, mint tíz, így le fut ismét a *for()* ciklusunk, majd a végén megnövelődik a számlálónk, aminek az értéke így már 2 lesz. Az *x* tömb 1-es indexű elemének értéke 1*7-re módosult. A következő feltétel vizsgálatnál 2<10 fog kiértékelődni, így *x[2]* értéke 14 lesz. Ez egészen x[9]=9*7-ig fog menni. Ezután *i* értéke már 10 lesz, és nem lépünk bele a *for()* ciklusba. Ezzel a ciklussal az *x* tömbben előállítottuk a 7 többszöröseit 0-tól 63-ig.

Már megismertünk egy ugró utasítást ez volt a **break**. Most megismerkedünk egy másik nagyon fontos ugró utasítással is, ami nem más, mint a **return**. Ennek segítségével fogjuk tudni majd a programunkat befejezni, illetve függvényeinkből (kicsit lejjebb majd látjuk mik is ezek) visszatérni. Kétféleképpen

szerepelhet: állhat magában `return;` vagy egy kifejezés is követheti `return kifejezés;` Függvények esetében például a függvény kimenetét tudjuk vele visszaadni.

Függvények

A függvények a programozásban is hasonló jelentéssel bírnak, mint matematikában. Valami kapcsolatot, funkciót írnak le. A függvényeket nevezhetjük alprogramoknak is. Általában olyan utasítás sorozatokat foglalunk függvényekbe, amit a programunk során többször is használunk, így téve átláthatóbbá és hatékonyabbá is a kódunkat. A következőképp épül fel egy függvény:

```
visszatérési_típus függvénynév(paraméterlista)
{
 ...függvénytörzs...
 return visszatérési_érték;
}
```

Minden függvény rendelkezik egy névvel, amivel később el tudjuk érni. A függvény neve előtt szerepel a visszatérési értékének a típusa. Továbbá a függvények rendelkeznek paraméterlistával is, amin keresztül tudunk bemenő változókat átadni. A függvénytörzsben maguk az utasítások foglalnak helyet és minden függvény a *return* utasítással végződik, amely a függvény kimenetével tér vissza. Nézzünk rá egyszerű példát:

```
/*Ez a függvény két számot képes összeadni
  Két bemenő paraméterünk van, az első és a második szám.
  Eredményül egy int típusú értéket várunk, magát az összeget
*/
int osszead(int elso_szam, int masodik_szam)
{
  //osszeg nevű változó létrehozása, az eredményt itt fogjuk eltárolni
  int osszeg;

  //Az összeadást megvalósító utasítás
  osszeg = elso_szam + masodik_szam;

  //Függvényünk (alprogramunk) vége, a kiszámolt eredmény
  visszaadásával zárul
  return osszeg;
}
```

A példa két szám összeadását elvégző függvényt valósít meg, amely két bemenő paraméterrel rendelkezik. Ezek integer típusú számok, amelyeket a függvény törzsében összeadunk. Az eredményt egy változóban tároljuk el, majd a függvényünket a *return* utasítással zárjuk, amely értékül a kiszámolt összeget adja vissza, vagyis ez a függvényünk kimenete.

Olyan is lehet, hogy nincsen bemenő paraméter vagy nincsen visszatérési értékünk. Ennek jelölésére a **void** típus szolgál. A *void* kulcsszó fogja jelenteni azt lényegében, hogy üres. Arra, hogy a függvényeinket, hogyan tudjuk majd használni később látunk példát.

HOGYAN ÉPÜL FEL EGY C KÓD?

Most már eljutottunk oda, hogy ismerjük annyira a C nyelv elemeit, hogy megnézzük, hogyan is épül fel egy C kód. Programozási nyelv oktatása során az első program, amit látni szoktunk az a *"Hello world!"* kiírása, nézzük meg mi is ezt. Ezen a kis programon jól bemutatható egy általános program felépítése.

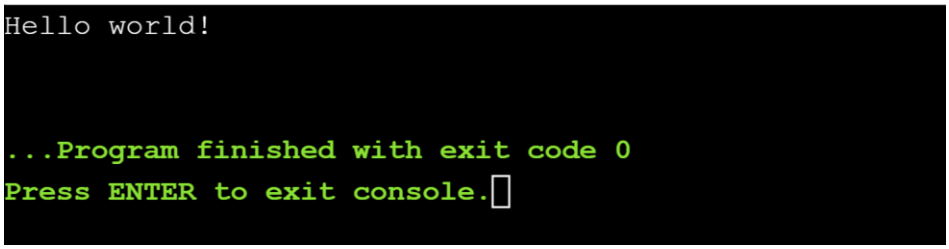
```
#include <stdio.h>
//Főprogramunk, a programunk belépési pontja
int main(void)
{
  /* Kiírja, hogy Hello world! szöveget a konzolra */
  printf("Hello world!\n");
  //Program végét a return zárja, a 0 azt hivatott jelenteni,
  hogy hiba nélkül futott le
  return 0;
}
```

Nézzük meg sorról sorra haladva, mi micsoda. Az első sorban található **#include <név>** kulcsszóval tudunk előre megírt programrészeket beilleszteni, importálni (mint például a korábban említett stdbool.h). Az stdio a **ST**andar**D I**nput **O**utput rövidítése, ami magyarra fordítva standard bemenet és kimenet-et jelent. Az stdio sok más mellett tartalmazza a **printf()** függvényt is, amivel mindenféle kiiratást el tudunk végeztetni. A **.h** magának a fájlnak a kiterjesztését mondja meg. A .h a header, vagyis fejlécállomány rövidítése. Az ilyen fájlok a függvények prototípusait és konstansokat tartalmaznak. Ezt követi a főprogram: az int main(void), ez minden C-s program belépési pontja. Látható, hogy ez lényegében egy függvény, amely egy integer-rel fog visszatérni és nincsen bemenő paramétere. Kapcsos zárójelek között helyezkedik el a függvény törzsünk.

Kódjainkat kommentekkel látjuk el, ahogy ezt a fenti példákban is tettük, `/* */` jelek közé kell írnunk a megjegyzéseinket, ha sorokon keresztül ívelő kommentet szeretnénk írni, azonban ha csak az adott sorban szeretnénk azt a `//` jelek kirakását követően tudjuk megtenni. Ez nekünk, akik a programot írjuk szól, a fordító ezeket a sorokat figyelmen kívül hagyja, mintha ott sem lennének.

Ezt követi a már említett `printf()` függvény, aminek a hasában áll a *"Hello world!*\n*"* szöveg, amely egy stringként van megadva. Emlékeztetőül a `stringek` a karakterek sorozata volt. A **\n** fogja jelenti az új sort. A `\n` mellett fontos karakter még a kiírásnál a **\r**, ezek általában együtt szoktak állni **\r\n** formában és a jelentésük: sor elejére és új sor, ezeket **vezérlő karakterek**nek nevezzük. Végül a programunk a `return 0;`-val zárul, ami befejezi a program futását és a 0 értékkel arra utal, hogy sikeresen végrehajtódott minden.

A lefordított és lefuttatott kód eredménye a terminálra kiírt *Hello world!*, ezt láthatjátok a következő ábrán.



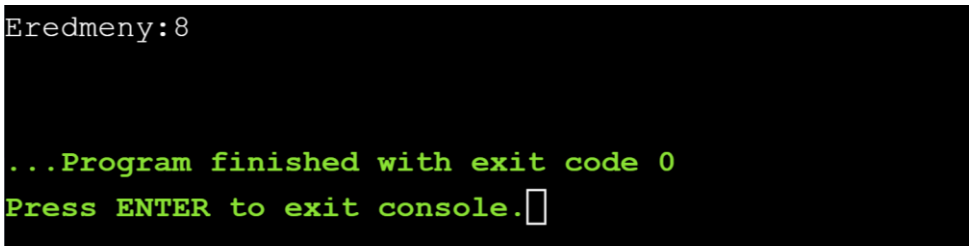
***1. ábra** - Hello world!*

Ki is tudod próbálni ezt a programot, ha szeretnéd. Ehhez nem is kell letöltened semmit, online is elérhetőek C fordítók. Ha a google keresővel a "c online compiler"-re rákeresel akkor találhatsz rengeteg ingyenes, regisztráció nélküli weboldalt, ahol a fenti kódot és a későbbieket is ki fogod tudni próbálni.

Nézzünk meg egy másik egyszerű programot, ami összead nekünk két egész számot. Ez az előző *"Hello world!"-höz* nagyon hasonlít.

```c
#include <stdio.h>
//Főprogramunk, a programunk belépési pontja
int main(void)
{
  /* Kiírja a művelet eredményét, a terminálképernyőre*/
  printf("Eredmeny:%d\n", 5+3);
  //Program végét a return zárja, a 0 azt hivatott jelenteni,
  hogy hiba nélkül futott le
  return 0;
}
```

Szembetűnő különbséget a *printf()-nél* látunk. A *printf()* függvénnyel nem csak egyszerű szöveget tudunk megjeleníteni, hanem akár egy matematikai kifejezés eredményét is. A **"%d"** helyére fog kerülni a kifejezésünk eredménye. A %d arra utal, hogy egy int típusú érték fog majd behelyettesítődni. Lebegőpontos szám esetén **%f**-et, karakter esetén **%c**-t, míg string esetén **%s**-et kell írnunk. A lefutatott programunk eredménye a következő.



***2. ábra** - printf() használatát bemutató számoló program kimenete*

Kitekintés

A főprogramunknak is lehetnek bemenő paraméterei. Általánosan a következő alakokban írható fel a main függvényünk: int main(int argc, char *argv[ ], char *env[ ])

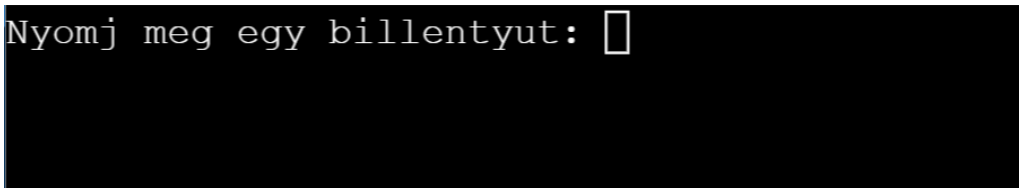
Az első paraméterként megadjuk az átadni kívánt argumentumok számát, majd az argumentumokra és esetlegesen a környezeti változókra mutató pointereket is. A pointer vagy magyarul mutató a memóriában tárolt érték címét adja meg. A paraméterek hátulról előre haladva el is hagyhatóak, ahogy láttuk is a példáinkban.

Az stdio könyvtárban sok másik hasznos függvény is található a *printf()* mellett. Most további két függvénnyel fogunk megismerkedni, amit a későbbi tananyagrészekben is használni fogtok. Ezek az *getchar()* és a *puts()* függvények. Nézzük meg ezeket egy-egy példán keresztül mire is használhatóak.

Az `int getchar(void)` függvény a standard inputról (például terminálról billentyűzet segítségével vagy fájlból) képes beolvasni egy karaktert. A függvény a beolvasott karakterrel fog visszatérni, bemenő paraméterrel pedig nem rendelkezik. A következő egyszerű program szemlélteti ennek használatát.

```
#include <stdio.h>
//Főprogramunk, a programunk belépési pontja
int main(void)
{
  char c;  //Ebbe a változóba fogunk majd beolvasni egy karaktert

  printf("Nyomj meg egy billentyut: ");  //Kiírja a megadott szöveget
                                         a terminálra
  c = getchar(); //Karakter beolvasását elvégző függvény
  printf("Beolvasott karakter: %c",c); //Kiírja a "c" változóban
                                       eltárolt karaktert a terminálra

  //Program végét a return zárja, a 0 azt hivatott jelenteni,
  hogy hiba nélkül futott le
  return 0;
}
```

Az indulásakor létrehozunk egy *c*-nek elnevezett változót, ebben fogjuk majd eltárolni a terminálról beolvasott karaktert. Szöveges üzenettel jelezzük magunknak, hogy mit kell tennünk, mire vár a gép. A programunk mikor elér a *getchar* függvényhez, addig várakozni fog, amíg meg nem nyomunk egy karaktert és le nem ütjük az *enter* billentyűt. Az *enter* lenyomását követően a programunk a következő utasítással folytatódik, ami szintén egy szöveges üzenet lesz. Ellenőrzésképp az imént beolvasott karaktert fogjuk kiírni, ezzel véget is ért a programunk. A következő két ábra a programunk futását mutatja be.



***3. ábra** - Várakozás a karakter lenyomására*

```
Nyomj meg egy billentyut: m
Beolvasott karakter: m

...Program finished with exit code 0
Press ENTER to exit console.
```

***4. ábra** - Beolvasott karakter sikeres kiírása*

Az `int puts(const char *str)` függvénnyel a standard kimenetre (képernyőre, fájlba vagy nyomtatónak) tudunk egy stringet küldeni. A függvény bemenő paraméternek egy stringet vár. Feltűnhetett, hogy ha csak kiírunk valamit, miért van visszatérési értékünk? A visszatérési értékben a függvény lefutásának állapotáról kapunk információt, hogy sikeres volt-e vagy sem. A következő egyszerű program szemlélteti ennek a függvénynek is a használatát.

```c
#include <stdio.h>
//Főprogramunk, a programunk belépési pontja
int main(void)
{
  char szoveg[] = "Egy string vagyok!";  //String létrehozása

  puts(szoveg);  // Ez a függvény az átadott stringet kiírja
                 a terminálra

  //Program végét a return zárja, a 0 azt hivatott jelenteni,
  hogy hiba nélkül futott le
  return 0;
}
```

Létrehozunk egy stringet *`szoveg`* néven, amelyben az *"Egy string vagyok!"* mondatot helyezzük el. Ezt a stringet adjuk át paraméterül a *`puts()`* függvénynek, amely kiírja nekünk az átadott szöveget a képernyőre.

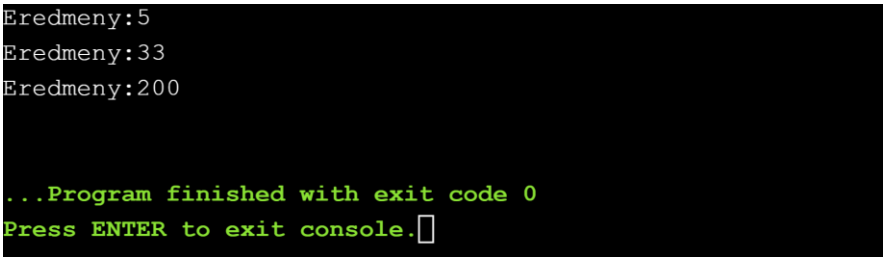
```
Egy string vagyok!


...Program finished with exit code 0
Press ENTER to exit console.
```

***5. ábra** - String kiírása puts() függvény használatával*

Emlékeztek még, azt ígértem, hogy látni fogjuk, hogyan kell használni az általunk megírt függvényt. Az utolsó példánk ezt fogja bemutatni. Az elkészített függvényeink a *`main`* függvényen kívül kapnak helyet. Egy függvény meghívására láthattunk már számos példát, gondoljunk itt a *`printf()`*-re akár. A következő kód tartalmazza az előzőleg már bemutatott *`osszead()`* függvényünket és annak használatát.

```c
#include <stdio.h>

//Korábban elkészített két szám összeadását megvalósító függvény
int osszead(int elso_szam, int masodik_szam)
{
  int osszeg;  // az eredményt itt fogjuk eltárolni

  osszeg = elso_szam + masodik_szam; //Az összeadást megvalósító
                                     utasítás

  return osszeg; // visszatérünk a kiszámolt eredménnyel
}

//Főprogramunk, a programunk belépési pontja
int main(void)
{
  int eredmeny;  //Ebbe a változóba fogjuk majd tárolni a függvény
                 kimenetét

  eredmeny = osszead(1, 4); //Meghívjuk az "osszead" függvényt 1 és 4
                            bemenettel
  printf("Eredmeny:%d\n", eredmeny); //Kiiratjuk az "eredmeny"-ben
                                     tárolt értéket

  eredmeny = osszead(12, 21); //Meghívjuk az "osszead" függvényt
                              12 és 21 bemenettel
  printf("Eredmeny:%d\n", eredmeny); //Kiiratjuk az "eredmeny"-ben
                                     tárolt értéket

  eredmeny = osszead(215, -15); //Meghívjuk az "osszead" függvényt
                                215 és -15 bemenettel
  printf("Eredmeny:%d\n", eredmeny); //Kiiratjuk az "eredmeny"-ben
                                     tárolt értéket

  return 0;
}
```

Ha megnézzük a főprogramot láthatjuk, hogy a függvényünk két egész számot kap meg, ami megfelel a paraméter listájának. Például az *osszead(1,4)* esetén az *elso_szam* helyére 1, míg a *masodik_szam* helyére 4 fog behelyettesítődni. Az eredményt minden esetben egy *eredmeny* néven létrehozott változóban fogjuk eltárolni. Látható, hogy a függvényt annyiszor és ott tudjuk meghívni ahol azt csak szeretnénk, csupán hivatkoznunk kell rá és helyes paramétereket átadnunk számára. A program kimenete a következő lesz:



***6. ábra** - Az összeadó függvény használatát bemutató program kimenete*

Ezzel befejeztük a programozás és a C nyelv alapjainak a megismerését is, illetve elkészítettük az első programjainkat. Természetesen a C nyelvnek vannak további, bonyolultabb elemei is, amely ezen tananyagon már túlmutat, de az alapok elsajátítását követően és kellő lelkesedéssel magad is folytathatod a C nyelv (később akár más nyelvek) megismerését.

SZOFTVERINTEGRÁCIÓ

Valós, ipari környezetben igen gyorsan elérjük azt az állapotot, amikor már egy-egy ember nem látja át a teljes rendszer működését, így a megírandó kód is olyan nagy és bonyolult lesz, hogy kénytelenek vagyunk azt valamilyen módon kisebb részekre bontani, ezt hívjuk modularizálásnak. Célunk, hogy a teljes rendszert nagyobb szolgáltatásokra, majd azokat további egyre kisebb és kisebb funkciókra – tehát modulokra – bontsuk. Így elérhetjük, hogy olyan szoftvermoduljaink legyenek, amelyek már könnyen átlátható kis egységei a rendszernek és így jóval könnyebben fejleszthetők és tesztelhetők. Ahhoz, hogy ezek a kis modulok megfelelően együtt tudjanak működni, a közöttük lévő kapcsolódási pontokat és interakciót is definiálni kell, ezek a kapcsolódási pontok az úgynevezett interfészek. Ha ezek megfelelően meg vannak határozva és karban vannak tartva, akkor nem csak azt nyerjük a modularizációval, hogy az egyes modulok könnyen módosíthatók és tesztelhetők, hanem egyszerűen cserélhetők is, illetve további új modulok – azaz új funkciók – is könnyen hozzáadhatók vagy feleslegesek kivehetők a rendszerünkből. Ez egy nagy ipari környezetben igen fontos szempont, de természetesen már nekünk is hasznunkra fog válni a kódolás folyamán.

Egy komolyabb rendszer esetében a modulok fejlesztéséért különböző szoftverfejlesztők felelnek. A szoftver integrátor (más néven koordinátor) feladata az interfészek karbantartása és a modulok egységes rendszerré alakítása és konfigurálása, míg a tesztelést megint mások végzik.

A legtöbb iparágban a szoftver integrálását és konfigurálását különböző szabványok is támogatják. A forráskód karbantartását és követhetőségét pedig különböző, úgynevezett verziókövető rendszerek segítik. A verziókövetésnek a lényege, hogy a forráskód egy adott állapotát bármikor vissza tudjuk állítani, illetve reprodukálni. Ennek többféle megvalósítása is lehetséges. A legegyszerűbb talán az, ha kódunkat minden nap új, dátummal elnevezett mappába rakjuk. Ez egészen hatékony megoldás lehet kis rendszereknél, vagy ha csak egy fejlesztő dolgozik rajta. Ám amint bonyolódik a kódunk, vagy mások is bekapcsolódnak a fejlesztésbe, gyakorlatilag karbantarthatatlanná válik a helyzet. Ezért sokkal célszerűbb valamilyen fejlettebb verziókövető rendszert használni. Ezeknek lényege, hogy a forráskódot egy mindenki számára elérhető központi szerveren tárolják, így mindig a legaktuálisabb kód érhető el. Ha módosítani kell, akkor a fejlesztés bármely szakaszában visszatölthető a megváltoztatott kód, így valóban mindig a legutolsó állapot érhető el mindenki számára. További nagy előnye ezeknek a rendszereknek, hogy lehetővé teszik különböző fejlesztési ágak létrehozását. Ez jól jön akkor, ha egy funkciót, vagy modult

többféleképpen lehet megvalósítani és azt akarjuk megtudni, hogy számunkra ezek közül melyik a legmegfelelőbb megoldás. Például, ha ki akarjuk próbálni, hogy megszakítással, vagy állandó lekérdezéssel jobb-e kezelni valamilyen perifériát (lásd később a „Megszakítások a szoftverben – interruptok” című tananyagrészben). Ilyenkor létre tudunk hozni mindkét lehetőségnek egy-egy ágat és így az eredeti kódot érintetlenül hagyva, egymástól teljesen függetlenül megírhatjuk és kipróbálhatjuk mindkét megoldást. Érezhető, hogy ebben a helyzetben a dátumozott mappás megoldás meglehetősen bonyolítaná az átláthatóságot. Ezzel szemben rengeteg jól használható verziókezelő rendszer segíti ennek a követhetőségét. Vannak teljesen ingyenesek is (például SVN, Git), érdemes kipróbálni, használni őket, jelentősen könnyítik a munkát.