**A C nyelv jellemzői:**

* compileres
* erősen típusos
* magas szintű nyelv, gépközeli lehetőségekkel
* általános célú
* kevés utasítás, gazdag függvénykönyvtár

**C modulok:** Egy C program több modulból áll. Egy modul egy önállóan fordítható file. A függvénykönyvtárak valójában lefordított modulok. (Tehát ha mi csak egy modulos programot írunk, akkor is több modulból áll a program, hiszen használjuk a függvény könyvtárakat, amelyek külön modulokban vannak.)

**Header (.h) file-ok:** Egy másik modulban definiált változó vagy függvény (pl. valamelyik függvénykönyvtárban) használatához a modulunkban deklarálnunk kell a változót vagy függvényt. Ennek megkönnyítése a .h file-ok. Csoportosítva tartalmazzák a könyvtári függvények deklarációit. Például a kiíró függvények az stdio.h file-ban vannak deklarálva, a matematikaiak a math.h fileban, stb. A szükséges .h file-okat a fordítóval beszúratjuk a forrás file-unk elejére.

**Konzolra írás:** Mivel nincs kiíró utasítás, kiírásra a printf könyvtári függvényt használjuk. A printf deklarációja a stdio.hban van. A #include a fordítónak szóló utasítás (nem C utasítás), hogy szúrja be a stdio.h file-t a forrás programba.

**Egy projekt buildelése:** Fordítás és linkelés. Az összes szükséges file fordítása és összeszerkesztése.

**Típuskonverziók:**

* automatikus konverzió: Magától elvgéződik
* kikényszerített konverzió: Csak típuskonverziós operátor segítségével.

Műveletvégzés során a a két operandus közül a szűkebb konvertálódik automatikusan a bővebbre(float és double osztása esetén a float konvertálódik double-re).

Értékadásnál az érték mindig konvertálódik a változó típusára: int a = 4.14 az 4 értéket vesz fel, míg char c = 68.12, ekkor az „A” karaktert kapja értékül (68 az az A karakter a karaktertáblában).

Nézzünk egy konkrét példát:

int b = 4.2 + 7/a; (az a változó a short, és az értéke 2)

1. az osztás: tehát 7 az int, „a” short, „a” konv. int-re

2. összeadás: 4.2 double és 3 int, 3 konvertálódik double-re

3. értékadás: a 7.2 double konvertálódik int-re

**Kikényszerített konverzió:** (típus) kifejezés

Akkor is végbe megy, ha csonkolnia kell. Hasznos lehet osztásnál, ha például a/b esetén mindkét operandus egész, akkor az osztás eredménye biztosan egy egész szám, viszont ha a kettő közül csak az egyiket is double-é kasztoljuk, akkor a másik autómatikusan konvertálódik a a bővebb double típusra, így az osztás eredménye egy double érték lesz.

Java nyelven a kikényszerítés azért nagyon hasznos, mert ott például nincs meg az értékadásnak az a tulajdonsága, hogy értékadásnál konvertálódik a változó típusára az érték.

Tehát Javaban az int a = 4.2; az hibát dob, míg C-ben nem,

ott csak int a = (int)4.2; -vel oldhatjuk meg.

int a = (int)4.2; ez mindkét nyelven jó!!!!!

**Kapcsos zárójel utasítás:**

{

utasítás;

}

Kapcsos blokk olyan helyre is írható, ahová egyébként csak egy utasítást írhatunk. Lényege, hogy hatáskört definiál változóknak, tehát amely változók a blokkon belül lettek definiálva, azok élettartama csak a blokk végéig érvényes. A blokkokat ok nélkül nem szoktuk használni. A blokkok értelem szerűen egymásba is ágyazhatóak.

**Változó deklaráció:** Blokkon belül deklarált változók, csak a blokk belsejében élnek. Blokkon belül bárhol deklarálhatunk változókat, viszont az a helyes gyakorlat, ha minél közelebb deklaráljuk a felhasználásához, így minél kisebb lesz a hatásköre és az élettartama.

A változókhoz tartozhatnak módosítók is:

* auto:
* register:
* static:
* extern:
* const: Csak egyszer kaphat értéket. const double MAXVALUE = 10;

Szokás a konstans változóneveket csupa nagybetűvel írni. Van még egy lehetőség konstans létrehozására, ami viszont nem C nyelvi utasítás, hanem a fordítónak szóló utasítás:

# define AA 12

Hatására a fordító, még a fordítás elején, a forrásprogramban levő összes AA azonosítót kicseréli 12-re. Vigyázat nem ellenőriz semmit, hanem egyszerű szöveg cserét hajt végre!! Nincs típusa sem.

**Printf könyvtári függvény használata:** Ezzel kiírhatunk a standard outputra (konzol).

printf (”szöveg”); /\*Ez kiírja a szöveget az outputra\*/

Viszont létezik a % operátor, melynek segítségével különböző típusú változók értékeit is kiírhatjuk:

printf(”Ma %d fagyit ettem.\n”, 3);

% jel után szerepelhetnek:

* d(int)
* ld(long)
* lf(double)
* c(char)

**Scanf könyvtári függvény használata:** Standard inputról való beolvasást tesz lehetővé (billentyűzet).

Ez is az stdio.h-ban van definiálva.

int alma;

scanf(”%d”, &alma);

A fenti két sor eredménye, hogy alma nevű változóba beolvasok egy számot a standard inputról.

Az első paraméter, hogy milyen típusú adatot olvasok be.

A második paraméter pedig, hogy milyen memóriacímre történjen a beolvasás.

**Matematikai függvények:** A math.h –ban vannak deklarálva.

**Logikai kifejezések számokkal:** A C nyelvben a számok is értelmezhetőek logikai kifejezésként. (a 0 az a hamis, míg minden más igaz értéket jelent).

**Tömb adattípus:**

A tömbnek két fajtája van:

* statikus: A tömb létrehozásakor konstansként meg kell adni az elemeinek a számát és a fordító lefoglalja neki a megfelelő méretű helyet a memóriában.
* dinamikus: A tömb definíciójakor változóval adom meg a tömb méretét. Ilyenkor a fordító nem tud területet foglalni a tömbnek, hiszen nem ismeri a méretét. Csak futási időben foglalódik a hely.

int a[10]; (Javaban -> int[] a = new int[10;])

Ez a fajta tömb megadás csak inicializáláskor lehetséges, tehát

int a[];

a = {10,20,20}; - > Ez hibás(utólag már csak egyesével adhatunk értékeit a tömb elemeinek)

int a[] = { 5, 3, 6 };

**Memória:** A tömbök mindig garantáltan egybefüggő területet foglalnak el a memóriában, ahol az elemek egymás után helyezkednek el. Tehát pl. ha egy double tömb első elemének a címe x, akkor a második elemének a címe double méretnyivel több (azaz 8-al), a harmadik pedig még egy double méretnyivel, stb. Ezt a tulajdonságát a tömböknek a C ki is használja.

**Függvény paraméterek:** A paraméterek a függvények csak a függvény blokkban láthatóak (hatókörük a függvény blokk) és a függvénybe lépéskor jönnek létre és a függvényt elhagyva el is pusztulnak (élettartamuk dinamikus).

Tehát nem tudjuk azt megcsinálni, hogy átadunk egy változót a függvénynek, az meg megváltoztatja az értékét, legalábbis csak addig változtatja meg, amíg a blokkon belül van.

void modify(int szam) {

szam = szam + 10;

printf("%d", szam);

}

int main()

{

int szam = 10;

printf("%d", szam);

modify(szam);

printf("%d", szam);

}

A fenti kódnak az eredménye : 10 20 10

Tehát látható, hogy csak a függvény blokk belsejében lehet a változó értékét megváltoztatni érték szerinti paraméter átadásnál.

A függvények paramétereinek két fajtája van:

* formális paraméter: A függvény definíciójában meg kell adni a függvény milyen paramétereket használ, sőt szigorúan típusos nyelveknél ezek típusát is. A függvény definícióban megadott paramétereket formális paramétereknek nevezzük.
* aktuális paraméter: A függvény meghívásakor meg kell adni, hogy a formális paraméterek milyen értékeket kapjanak. A meghíváskor felsorolt értékek az aktuális paraméterek.

A függvény híváskor tehát az első lépés, hogy az aktuális paraméterek értékei valahogy átadódnak a formális paramétereknek.

Híváskor a hívó bedobálja egy verembe az aktuális paraméterek értékeit. Mikor pedig átadódott a vezérlés a függvényre ő felolvas annyi adatot a veremről, ahány formális paramétere van. A paraméterátadás egyirányú, azaz az aktuális paraméter átadja értékét a formális paraméternek. Vissza irányba nem működik. Tehát hiába változtatom meg a formális paraméter értékét annak semmilyen kihatása nincs az aktuális paraméterre. Ez az érték szerinti paraméterátadás esetén.

**Mutatók:**

Olyan változók, amelyek:

- értékül egy memória címet tartalmaznak (tehát tulajdonképpen egy egész számot tartalmaznak)

* lehet műveleteket végezni az értékével, azaz a memóriacímmel
* valamint definiálva vannak műveletek, amelyek képesek kezelni a mutató által tartalmazott címen levő tartalmat.

**Mutatók típusa:** A mutatóknak bár értékül egyformán egy címet tartalmaznak, mégis van típusa. A mutató típusa azt adja meg, hogy a hivatkozott címen levő adatot milyen típusként kell kezelni. Tehát egy double típusú mutató nem azt jelenti, hogy a mutató értéke (a cím) double, hanem hogy a hivatkozott címen levő adatot kell double-ként kezelni.

definiálása: double \*a;

**Mutatónak való értékadás:** A mutatóknak tehát egy értelmes memória címet kell értékül adni, és ezt három módon tehetjük meg:

* Egy már létrehozott változó címét adom értékül:

Használhatjuk a már ismert címképző operátort.

int a;

int \*mut;

mut = &a;

int \*mut2;

mut2 = mut;

A mut értéke az „a” változó címe lesz. A mut2 értéke mut értékét veszi fel, azaz szintén az a változó címe lesz. (A mutatók értékei ki is írhatók %ld formázóval.)

* Dinamikus helyfoglalással lefoglalt memória terület címét adom értékül:
* Mutató között is létezhet értékadás, tehát egy másik mutató értékét kapja értékül:

mut2 = mut;

* Illetve van még egy módszer, amikor null-t adok neki értékül, bár ilyenkor a semmibe mutat:

A \* operátort már ismerjük szorzás operátorként, valamint a mutatók deklarálásához is szükség van rá. A harmadik szerepe a hivatkozás a mutató által mutatott területre. „\*” operátor szerepe, hogy a mutató hivatkozott területét adja. A \*mut kifejezés értéke tehát a mutató által mutatott területen levő érték.

A \*mut egyben balérték is tehát értéket is felvehet és ilyenkor a mutatott területre íródik be az érték.

double a = 2;

double \*mut = &a;

printf("%lf\n", a);

\*mut = 5.3;

printf("%lf\n", \*mut);

printf("%lf\n", a);

Az első kiírás eredménye 2, ugyanis ott nem változtattunk az értékén. Majd a mut mutatónak értékül adtuk az „a” változó címét, majd a \*mut=5.3 –al a mutatott területen lévő értéket (ami 2), azt most átírjuk 5.3-ra. Mivel „mut” és „a” is ugyan arra a memóriacímre mutat, ezért értékük ugyancsak 5.3 mindegyiknek.

Hivatkozás a mutatott területre Figyelem! Az egyik legnagyobb hiba, ha olyan mutatónál hivatkozunk a mutatott területre, amelynek nem adtunk értelmes értéket.

Mint pl. double \*mut; \*mut = 5.3;

A mut nem kapott értéket tehát a memóriában ott maradt szemét az értéke, valószínűleg 0. És a második utasítással ennek a memória területnek akarom megváltoztatni az értékét.

**Indirekciós és címképző operátorok:**

* Indirekciós operátor(\*):
* Címképző operátor(&):

Az indirekciós operátor és a címképző operátor egymás ellentettje: &(\*mut) ua. mint mut ua. mint \*(&mut)

**Egyéb műveletek a mutatókkal:**

Egész szám hozzáadása, kivonása Ha mut értéke x, akkor mut+1 értéke nem x+1, hanem x+(mérete a mutató típusának).

Tehát például double \*mut; esetén a mut+1 kifejezés értéke 8 byte-nyival nagyobb értékű, mint mut, hiszen a double 8 byte-os.

double a;

double \*mut = &a;

printf(”mut=%ld, mut+1=%ld\n”, mut, mut+1);

Ilyenkor kiírja, hogy 1232420 és 1232428

Integer esetén ez csak 4(4byte)-vel emelkedne.

**Index operátor mutatókra:** Itt az a lényeg, hogy a mutatókra is használható a tömböknél is megszokott index operátor.

írtunk volna, azaz mut[0] ua, mint \*mut

mut[1] ua, mint \*(mut+1)

Az index operátorral tehát a mut címhez képesti egymást követő adatokat érhetjük el, feltéve hogy mindegyik azonos típusú. Ennek tehát akkor van értelme ha a mutató egy tömb kezdőcímét kapja meg, hiszen akkor tudjuk, hogy a tömb elemei a memóriában folytonosan helyezkednek el és azonos típusúak. Tehát látható, hogy elég a tömb kezdő címe, és ezen módszerrel végigiterálhatunk az elemeken, ugyanis a memóriában sorban helyeszkednek el a tömb elemei.

int szamok[10];

int \*mut = &szamok[0]; esetén tehát

mut[0] ua, mint szamok[0]

mut[1] ua, mint szamok[1]

Ami észrevehető a fenti példából, hogy létrehoztunk egy 10 elemű tömböt(lefoglalva neki a memóriát a deklaráció során). Majd a \*mut-nak értékül adtuk a tömb első elemének címét. Innentől kezdve, hogy megvan az első elem címe, könynedén végigmehetünk a tömb elemein, a pointeren keresztül.

**Tömb és mutató közötti hasonlóság:** Valójában azért ugyanolyan a tömbindexelés és a mutató indexelés, mert a tömb neve egy mutató.

Például: int szamok[10]; esetén szamok egy konstans mutató a tömb kezdetére és a

szamok[2] hivatkozás a valóságban \*(szamok+2)-ként szerepel.

Bármikor használhatnám így is. A két hivatkozás fajta ekvivalens. (így már tudjuk mi történik, ha túl hivatkozok a tömbindex határokon.)

**Mutató szerepe paraméterátadáskor:** Mutató segítségével el lehet érni, hogy egy függvény képes legyen módosítani a hívó egy változóját!! Tehát nem csak a függvény blokkjában változtatja meg az értékét, hanem ténylegesen globálisan. Ha egy függvény paramétere mutató, akkor híváskor a hívó egy változójának a címét fogja megkapni. Mivel az indirekciós(\*) operátorral tudunk hivatkozni a mutatott területre, akár értéket is adhatunk neki, így a függvényben közvetve a hívó egy változójának adunk értéket.

void duplaz(int \*a)

{

\*a = \*a \* 2;

}

int main()

{

int x = 4;

duplaz(&x);

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

}

Látható, hogy itt a hívó x változó értékét módosítjuk. A meghívott metódusnak átadjuk paraméterként a változónk címét(cím szerinti paraméter átadás).

A másik oka annak, hogy mutatókat használunk az a gyorsítás: Ha van egy nagyobb méretű adat, pl. egy tömb, annak átadása (vagy visszaadása) viszonylag sok memória területet és időt igényelne. Ha csak a címét adjuk át az csak néhány byte a tömb méretétől teljesen függetlenül.

**Tömb függvény paraméterként használata:** Egy függvénynek tehát, amely tömböt kell kapjon, kell hogy legyen egy mutató paramétere, amely megkapja a tömb kezdő címét. Ezenkívül kell hogy legyen egy másik paramétere, amelyben a tömb méretét kell átvegye!!!

int tombosszeg(int tomb[], int n)

{

int ossz = 0;

int j;

for (j = 0; j < n; j++)

ossz += tomb[j];

return ossz;

}

int tombosszeg(int \*tomb, int n)

{

int ossz = 0;

int j;

for (j = 0; j < n; j++)

ossz += tomb[j];

return ossz;

}

A fenti két metódus teljesen ugyan azt csinálja, ugyanis a tömb neve az egy konstans mutató a tömb kezdetére. szamok[2] hivatkozás a valóságban \*(szamok+2)-ként szerepel.

A fenti metódusok használata:

int t[4] = {10,20,30,10};

int szam = tombosszeg(t, 4);

printf("%d", szam);

a tömböt azért nem &t –vel kell átadni, mert maga a tömb neve is egy kezdeti cím a tömb kezdetére. Soha ne feledd-> a tömb neve a tömb elemeinek kezdőcímét tartalmazza!!!!!!!

Mivel a tömbnek csak a címét adjuk át, így ha a függvény módosítja a tömb elemeit, akkor az a hívó tömbjére fog vonatkozni.

**Mutató visszatérési érték:** tipus \*fgvnev (parameterek)

Ennek szerepe, hogy összetett adatot tudjunk visszaadni. Tehát nem a tömböt hanem a címét, nem a struktúrát hanem a címét adjuk vissza.

Az alábbi példa egy súlyos hibát fog bemutatni:

int \*fgv()

{

int x = 2;

return &x;

}

int main()

{

int \*a;

a = fgv();

}

Mi itt a probléma? Az x a függvény változója elpusztul a függvényt elhagyva. Tehát egy olyan változó címét adtam vissza, amelyhez tartozó terület felszabadult, tehát rövidesen másra fogja használni a program. Ügyeljünk tehát arra, hogy olyan terület címét adjuk vissza, ami nem pusztul el a függvényt elhagyva. Tipikusan ilyen a dinamikusan foglalt terület vagy egy paraméterben megkapott cím.

Egy olyan függvény, amely beolvas egy tömbbe nem a függvényen belül hozza létre a tömböt, hanem meg kapja paraméterben. Hibás!!!!!:

int \*beolvas()

{

int tomb[10];

int j;

for (j = 0; j < 10; j++) {

printf(”Kerem a %d.szamot: ”, j);

scanf(”%d”, &tomb[j]);

}

return tomb;

}

A fenti kód azért hibás, mert nem tudunk visszatérni a címmel, mert az a blokk végén megszűnik, tehát csak olyan tömbbe olvashatunk be, amit paraméterként kapunk.

**Mutatóra mutató:** Lehetséges olyan mutatót is készíteni, amely egy mutató címét tartalmazza, így a hivatkozott adat csak kétszeres indirekcióval érhető el. Pl.

int \*\*mut;

és a hivatkozás is

\*\*mut = 4;

Ennek haszna a többdimenziós tömbök kezelése, illetve tömbök tömbjének kezelésében van.

**A C nyelv és a String adattípus:** Más nyelvekbe definiáltak String adattípust, de ezt a C nyelvből kihagyták.

Karakter tömbökkel valósíthatjuk meg a string adattípust. Probléma, hogy a tömbnek van fix mérete, ha nem töltjük fel végig, akkor kell hozzá még egy olyan adat, amely a szöveg hosszát tartalmazza. A string műveletet végző függvények egy 0 kódú karakterig olvas, azaz az utolsó használt karakter mögé be kell tenni még egy 0 kódú karaktert. Tehát arra figyeljünk, hogy a tömb mérete eggyel nagyobb legyen mint a tárolni kívánt szöveg, mert gondolni kell a nulla kódú karakterre.

char szov[50] = {'a','l','m','a','\0'};

char szoveg[50] = "alma";

A fenti két sor eredménye ugyan az, a második hatására automatikusan beszúródik a 0 kódú karakter a végére.

Ha nem adok meg méretet, akkor ugyan úgy akkora mérettel fogja létrehozni mint sima int tömbnél.

char szoveg[] = "alma";

printf("A kedvenc gyümölcsöm az %s", szoveg);

**Mátrixok a C nyelvben:** Egy egyindexes tömb esetén (pl int a[5];) a tömb neve egy konstans mutató a tömb kezdetére és az index operátor \*(tombnev+index) alakra átírható, de mi van a tömbök tömbjés két dimenziós mátrixxal?

\*(tombnve) -> ez a tömb kezdőcímét edó érték, tehát a nulladik elem.

\*(tombnev + 1) -> ez az első tömbelemet adja

int matr[][3] = {

{1,2,3},

{4,5,6}

};

printf("%d",\*(\*(matr + 1) + 1));

A fenti példának az eredménye az „5”.Illetve az is észrevehető, hogy kétdimenziós mátrix esetén a sor indexet nem kötelező megadni deklaráláskor.

\*(\*(a+sor) + oszlop)

de ez ekvivalens lenne ezzel az utasítással,

printf("%d",\*(matr[1]) + 1);

**Egyindexes tömb mutatónak való értékadása:** Tehát egy egyindexes tömb értékül adható egy mutatónak.

int a[5];

int \*mut = a;

mut[2] = 3;

Tehát itt van az „a” tömbünk. A mut mutatóval ezen tömbre mutatunk, tehát ennek a tömbnek az első elemére. Majd mutatón keresztül tudunk hivatkozni a tömb területére.

**Mutatók tömbje:** Az int\* mut[3] ez egy olyan tömb, amely három mutatót tartalmaz, tehát mutatók tömbje.

int a, b;

int \*tomb[2];

tomb[0] = &a;

tomb[1] = &b;

\*tomb[1] = 5; ua.mint b = 5;

printf("%d", \*tomb[1]);

printf("%d", b);

A kettő ugyan arra a memória területre mutat.

int t1[3] = {1,2,3};

int t2[4] = {10,11,12,13};

int \*mut[2] = {t1, t2};

esetén

mut[1][3] értéke 13

Tömbök tömbje mutatós megoldással használhatjuk.

**Dinamikus memóriakezelés:** Minden eddig használt változónak a területfoglalás már fordításkor eldőlt és a változó élettartamától függően foglalta a helyet a blokk végéig vagy a program végéig. A dinamikus memória kezeléssel lehetőségünk van futás közben foglalni területet és felszabadítani.

Ennek előnyei:

* olyan adatoknál (pl. tömböknél) amelyek méretét csak futásidőben tudjuk meg( ilyenkor amikor megtudjuk a tömb méretét, dinamikusan olyan méretűt foglalunk le)
* futás közben a területet fel is szabadíthatjuk

Ezeket a lefoglalt területeket a programozónak fel kell szabadítani programkódból, mert ha nem teszi memória szivárgás történik, minden újra és újra meghívásakor új területet foglal le.

**Dinamikus memória foglaló függvények:**  A C nyelvben vannak definiált metódusok memória foglalásra. Ezek visszatérési értéke void\*, ami azért van, mert előre nem tudhatták a függvény írói, hogy milyen típusú (hány byte) adatnak foglal helyet.

Ez a dinamikus memória foglalás azért előnyös, mert bizonyos esetekben nem tudhatjuk előre, pl. egy tömb, egy karakter, vagy egy láncolt lista elemszámát, ezért a statikus memóriakezelés teljesen csődöt mond. Arról nem is beszélve, hogy pl. egy statikusan kezelt globális hosszú tömb a program egész futási ideje alatt ott foglalja a memóriát, míg egy dinamikusan lefoglalt területet bármikor felszabadíthatunk, ha már nem kell.

int n = 3;

int tomb[n];

a fenti kódra már a fordító is hibát dob, ugyanis tömb paraméternek egy konstanst vár.

A felhasználáshoz tehát a függvény által visszaadott mutatót konvertálni kell a megfelelő mutató típusra. Igen, a mutatót is lehet konvertálni (lényegében az is változó).

**A malloc függvény:** Ezzel a függvénnyel

void \*malloc(size)

Lefoglal egy byteokban megadott méretet. A sizeof() metódus segítségével lekérdezhetjük, hogy az adott adattípus hány byteon tárol, majd azt beszorozva egy n értékkel(a példában 10), egy n elemű tömbnek foglalunk le helyet. Látható, hogy a visszatérési értéket int\* -á konvertáljuk(a void\* -ot). A metódus NULL értékkel tér vissza, ha nem sikerült a helyfoglalás(például nincs elég memória).

int \* t = (int \*)malloc(10 \* sizeof(int));

t[0] = 10;

t[1] = 21;

printf("%d", t[0]);

printf("%d", \*(t + 1));

A fenti példában a lefoglalt területet használjuk.

Használatához inkludálnunk kell a #include "malloc.h"

**A calloc függvény:**

void \*calloc(n, size)

Ennél két paraméter van, tehát lefoglal n darabnyi, size méretű egybefüggő területet, és ezt a területet ki is nullázza(tehát minden tömb elem értéke nulla). Ez gyorsabb kinullázás mintha malloccal lefoglalnék egy területet és ciklusban nulláznám ki.

int \*t = (int \*)calloc(10, sizeof(int));

A calloc függvény használatához is #include "malloc.h"

szükséges.

**Lefoglalt terület felszabadítása:** Felszabadítja a mutató által mutatott területet. Ha ez nem egy malloc vagy calloc által foglalt terület akkor nem működik(tehát sima mutatókra amik nem malloccal vagy calloccal lettek létrehozva nem működik).

void free(void \*pointer)

int \*t = (int \*)malloc(n \* sizeof(int));

//t használata

free(t);

felszabadítás szükségessége: Ha van egy olyan metódus amiben malloccal területet foglalok, és a metódus végén nem szabadítom fel, akkor a program futásában akár hányszor meghívódik ez a metódus az mindig új területet fog lefoglalni, és elkezd dinamikusan nőni a memória használat(ez persze a program leállása után felszabadul, de a program futása közben okozhat problémát). Tehát ha egy eseménykezelő metódus memóriát foglal, de nem szabadít fel, akkor sokszor keletkezik ez az esemény, akkor ez betelítheti a memóriát.

Van egy fgv. amelyben dinamikusan lefoglalok egy területet egy lokális változóval hivatkozok rá és fel is kellene a végén szabadítanom, de elfelejtem. Ilyenkor a függvényt elhagyva a változó, amivel hivatkoztam a területre megszűnik, tehát a terület megmarad, de nem elérhető többet és ráadásul a függvény minden meghívásával egyre több memóriát használ a program.

Sokkal nehezebb ügyelni a felszabadításra egy függvény belsejében foglalt terület esetén, hiszen ez minden híváskor foglalódik. Ilyenkor ha lehetséges szabadítsuk fel még a függvény elhagyása előtt. Ha ez nem lehetséges, akkor jól dokumentáljuk a függvényt.

**Lefoglalt terület átméretezése:** A malloc-al vagy calloc-al lefoglalt terület méretét utólag is megváltoztathatjuk.

void \*realloc(void \*ptr, size\_t size);

Az alábbi példában

int \*tomb = (int \*)malloc(10\*sizeof(int));

realloc(tomb, 5 \* sizeof(int));

A size az új méret. Ha nagyobb, mint a korábbi, akkor az új területek értékei definiálatlanok. Ha kisebb, mint a korábbi, akkor a felesleges területek felszabadulnak, a megmaradt területen levő értékek megmaradnak. Ha NULL az eredmény, ha nem sikerül a foglalás, vagy ha méretnek 0-t adunk.

**Dinamikus memória kezelés más nyelvekben:** Dinamikus helyfoglalás minden nyelvben létezik, de pl. a Java és c# nyelvben a foglaláskor a típust kell megadni amilyen típusú adatnak helyet akarok foglalni: pl. new int[10]; És mivel nincsenek mutatók a lefoglalt területre egy referenciát kapok, amivel sokkal könnyebb dolgunk van, mint egy mutatóval. Valamint a felszabadítás gondját leveszi vállunkról a futtató környezet.

**Élettartam, hatáskör:**

* élettartam: amíg a változó él, azaz foglalja a memória területet.
* hatáskör: a program kód azon része, ahol az azonosító ismert, használható. (Nem csak változóra érvényes)

Változók hatásköre: Blokkon belül definiált változók hatásköre a blokk végéig tart. Élettartamuk alapján a blokkon belül definiált változóknál két csoport létezik:

* automatikus változó: Az auto kulcsszóval, vagy annak elhagyásával használható. (tehát ha nem írok a változó mellé semmit csak a típust és a nevet, akkor az automatikusan automatikus változó). Amelyek definiálásukkor létrejönnek és amelyek a blokk végén automatikusan meg is szűnnek. Ha blokkba újra belép a végrehajtás újra létrejön.
* statikus változó : A blokkot elhagyva nem szűnik meg csak nem lesz elérhető. Ha a blokkba újra belép a végrehajtás, akkor újra látható lesz és értéke annyi lesz, mint ami volt. Definiálás a static kulcsszóval. (Példa: fgv. ami kiírja hanyaggyára hívják meg.)

void foo()

{

int a = 10;

static int sa = 10;

a += 5;

sa += 5;

printf("a = %d, sa = %d\n", a, sa);

}

A fenti metódust 10-szer meghívva a következő kimenetet kapjuk:

15 15

15 20

15 25

15 30

15 35

15 40

15 45

15 50

15 55

15 60

Látható, hogy a static változó értéke a blokkba visszakerülve mindig az előző értéket hordozza.

Függvényen kívül deklarált változók: Élettartamuk alapján statikusak, azaz a program végéig élnek. Hatáskörük alapján:

* Modulra lokális: Csak az adott modulban látható, tehát több modulos programnál a többi modulban nem látható használható. Az adott modulban a deklarálás pontjától a modul végéig minden függvény látja. A definiálásakor ki kell tenni a static kulcsszót.

static int valami;

* Globális: Bármelyik modulban látható. Kulcsszó nem kell definiáláskor. Ha viszont egy másik modulban látni akarjuk, akkor abban a modulban deklarálni kell az extern kulcsszóval. Tehát egy modulban kulcsszó nélkül definiált változó látható az adott modulban és az olyan modulokban ahol extern kulcsszóval deklaráltuk.

**Függvények hatásköre:**

a) modulra lokális Csak az adott modulban látható. A static kulcsszót kell használni.

b) globális Minden modulban látható. Kulcsszó nélkül kell definiálni. Abban a modulban viszont, ahol látni akarjuk extern kulcsszóval deklarálni kell.

**Előfordító:**  A fordítás első lépése, hogy a programba írt fordítónak szóló utasításokat végrehajtja. Ezek tulajdonképpen még a forráskódon manipulálnak. Szokás a fordító azon részét, amely ezt a lépést végzi előfordítónak nevezni.

* file beszúrás:

#include<file>

#include ”file” Beszúrja a megadott file-t a forráskódba.

* konstans definíció:

#define NEV érték

Hatására az fordító a forráskódban mindenütt kicseréli a NEV azonosítót érték-re.

**Enum:**

enum meretek {

KICSI = 0, KOZEPES = 5,

NAGY

};

int main()

{

meretek meret = KICSI;

if (meret == KICSI) {

printf("alma");

}

printf("%d", meret);

return 0;

}

A lényeg, hogy az enum elemekhez, ha nem írunk számot, akkor az első elem a 0, majd ezt követően mindegyik eggyel nagyobb, tehát a fenit példában a NAGY konstans mögött a 6-os szám áll, mert az őt megelőző érték az 5.

A típus definíciókra is létezik hatáskör, bár itt nem jelent problémát ha függvényeken kívül definiálunk, ellenben a változóval.

**A typedef kulcsszó:** Lehetséges egy típust alias névvel ellátni a typedef segítségével. Ennek jelentősége a hosszabb típus definícióknál van pl. enum, vagy struct típusoknál, de máshol is használható.

typedef int egesz;

egesz szam = 10;

printf("%d", szam);

Látható, hogy az integer típusra bármikor használhatom az egesz kulcsszót is típusparaméternek.

typedef enum meretek {

KICSI = 0, KOZEPES = 5,

NAGY

} Meretek;

int main()

{

enum meretek meret = KICSI;

Meretek nagy = NAGY;

if (meret == KICSI) {

printf("%d", meret);

}

printf("%d", nagy);

int bekeres;

scanf\_s("%d", &bekeres);

return 0;

}

Látható, hogy most már simán simán a typedeffel megadott Meretek szóval megadható a típus (bár a meretek is jó, nem fontos elé elvileg az enum kulcsszó).

Szokás a típusnevet elhagyni, és csak aliast definiálni:

typedef enum{

KICSI = 0, KOZEPES = 5,

NAGY

} Meretek;

**Struktúra:** egy összetett adattípus, amelynek elemei lehetnek különböző típusúak, minden elemnek saját neve van, az elemeire tehát névvel hivatkozunk. Jelentősége összefogni egy adatszerkezetbe a logikailag egybe tartozó adatokat. Pl. egy személy neve, életkora, fizetése, stb.

typedef struct {

char nev[30];

unsigned int kor;

unsigned int fizetes;

} Szemely;

int main()

{

Szemely a;

strcpy\_s(a.nev, "Kiss Viktor");

a.kor = 22;

a.fizetes = 1000000;

printf("%s", a.nev);

return 0;

}

A string.h –ban vannak stringkezelő függvények definiálva, egyik hasznos az strcpy(), amely egy string tömbbe belerak egy szövegeget.

Szemely masik = a; ez teljesen megengedett, az értékadás ugyan azon típusú változók között.

**Struktúra mutatók:** A C nyelvben természetesen a struktúrákat is legtöbbször mutatókkal kezeljük.

Például egy Szemely típusú mutató két féle képpen kaphat értéket:

* Egy létező Szemely típusú változónak a címét kapja értékül.

Szemely \*mut = &a;

printf("%s", mut->nev);

* Dinamikus memória foglalással:

Szemely \*mut = (Szemely \*)malloc(sizeof(Szemely));

mut->kor = 10;

printf("%d", mut->kor);

A struktúra is egybefüggő memória területet foglal, mint a tömb, de mivel az egyes mezők kezdő címei csak gépi szó határnál lehetnek (pl. a gépi szó 4 byte, akkor csak 4-el osztható címeken kezdődhetnek mezők), ezért előfordul, hogy nagyobb területet foglal a teljes struktúra, mint az egyes mezők helyfoglalásainak összege. sizeof(struct tipus) >= sizeof(mezo1) + sizeof(mezo2) + …

Ha dinamikusan foglalunk területet, akkor mindig a sizeof (tipust) használjuk!

Hivatkozás mutatón keresztül:

(\*mut).kor = 10;

mut->kor = 10;

A második megoldás is ugyan azt jelenti mint az első, csak az valamelyest olvashatóbb.

scanf\_s("%d", &(\*mut).fizetes);

scanf\_s("%d", &a.fizetes);

printf("mutatosfizetes:%d", mut->fizetes);

printf("simafizetes:%d", a.fizetes);

Az első scanf egy mutatós Személy fizetését kéri be, a másik pedig sima Szemely változóét.

A struktúra függvényeket is érdemes mutatókon keresztül használni:

Szemely \* fv(Szemely \*szemely) {

szemely->fizetes = 100;

szemely->kor = 10;

strcpy\_s(szemely->nev,"Kiss Viktor");

return szemely;

}

fv(&a);

Vigyázat, nehogy egy lokális változó címét adjuk vissza, mert akkor olyan változó címét adjuk vissza, ami megszűnik a függvényblokk elhagyása után, és az a memóriaterület másra fog használódni, az os mást pakol oda, viszont mi azt a címet kapjuk vissza a függvénytől, és azt kezeljük, ott pedig már teljesen más dolog lesz.

**Struktúra összetett szerkezete (struktúrában tömb, struktúrában struktúra):**

typedef struct {

char nev[30];

int kor;

int fizetes;

} Szemely;

Szemely \*mut = &a;

int i = 0;

for (i = 0; i < strlen(mut->nev);i++) {

printf("%c\n", mut->nev[i]);

}

A fenti példában, struktúrában tömb van (karakter tömb), és annak iterálok végig az elemein.

Az alábbiakban, a struktúrában való struktúra tárolást vizsgáljuk:

typedef struct {

int ora;

int perc;

} Ido;

typedef struct {

char esemeny[50];

Ido ido;

} Esemeny;

használata:

Esemeny es;

strcpy\_s(es.esemeny,"almaszedes");

es.ido.ora = 10;

es.ido.perc = 32;

Struktúrában nem lehet olyan tag, mint amit éppen definiálunk, viszont lehet benne olyan mutató.

typedef struct {

Valami a;

} Valami;

De lehet :

typedef struct {

Valami \*a;

} Valami;

A stuktúrában struktúra láncolt lista megvalósításában lehet érdekes.

Ido masodik;

masodik.ora = 30;

masodik.perc = 43;

masodik.kovetkezo = NULL;

Ido elso;

elso.ora = 10;

elso.perc = 21;

elso.kovetkezo = &masodik;

printf("%d", elso.kovetkezo->ora);

Ennek a kiírásnak az eredménye 30 lesz.

**Struktúra tömbök:** Az a lényeg, hogy nem csak gyári elemi típusokból készíthetünk tömböt(integer, double), hanem struktúra típusokból is.

Ido idok[2];

idok[0].ora = 10;

**Union összetett adattípus:** Ez is egy összetett adattípus, amelynek különböző típusú elemei lehetnek és az elemeire névvel hivatkozunk.

typedef union{

int m1;

char m2[4];

} Tip;

Tip a;

a.m1 = 10;

printf("%d", a.m1);

Egy union memória foglalása, mindig akkora mint a mezők közül a leghosszabb memóriafoglalása.

**Filekezelés:**

Lépések:

* file megnyitása
* file olvasása
* file lezárása

File megnyitása: FILE \*fopen(const char \*filename, const char \*mode)

a filename értelem szerűen a megnyitandó file neve

a mode:a file megnyitási módja

* r: létező file olvasásra (ha nem létezik, hiba)
* w: nem létező file írásra (ha létezik felülírja)
* a: létező file továbbírásra (ha nem létezik létrehozza, a file pozíció a file végére áll)
* r+:létező file olvasásra és írásra
* w+:nem létező file olvasásra és írásra
* a+:létező file olvasásra és továbbírásra

másrészt hogy text vagy bináris file-ként akarome megnyitni, „t”: text file-ként „b”: bináris file-ként

rt

rb

az fopen() metódus egy file struktúrának a pointerét adja vissza, vagy NULL értéket, ha nem sikerült megnyitni a filet.

**Fájlok fajtái:**

* szöveges: Az adatokat szövegekké alakítva tárolja. Pl egy egész számot 10000, mint szöveget „10000” tárolja.
* bináris: Az adatokat binárisan tárolja. Olyan formában, ahogyan a memóriában. Pl. egy int számot 4 byteon.

**File olvasó függvények:**

long int ftell(FILE \*stream) , ezzel a file pozíciót kapjuk vissza.

int feof(FILE \*stream) , igazat ad, ha a file végén vagyunk(igaz 1 vagy bármi nem nulla érték)

int fseek(FILE \*stream, long int offset, int whence) , Beállítja a file pozíciót. A whence adja meg, hogy mihez képes, az offset, hogy hova. A whence lehetséges értékei: SEEK\_SET, SEEK\_CUR, SEEK\_END

void rewind(FILE \*stream) , Vissza állítja a file pozíciót a file elejére.

int c;

while ((c = fgetc(fp)) != EOF) {

printf("alma");

printf("%d", c);

}

int fputc(int char, FILE \*stream)