7. gyakorlat

Struktúrák, enumerációk, függvények.

Struktúrák

Az adatstruktúrák segítségével logikailag összetartozó adatokat együtt kezelhetünk. Az adattagok lehetnek különböző adattípusúak. Minden adattag nevet kap, a névnek csak a struktúrán belül kell egyedinek lennie. Az összekapcsolt adatokat együtt, a struktúra példányának nevével tudjuk elérni, mozgatni.

A struktúrákat a fájl tetején (a main függvényen kívül) szokás létrehozni. A struktúra definíciója egyfajta prototípusként szolgál -- az adattagok nevei és típusai szerepelnek benne. A struktúrából ezután a kódunkban példányokat hozhatunk létre, amelyek konkrét adatokat tartalmaznak. A struktúradefiníció utasítás, ezért kell a végére pontosvessző.

Egy egyszerű példa struktúra létrehozására.

```
struct Cim
{
	char orszag[50];
	char telepules[50];
	char kozterulet[50];
	char hazszam[20];
	int iranyitoszam;
};
```

A char [] mezőkben szöveget lehet tárolni.

A struktúra létrehozásakor az adattagokat ugyanúgy deklarációval készítjük el, mint ahogy a main függvényben tennénk. Kezdeti értéket nem adhatunk nekik, hiszen ez a struktúra prototípusa. Az adatok majd a példányokba kerülnek.

A fent létrehozott struktúrából készítsünk egy példányt, és töltsük fel az elemeit! A példányt ugyanúgy kell létrehozni, mint bármely változót. Az adattípus esetünkben `struct Cim`. Két szóból áll, a struct és a struktúranév együtt alkotják az adattípust.

```
int main()
{
	struct Cim BME;
	BME.orszag = "Magyarorszag";
	BME.telepules = "Budapest";
	BME.iranyitoszam = 1111;
	BME.kozterulet = "Muegyetem rkp.";
	BME.hazszam = "3";
	...
}
```

A `struct Cim BME` létrehoz egy példányt, azaz egy `BME` nevű, `struct Cim` típusú változót, üres adattagokkal. A további sorokban az adattagoknak adunk értéket.

Az adattagokat megadhatjuk egyszerre is, a tömbökhöz hasonlóan. Lehet sorrend alapján feltölteni, vagy az adattagok neveit megadva. (Névvel célszerűbb).

```
struct Cim BME1 = {"Magyarorszag", "Budapest", "Muegyetem rkp.", "3", 1111};

struct Cim BME2 = {.orszag="Magyarorszag", .telepules="Budapest",
    .iranyitoszam=1111, .kozterulet="Muegyetem rkp.", .hazszam="3"};
```

A struktúrák elemeit a példány nevének és az adattag nevének együttesével érhetjük el.

```
BME.telepules = "Nyiregyhaza";
printf("%s, %d %s, %s %s.", BME.orszag, BME.iranyitoszam, BME.telepules,
    BME.kozterulet, BME.hazszam);
```

A printf a struktúrát nem tudja kiírni, csak az adattagjait egyesével.

A struktúra létrehozása és példányok létrehozása elvégezhető egyszerre is, ha a záró `}` után írunk példányneveket. Általában jobb a példányosítást külön írni.

```
struct Cim
{
      char orszag[50];
      ...
} BMEcim, BCEcim, OEcim, MOMEcim;
```

Struktúráknak lehet struktúra adattagja is. A belső struktúrát lehet kívül is definiálni, vagy az előbb bemutatott módon belül is.

```
struct Nevjegy
{
      char nev[50];
      char telefonszam[30];
      struct Cim cim;
      char email[50];
};
```

A struktúrák sima változóként működnek, például készíthetünk tömböt is.

```
struct Nevjegy kollegak[100];
```

typedef

A typedefről már volt szó. Segítségével új nevet adhatunk adattípusoknak.

```
typedef réginév újnév;
```

Bevett szokás, hogy a struktúrákat typedeffel átnevezik. A struct kulcsszót így nem kell kiírni többé, hanem elég az új típusnevet használni. Leggyakrabban egyszerűen struktúra nevét használják típusnévként.

```
struct Nevjegy
{
      char nev[50];
      ...
};
typedef struct Nevjegy Nevjegy;
```

```
Nevjegy Bela;
```

Ezután a példányokat `Nevjegy` típussal hozhatjuk létre.

A typdefet a struktúradefiníció köré lehet írni, leggyakrabban így találkozhatunk vele. Ez ugyanazt eredményezi, mint az előző kódrészlet.

```
typedef struct Nevjegy
{
      char nev[50];
      ...
} Nevjegy;
```

A fölső „Nevjegy” helyett írhatnánk akármi mást, de így szokás írni.

Enumerációk

A változókban tárolt adatok sokszor csak meghatározott értékeket vehetnek fel. Ilyenkor jó ötlet kódolni az adatot. A kódolás azt jelenti, hogy a lehetőségek halmazának elemeihez számokat rendelünk, célszerűen nullától növekvő egész számokat szokás használni.

```
Termék színe
```

```
    +--------+     +---+
    | fekete |     | 0 |
    | fehér  |     | 1 |
    | piros  | ==> | 2 |
    | kék    |     | 3 |
    | sárga  |     | 4 |
    +--------+     +---+
```

A kódolásnak két előnye van:

1. Könnyebben használható adattípussal, egész számmal írjuk le az adatot. Kevesebb memóriát foglalunk. Az összehasonlítás jobban működik.
2. Értékadásnál a `szin = "ságra"` kifejezésben a hibát nem könnyen vesszük észre, és a fordító nem fog szólni érte. A kódolt adatnál viszont csak definiált értékeket adhatunk meg, így ilyen hibát nem véthetünk.

Az enum kódolások létrehozására használható. A struktúrához hasonlóan egy új típust hozhatunk létre vele. Az ilyen típusú változó az előre meghatározott értékek egyikét veheti fel. Nézzük meg a színes példával!

```
enum Szin
{
    fekete,
    feher,
    piros,
    kek,
    sarga
};
```

```
enum Szin szin1 = piros;
```

Alapértelmezetten nullától felfelé számozza meg az elemeket. A fekete így 0-t jelent, a feher 1-et, stb. A szin1 változó tartalma 2 lesz.

Kézzel is megadhatjuk a kódokat.

```
enum MachineState
{
    DISCONNECTED = 0,
    STANDBY = 10,
    INIT = 11,
    RUN = 15,
    DIAGNOSTIC = 16,
    SHUTDOWN = 17
};
```

A dekódolást switch-csel lehet csinálni, ha szükség van rá.

```
switch (szin1)
{
    case fekete:
        char *szin = "fekete"; break;
    case feher:
        char *szin = "feher"; break;
    ...
}
```

Függvények

A számítástechnikában a függvényeknek kicsit más a jelentése, mint ami matekból ismerős lehet. Matematikában a függvény alatt relációt értünk, hozzárendelést két halmaz elemei közt. Számítástechnikában a függvény névvel ellátott, újra felhasználható kódrészletet jelent. A függvénynek van 0 vagy több bemenete és 0 vagy 1 kimenete. A függvényeket le lehet futtatni, másnéven *meghívni* a programon belül. Ekkor a bennük lévő kódsorokat lefutnak, majd a vezérlés kilép a függvényből.

Beépített függvényeket már használtunk, de függvényeket mi is készíthetünk. Tekintsük a következő kódot!

```
#include <stdio.h>

int main()
// Pénztárgép
{
    char termek1_nev[] = "kifli";
    int termek1_mennyiseg = 4;
    double termek1_egysegar = 25;

    printf("\n");
    printf("******* NYUGTA *******");
    printf("\n");
```

```
    printf("Lali ABC\n");
    printf("Lali és Társa 2002 Kft.\n");
    printf("Budapest, Bercsényi u. 73/5.");
    printf("\n");

    printf("%s \n x %3d db ..... %8.2lf Ft\n", termek1_nev, termek1_mennyiseg,
        termek1_mennyiseg * termek1_egysegar);
}
```

A precízen megkonstruált nyugta fejléc egy összefüggő kódrész. Felmerülhet bennünk, hogy ezt egy külön függvényben tároljuk. Ennek több előnye is van.

Először is a fejlécet más programba is egyszerűen átvihetjük. Csak át kell másolnunk a függvénydefiníciót, és már használhatjuk is a függvényt. Esetleg egy külső fájlba is elhelyezhetjük, ahonnan több program is használni tudja.

Másik előny, hogy a valószínűleg hosszúra nőtt main függvényünket így le tudjuk rövidíteni. Az összetartozó utasításokat függvényekbe rendezzük, és kivesszük a main-ből. A kódunk így átláthatóbb és követhetőbb lesz.

A függvénykészítés menete: a main fölött(!) definiáljuk az új függvényt, majd a mainben (vagy másik függvényben) meghívjuk. A definíció részei:

```
kimenet_tipus fuggvenynev(tipus param1, tipus param2)
{
    függvénytörzs
    ...
    return kimenet;
}
```

- A függvény típusát a kimenet típusa adja meg. Ha nincs kimenet, ide `void` kerül.
- A paramétereket (bemeneteket) típusaikkal együtt kell felsorolni.
- A függvénytörzsbe kerülnek a függvény által végrehajtott utasítások.
- A nem-void függvények végén a return szolgáltatja a kimeneti adatot. A return azonnal kilép a függvényből.
- A függvénytörzsben létrehozott változók a függvényből való kilépéskor törlődnek.

Az elkészítendő függvényünk mindig ugyanazt kell hogy kiírja, nincs bemenő adata. Kimenete sincs. Típusa tehát `void`, paraméterei pedig nincsenek.

```
#include <stdio.h>

void print_fejlec()
{
    printf("\n");
    printf("******* NYUGTA *******");
    printf("\n");
    printf("Lali ABC\n");
    printf("Lali és Társa 2002 Kft.\n");
    printf("Budapest, Bercsényi u. 73/5.");
    printf("\n");
}
```

```
int main()
// Pénztárgép
{
    char termek1_nev[] = "kifli";
    int termek1_mennyiseg = 4;
    double termek1_egysegar = 25;

    print_fejlec();

    printf("%s \n x %3d db ..... %8.2lf Ft\n", termek1_nev, termek1_mennyiseg,
        termek1_mennyiseg * termek1_egysegar);
}
```

A kódot tovább szemlélve rájöttünk, hogy az áruk nyomtatását is jó lenne függvénybe foglalni, hiszen minden árut ugyanúgy kell a nyugtára írni. Ennek a függvénynek már lesznek bemenő adatai: a megnevezés, a mennyiség és az egységár, mivel a kiíráshoz ezekre szükség van. Kimenő adat továbbra sincs.

```
#include <stdio.h>

void print_fejlec()
{ ... }

void print_arucikk(char nev[], int mennyiseg, double egysegar)
{
    printf("%s \n x %3d db ..... %8.2lf Ft\n", nev, mennyiseg,
        mennyiseg * egysegar);
}

int main()
// Pénztárgép
{
    char termek1_nev[] = "kifli";
    int termek1_mennyiseg = 4;
    double termek1_egysegar = 25;

    print_fejlec();
    print_arucikk(termek1_nev, termek1_mennyiseg, termek1_egysegar);
}
```

A további árucikkek nyomtatása sokkal egyszerűbb lesz, a kiírás formátumával többet nem kell törődnünk. Egyszerűen a print_arucikk függvényt kell újra és újra használnunk.

Készítsünk függvényt ami bruttósítja az árakat! A különböző termékkategóriáknak eltérő az áfája, ezért jó ötlet ezt a számolást fügvényként megírni. Itt lesznek bemeneteink: az ár és a termékkategória, de lesz kimenetünk is, a bruttó ár. A termékkategóriákat kódolva tárolom, enum segítségével.

```
enum NavKategoria {CsirkeComb, CsirkeMell, SertesSzalonna, ...};
```

```c
double brutto(double netto_ar, enum NavKategoria termekkategoria)
{
    double afakulcs;
    switch (termekkategoria)
    {
        case CsirkeComb:
            afakulcs = 5; break;
        case CsirkeMell:
            afakulcs = 27; break;
        case SertesSzalonna:
            afakulcs = 5; break;
        case TehenTej:
            afakulcs = 15; break;
        default:
            afakulcs = 27;
    }
    return afakulcs * netto_ar;
}
```

Paraméterek

A függvény bemenő adatait hívjuk paramétereknek. A C nyelvben a függvény a paraméterek **másolatát** (értékét) kapja meg.

```c
void novel(int i)
{
    i++;
}
int main()
{
    int i = 0;
    novel(i);
    printf("%d", i);
}
```

Mi lesz a kimenet?

A fenti programban a függvény megkísérli növelni az átadott paraméter értékét. A novel függvény az i paraméter másolatát kapja meg, és annak az értékét növeli. A függvényből való kilépéskor ez a belső változó törlődik, a printf ismét a külső i értékét kapja meg, amely nem növekedett. A kimenet 0 lesz.

Az értékként átadott paramétereken a függvény nem tud változtatni. Ez egy nagyon hasznos dolog. A változókat átadhatjuk bármilyen függvénynek, azok nem tudják "tönkretenni" a változóinkat, hisz csak a másolataikat kapják meg.

Általában a bemeneteken nem akarunk változtatni, a függvény eredménye a kimeneten képződik. Előfordul viszont, hogy mégis meg kell változtatni a paraméterként átadott értékeket. Ha például a kimenet hibakód, de szükségünk van más kimenő értékre is; vagy kifejezetten a bemenő adatokon kell módosítást végrehajtanunk. Ilyenkor a paraméter helyett egy rá mutató pointert kérünk be.

```c
void novel(int *i)
{
    *i++;
}
int main()
{
    int i = 0;
    novel(&i);
    printf("%d", i);
}
```

Itt is a paraméter másolatát kapja meg a függvény, de a paraméter az adat memóriacíme. A függvény a memóriacím alapján megtalálja az eredeti adatot, és módosítást hajt végre rajta.

Tömbök átadásánál figyeljünk, mert azoknak a pointerekhez hasonlóan a címét adja át, tehát a függvény az eredeti tömböt módosítja.

Kimenetek

Kimeneteket (visszatérési értékeket) a `return` utasítással adhatunk meg. A kimeneti érték típusának meg kell egyeznie a függvény neve előtt megadott típussal. Egy függvényben több return is lehet (bár nem kívánatos), ilyenkor az első elérése után kilép. Ilyet hibaellenőrzésre szoktunk használni.

```c
double terulet(double a, double b)
// téglalap területe
{
    if (a <= 0 || b <= 0)
        return -1; // nem létezik ilyen téglalap
    double terulet = a * b;
    return terulet;
}
```

Egy függvénynek csak 1 kimenete lehet. Ha több kimenő adatot akarunk, akkor erre két ok lehet:

1. Rosszul akarjuk, nem is kell több kimenet. Bontsuk a feladatot több kisebb függvényre, minden függvény egy részfeladatot lásson el.
2. A kimenő adataink logikailag összefüggenek. Tegyük őket struktúrába, és használjuk azt kimenő adatként!

Példa a struktúra használatára:

```c
struct Trabant
{
    int kivitel; // 0: limuzin  1: kombi
    int szin;    // 0: fehér  1: kék  2: sárga  3: zöld
};

struct Trabant uj_trabant(int kivitel, int szin)
{
    struct Trabant trabant;
    double mazli = (double)rand() / RAND_MAX;
    if (mazli < 0.4)
```

```
    {
        trabant.kivitel = rand() % 2;
        trabant.szin = rand() % 4;
    } else
    {
        trabant.kivitel = kivitel;
        trabant.szin = szin;
    }
    return trabant;
}
```

Függvényprototípusok

A fordító mindig csak azokról a függvényekről tud, amik a fájlban a meghívás helye **előtt** már definiálva lettek. Ha egy függvényt a mainben használunk, de csak alatta definiáljuk, nem fog lefordulni a program.

Ezt kiküszöbölhetjük úgy, hogy már a fájl elején megmondjuk, hogy milyen függvények lesznek majd a fájlban. Ezeket az utasításokat hívjuk függvényprototípusoknak. A prototípus tulajdonképpen a függvénydefiníció első sorának másolata. Az uj_trabant függvénynek ez a prototípusa:

```
struct Trabant uj_trabant(int kivitel, int szin);
```

A prototípus rögzíti a függvény nevét, típusát, és paramétereit. Így már ettől a ponttól lefelé tudjuk használni a függvényt, akárhol is van a definíciója a fájlon belül.

A függvényprototípusokat, struktúradefiníciókat, enumerációkat, konstansokat egy külön .h fájlba szoktuk összegyűjteni. Ennek a neve megegyezik a .c fájléval. A .c fájba include-oljuk így:

```
#include "fajlnev.h"
```

Ezzel a definíciók a fájlunk tetejére másolódnak.

Feladatok

1. Struktúra gyakorlás

1. Hozz létre egy `struct Hallgato` struktúrát! Mezői:
   - teljes név (50 karakter)
   - Neptun kód (6 karakter)
   - szak neve (50 karakter)
   - beiratkozás éve (egész szám)
   - aktív féléven van (igaz = 1 / hamis = 0)
2. Tárold el a saját adataidat egy ilyen struktúrában!
3. Hozz létre egy 120 elemű tömböt `struct Hallgato` adatokból! A 45. elemébe írd be a saját adataidat!
4. Olvasd ki a tömbből a 45. hallgató Neptun kódját!

2. Írj függvényt, ami egy paraméterként átadott struct Hallgato mezőit a képernyőre írja!

Az adattagok kiírására a printf() függvényt használhatod.

Várt kimenet:

```
Név: Kovács Béla
Neptun kód: Z4QQQB
Szak neve: Mechatronikai mérnöki BSc
Beiratkozás éve: 2019
Aktív félév: igen
```

3. Írd át a trabantos példát úgy, hogy enumokat használjon a kivitelre és a színre! Készíts dekódoló függvényt is a kiíráshoz!

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

struct Trabant
{
    int kivitel; // 0: limuzin  1: kombi
    int szin;    // 0: fehér  1: kék  2: sárga  3: zöld
};

struct Trabant uj_trabant(int kivitel, int szin)
{
    struct Trabant trabant;
    double mazli = (double)rand() / RAND_MAX;
    if (mazli < 0.4)
    {
        trabant.kivitel = rand() % 2;
        trabant.szin = rand() % 4;
    } else
    {
        trabant.kivitel = kivitel;
        trabant.szin = szin;
    }
    return trabant;
}

int main()
{
    srand(time(NULL));
    struct Trabant trabi = uj_trabant(1, 1);
    printf("Az új trabid %d kivitelű és %d színű.\n",
        trabi.kivitel, trabi.szin);
    return 0;
}
```