Imperatív programozás

Imperatív programozás

Kozsik Tamás és mások

1 Tantárgyi követelmények

A tárgy célja

- Fogalomrendszer
- Terminológia magyarul és angolul
- Tudatos nyelvhasználat
  - Imperatív programozás
  - Procedurális programozás
  - Moduláris programozás
- Részben: programozási készségek
- Linux és parancssori eszközök használata
  - részlet a Jurassic Parkból (link)
  - és TadeusTaD megjegyzése: `$su root -c "killall raptors"`

A tárgy célja, hogy programozási nyelvekkel kapcsolatos fogalmakkal ismertesse meg a hallgatókat (tudás), melyek alapján a hallgatók a programozás során képesek lesznek tudatosan választani a nyelvi eszközök közül (kompetencia). A tárgyalt ismeretkör az imperatív, a procedurális és (kisebb részben) a moduláris programozási paradigmát fedi le, alapot teremtve későbbi, az objektum-orientált és konkurens programozási paradigmákat tárgyaló kurzusoknak. A tárgy utal a vele egy időben tartott *Funkcionális programozás* kurzusra is (és viszont). A tárgy szoros kapcsolatban áll a vele egy időben tartott *Programozás*, illetve (kisebb mértékben) a *Számítógépes rendszerek* kurzusokkal. Jelen tárgynak nem célja, hogy a hallgatókat programozni tanítsa – ez a *Programozás* kurzus feladata –, de természetesen hozzájárul a hallgatók programozási készségeinek fejlődéséhez.

A tárgy figyelembe veszi, hogy az első félévben szerepel a tantervben, így igyekszik nem építeni meglévő ismeretekre. Másrészt nem „bevezető” kurzus próbál lenni: a meghatározott ismeretkört teljes mélységében (a BSc-záróvizsga szintjén) át kell adja.

A gyakorlatokon Linux operációs rendszert használunk, így a tárgy hozzájárul ahhoz is, hogy a hallgatók felhasználói szinten megismerkedjenek ezzel az operációs rendszerrel, ezen belül is főleg a parancssor használatával. A nyelvi eszközök tudatos használatát azzal is erősíteni kívánja a tárgy, hogy nem integrált fejlesztői környezetben készítjük majd a programokat, hanem közönséges (programozói) szövegszerkesztőkben. Ennek köszönhetően a programírás nem az eszköz által felkínált lehetőségek közüli választás lesz, hanem egy sokkal tudatosabb folyamat.

Használt programozási nyelv: C

(Miért is tanulunk C-t? link!)

A C az egyik legprimitívebb *magas szintű programozási nyelv*, melyet régóta használnak nagyon sokféle alkalmazási területen. Jelenleg is az egyik legerőteljesebben használt nyelv. Az egyszerűségéből fakadóan kicsit macerás programozni benne, de a gépközelisége miatt a programozók igen hatékony kódot tudnak írni benne. Sokszor arra is használják a C nyelvet, hogy a más nyelveken írt programokat először erre a nyelvre fordítsák, majd ebből generáljanak gépi kódot.

A képzés formája

- Előadás
- Gyakorlat
- Konzultáció

A tárgy heti 2x45 perc előadásból, 2x45 perc géptermi gyakorlatból és 45 perc (gépterembe beosztott) önálló munkából. Ez utóbbi során a jelen lévő demonstrátortól segítséget lehet kérni. Mind az előadásokon, mind a gyakorlatokon és konzultációkon kötelező a részvétel. Összesen háromszor lehet büntetlenül hiányozni. A negyedik hiányzás még tolerálható (plusz feladatot jelöl ki a gyakorlatvezető). Ennél több hiányzás a jegy megtagadásával jár.

Számonkérés

- Rendszeresen: tesztek, feladatok
- Félév végén: vizsgazárthelyi

Rendszeresen kell a gyakorlatokon teszteket kitölteni, feladatokat megoldani, ezekre pontokat lehet kapni. A félév végén pedig lesz egy géptermi vizsga, mely programozási és elméleti feladatokat is tartalmaz. Két alkalommal van lehetőség a vizsgán javítani.

A megszerzett pontok alapján kap a hallgató jegyet a kurzusra. **A pontszámítás részleteiről a tárgy honlapja tájékoztat.**

A számonkérés során ügyelni fogunk a plágium detektálására. A plagizálást és puskázást szigorúan büntetni fogjuk, enyhébb esetben pontlevonással, súlyosabb esetben az érdemjegy megtagadásával. Kiemelkedően súlyos esetben fegyelmi eljárást kezdeményezünk a vétkesekkel szemben. Az is vétkes, aki másnak odaadja a saját munkáját, és az is, aki más munkáját a sajátjaként tünteti fel.

Az elvárt munka

Összesen 150 munkaóra

- Tanórák: 13x5
- Gyakorlás, otthoni tanulás, házi feladatok: 12x5
- Készülés vizsgára: 20
- Vizsga: 5

Mivel a tárgy 5 ECTS kreditpontot ér, az átlagos képességű hallgatóktól 5x30, azaz 150 óra munkabefektetést igényel a kurzus elvégzése. Ennek a javasolt beosztását tartalmazza az alábbi lista. A ténylegesen befektetendő munka mennyisége a hallgató képességeitől, valamint a megszerezni kívánt érdemjegy értékétől függ.

- 65 munkaóra a heti 2+2+1 tanórán, 13 héten át;
- 50 munkaóra rendszeres tanulás, gyakorlás, házi feladatok (12 héten át heti 5);
- 20 munkaóra a félév végén a vizsgára készülés (ismétlés);
- 5 munkaóra a félév végén a vizsga.

További információk

A tárgy honlapján:

http://kto.web.elte.hu/hu/oktatas/

A canvasben

http://canvas.elte.hu/

2 Paradigmák és nyelvek

Programozási nyelvek

- Ember-gép kommunikáció
- Ember-ember kommunikáció

A programozási nyelv segítségével vesszük rá a számítógépet arra, hogy kiszámoljon nekünk valamit, illetve, hogy azt csinálja, amit parancsolunk neki. A programozási nyelvnek tehát az az egyik célja, hogy minél jobban megértessük magunkat a számítógéppel. Így a leírt program hatékonyan végrehajtható lesz a számítógépen.

Van azonban egy másik, talán még fontosabb cél is: nagy szoftverek fejlesztésénél a programkód az egyik legfontosabb eszköz a fejlesztők közötti kommunikáció során. Tehát a programozási nyelvnek alkalmasnak kell lennie arra, hogy az egyik ember elmondhassa a gondolatait, és azt egy másik ember könnyen megérthesse. Tehát a programozási nyelvnek az emberi gondolkodási sémákat követőnek (is) kell lennie.

Programozási paradigmák

Gondolkodási sémák, szükséges nyelvi eszközök

Például:

- Imperatív programozás
- Funkcionális programozás
- Logikai programozás

- Szekvenciális programozás
- Konkurens programozás
- Párhuzamos programozás
- Elosztott programozás

- Procedurális programozás
- Moduláris programozás
- Objektumelvű programozás

- Aspektuselvű programozás
- Komponenselvű programozás
- Szolgáltatáselvű programozás
- Szerződésalapú programozás

A programozási paradigmák két kérdésre próbálnak válaszolni.

- Milyen gondolkodási sémát kívánunk kifejezni?
- Milyen eszközökre van szükség ehhez?

Sokféle programozási paradigma létezik: ezek a kurzuson az imperatív és a procedurális paradigmával foglalkozunk részletesebben, de a moduláris programozás témakörébe is betekintünk. Ugyanebben a

félévben egy másik tárgyból a funkcionális programozási paradigma kerül bemutatásra. A logikai programozással inkább a mesterszakon lehet találkozni. A következő félévben az objektumelvű programozással foglalkozunk majd. Az alapszakon előkerül még a konkurens programozás is, míg a párhuzamos és elosztott programozás inkább a mesterszakon kerül elő. A negyedik blokkban felsorolt paradigmákkal is találkozhatunk a mesterszakon.

Egy programozási nyelv akár több paradigmát is támogathat.

Imperatív programozás

Akkor beszélünk imperatív programokról, amikor explicit mi vezéreljük, hogy a program hogyan változtatja meg az állapotát. A program utasítások sorozataként van megadva, melyeket egymás után végrehajtunk. Az utasítások a számítógép memóriájába írhatnak, onnan olvashatnak. A memória pillanatnyi tartalma határozza meg a program *állapotát*.

Procedurális programozás

A megoldandó feladatot felbonthatjuk az elvégzendő feladatok (algoritmusok) szerint. Ezeket alprogramokként (függvények, eljárások) valósítjuk meg, köztük pl. paraméterátadással, függvény visszatérő értékével kommunikálunk. Ez a procedurális programozás. Ebben az esetben probléma lehet, hogy háttérbe szorulnak az adatszerkezetek. Pl. FORTRAN, Algol60, C, Go nyelvek.

Kezdetben döntően procedurális nyelvek léteztek, hiszen az assembly programok, a FORTRAN, COBOL, Algol60 és társai ilyen elvek mentén épültek fel, bár a Lisp 1957-ben már funkcionális nyelv volt.

Objektumelvű programozás

Amikor a valós világ objektumait próbáljuk modellezni, akkor összegyűjtjük a hasonló tulajdonságúakat, elhanyagoljuk a feladat szempontjából kevéssé fontos különbségeket és absztrakció segítségével egymással egy szűk interfészen kommunikáló osztályokat alkotunk belőlük. Itt az osztályok adatszerkezetén és a rajtuk értelmezett műveleteken van a hangsúly. Ez az objektumelvű (object-oriented) programozás. Pl. Simula67, Smalltalk, Eiffel, Java, C#.

Deklaratív programozás

Más esetekben egyszerűen csak deklarálni akarjuk a program elvárt működését, nem akarjuk explicit meghatározni annak mikéntjét. Ez a deklaratív programozás, amit több kategóriára szoktak bontani.

Funkcionális programozás

A kívánt eredmény egymást hívó függvényekként van definiálva. Ezek a függvények mellékhatás-mentesek, nincsen értékadás, minden memóriaterület egyszer kap csak értéket, és később ez az érték nem változik (referencial transparency). Az ilyen programok helyességét könnyebb belátni. A számításokat, függvényeket könnyen át lehet adni paraméterként ún. magasabb rendű függvényeknek. Ilyen nyelvek pl. Lisp, ML, Haskell, Clean.

Logikai programozás

A rendszer tényeit és következtetési szabályait adjuk meg. Pl. Prolog.

2.1 Alacsony szintű és magas szintű programozás

Gépi kód

Eleinte a számítógépek programozása úgy történt, hogy bedrótozták (de tényleg, a szó szoros értelmében) a programot. Minden újabb program betáplálása újradrótozást igényelt. Ehhez képest már nagy előrelépést jelentett, amikor újradrótozás nélkül lehetett bevinni a gép memóriájába a gépi kódot. A gépi kódot is nehéz programozási *nyelvként* felfogni.

Assembly

```
quickSort:
.LFB1:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $40, %esp
        movl    12(%ebp), %eax
        cmpl    16(%ebp), %eax
        jge     .L6
        movl    16(%ebp), %eax
        movl    %eax, 8(%esp)
        movl    12(%ebp), %eax
        movl    %eax, 4(%esp)
```

Az assembly nyelvek már egyértelműen programozási nyelvnek tekinthetők. Nagyon közel vannak a számítógép utasításaihoz: a gépi kódhoz könnyebben megjegyezhető, szöveges parancsokat határoz meg.

„Magas szintű” programozási nyelvek

- Fortran
- LISP
- Algol
- COBOL
- BASIC
- **C**

stb.

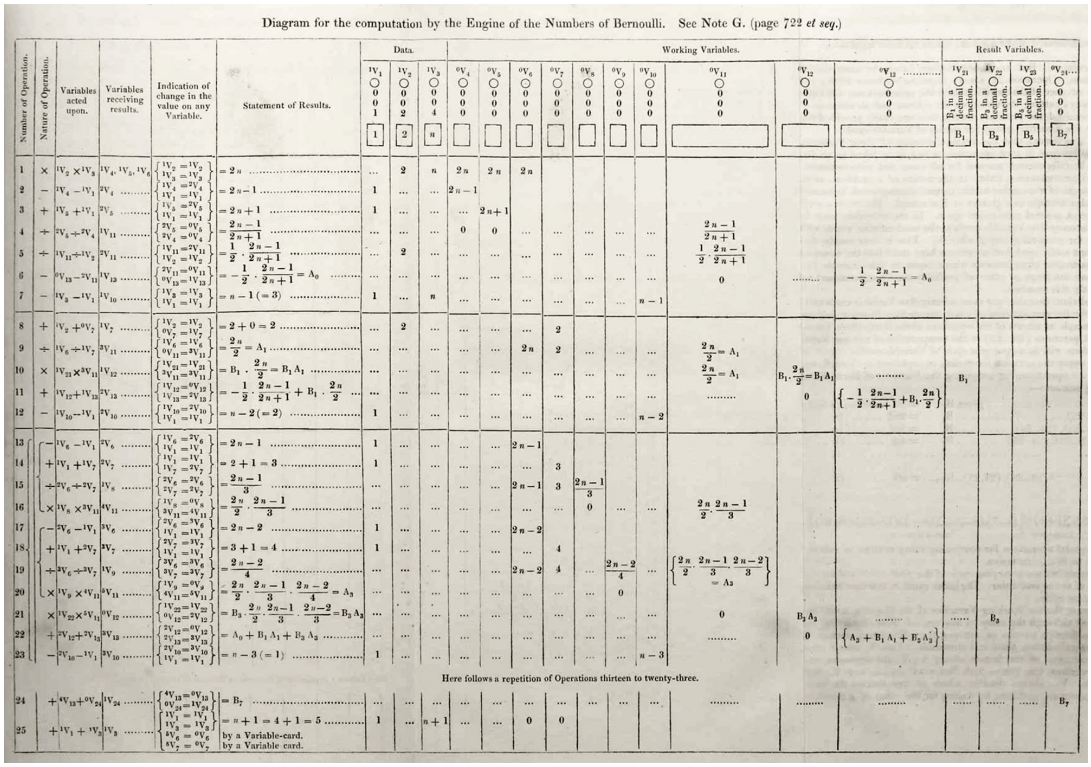
Modern, kényelmes nyelvek

- Python
- **Haskell**
- C++
- Java
- Ada

stb.

2.2 Programozási nyelvek történelme

Ada Lovelace (Analytical Engine, Charles Babbage)



Augusta Ada King, Countess of Lovelace (née Byron, 1815–1852)

A programozás őskora

- Fizikai huzalozás (pl. ENIAC, 1945)
- Gépi kód (Neumann-architektúra, 1945)
- Assembly (1949–)
- Magas szintű programozási nyelvek
  - Plankalkül (Konrad Zuse, 1942–1945)
  - Fortran (John Backus et al., 1954)
  - LISP (John McCarthy, 1958)
  - Algol (1958, 1960, 1968)
  - COBOL (1959)
  - BASIC (Kemény–Kurtz, 1964)

Néhány fontos nyelv

- Simula-67 (Dahl–Nygaard, 1967)
- Pascal (Niklaus Wirth, 1970)
- C (Dennis Ritchie, 1972)
- Ada (1980)

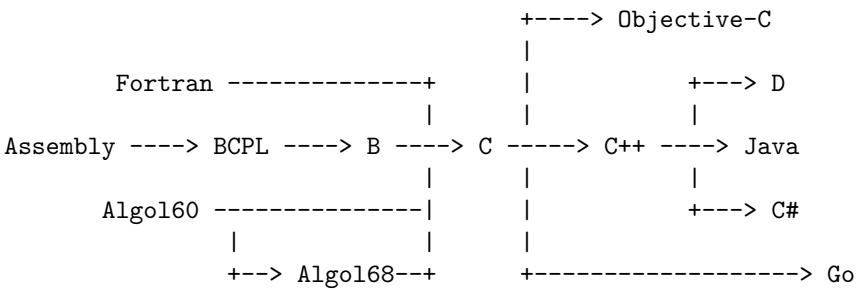
- SQL (Chamberlin–Boyce, 1974)
- C++ (Bjarne Stroustrup, 1985)
- Eiffel (Bertrand Meyer, 1986)
- Erlang (Armstrong–Virding–Williams, 1986)
- Haskell (1990)
- Python (Guido van Rossum, 1990)
- Java (James Gosling, 1995)
- JavaScript (Brendan Eich, 1995)
- PHP (Rasmus Lerdorf, 1995)
- C# (2000)
- Scala (Martin Odersky, 2004)

A Simula-67 volt az első objektum-orientált programozási nyelv. A Pascal is nagyon jelentős, számos másik nyelv származik belőle. Oktatási céllal ma is használják, mert egyszerű és logikus. A többi nyelvet azért soroltam fel, mert ezeket mind tanítjuk az IK-n. Lehet, hogy másokat is, de most ez jutott az eszembe.

Legnépszerűbb nyelvek (2018. szeptember, TIOBE-index)

Sep 2018	Sep 2017	Change	Programming Language	Ratings	Change
1	1		Java	17.436%	+4.75%
2	2		C	15.447%	+8.06%
3	5	^	Python	7.653%	+4.67%
4	3	v	C++	7.394%	+1.83%
5	8	^	Visual Basic .NET	5.308%	+3.33%
6	4	v	C#	3.295%	-1.48%
7	6	v	PHP	2.775%	+0.57%
8	7	v	JavaScript	2.131%	+0.11%
9	-	^^	SQL	2.062%	+2.06%
10	18	^^	Objective-C	1.509%	+0.00%
11	12	^	Delphi/Object Pascal	1.292%	-0.49%
12	10	v	Ruby	1.291%	-0.64%
13	16	^	MATLAB	1.276%	-0.35%
14	15	^	Assembly language	1.232%	-0.41%
15	13	v	Swift	1.223%	-0.54%
16	17	^	Go	1.081%	-0.49%
17	9	vv	Perl	1.073%	-0.88%
18	11	vv	R	1.016%	-0.80%
19	19		PL/SQL	0.850%	-0.63%
20	14	vv	Visual Basic	0.682%	-1.07%

A C nyelv kialakulása



A C egy általános célú programozási nyelv, melyet Dennis Ritchie fejlesztett ki Ken Thompson segítségével 1969 és 1973 között a UNIX rendszerekre AT&T Bell Labs-nál. Idővel jóformán minden operációs rendszerre készítettek C fordítóprogramot, és a legnépszerűbb programozási nyelvek egyikévé vált. Rendszerprogramozáshoz és felhasználói programok készítéséhez egyaránt jól használható. Az oktatásban és a számítógép-tudományban is jelentős szerepe van.

A C minden idők legszélesebb körben használt programozási nyelve, és a C fordítók elérhetők a ma elérhető számítógép-architektúrák és operációs rendszerek többségére. (from wikipedia).

A C nyelv fejlődése

- 1969 Ken Thompson kifejleszti a B nyelvet (egy egyszerűsített BCPL)
- 1969 Ken Thompson, Dennis Ritchie és mások elkezdenek dolgozni a UNIX-on
- 1972 Dennis Ritchie kifejleszti a C nyelvet
- 1972-73 UNIX kernel-t újraírják C-ben
- 1977 Johnson Portable C Compiler-e
- 1978 Brian Kernighan és Dennis Ritchie: The C Programming Language könyve
- 1989 ANSI C standard (C90) (32 kulcsszó)
- 1999 ANSI C99 standard (+5 kulcsszó)
- 2011 ANSI C11 standard (+7 kulcsszó)
- 2018 C18 ISO/IEC standard

Mi alapvetően az ANSI C-t, azaz a C90-et fogjuk használni.

3 Programok felépítése

Programok felépítése

- Kifejezések
- Utasítások
- Alprogramok (függvények/eljárások, rutinok, metódusok)
- Modulok (könyvtárak, osztályok, csomagok)

A legtöbb programozási nyelvben a kód felépítéséhez használt főbb szerkezeti elemek a következők.

- Kifejezés – **expression**: például `v+1`
- Utasítás: értékadás, elágazás, ciklus stb.
- Alprogram: egy jól meghatározott számítási feladat (újrafelhasználható) megoldása. Különböző *paraméterekkel* elvégezhető ugyanaz a számítás a program különböző pontjain.
- Modul: egy összetettebb probléma megoldásához szükséges adatszerkezetek és algoritmusok gyűjteménye (egységbe foglalása, egységbe zárása).

Példa

```
int factorial( int n )
{
    int result = 1;
    int i;
    for(i=2; i<=n; ++i)
    {
        result *= i;
    }
    return result;
}
```

Kifejezések

```
   n
                                        "Hello world!"

                    100

                                    n+1

           ++i
```

```
                                                range(2,n+1)


            employees[factorial(3)].salary * 100
```

Melyik kifejezés a kakukktojás?

Utasítások

```
   result = 1;

           result *= i;
                                       return result;


       for( i=2; i<=n; ++i ){ result *= i; }



                     while(1) printf("Gyurrrika szép!\n");
```

Egyszerű utasítások

- értékadás (kifejezéskiértékelő utasítás)
- üres utasítás
- alprogramhívás
- visszatérés függvényből

Vezérlési szerkezetek

- elágazások
- ciklusok stb.

```
int gcd( int n, int m )
{
    while( n != m )
        if( n > m )
            n -= m;
        else
            m -= n;
    return n;
}
```

Sok mindent megfigyelhetünk ebben a kódrészletben, nem csak a kifejezések és utasítások használatát. A kódban szerepelnek típusok. Zárójelekbe írjuk a feltételeket a ciklusban és az elágazásban is. A *függvény törzsét* kapcsos zárójelek közé írjuk, így választjuk el a *függvény fejlécétől* (*specifikációjától*) – a kettő együtt a *függvény definíciója*.

Sokat segít a kód olvashatóságán az indentálás, és az a konvenció, hogy mindent ugyanolyan szépen, tökéletesen indentálunk. Rosszul indentált C program is lehet tökéletesen helyes: az egymás mellett álló *fehér szóközök* (*whitespace*) valójában csak egynek számítanak – sok helyre beírhatjuk őket, sok helyről törölhetjük őket. (Találd meg a lenti C kódban azt, hogy hol nem lehet kitörölni a szóközt.)

Az alábbi C kód tökéletesen helyes.

```
int gcd( int n,
         int m ){
    while(
 n !=
```

```
            m )if(n>m)n-=m;else m
-= n;
    return n;}
```

Természetesen senki nem ír ilyen kódot, csak maximum heccből. Kínosan ügyelünk a helyes indentálásra – mintha a program helyessége múlna ezen.

Kapcsos zárójelek vezérlési szerkezetekben

Elhagyott kapcsos zárójelek

```
int gcd( int n, int m )
{
    while( n != m )
        if( n > m )
            n -= m;
        else
            m -= n;
    return n;
}
```

Bolondbiztos megoldás

```
int gcd( int n, int m )
{
    while( n != m ){
        if( n > m ){
            n -= m;
        } else {
            m -= n;
        }
    }
    return n;
}
```

Sokan, sok helyen megkövetelik a kapcsos zárójelek redundáns, de bolondbiztos használatát. Ha egy vezérlés szerkezet törzsében (pl. ciklusmagban vagy elágazás egy ágában) csak egy utasítás szerepel, felesleges kiírni a kapcsos zárójeleket köré. Viszont segít bizonyos programozási hibák elkerülésében, ha ilyen esetben is kiírjuk azokat. Az adott környezet (munkahely, csapat) meghatározhatja ezt a biztonságra törekvő stílust (coding style) követelményként, betartandó konvencióként (coding convention).

Csellengő else (dangling else)

Ezt írtam

```
if( x > 0 )
   if( y != 0 )
      y = 0;
else
   x = 0;
```

Ezt jelenti

```
if( x > 0 )
   if( y != 0 )
      y = 0;
```

```
   else
      x = 0;
```

Ezt akartam

```
if( x > 0 ){
   if( y != 0 )
      y = 0;
} else
   x = 0;
```

Lásd még...

goto-fail (Apple) link!

Kiírás a szabványos kimenetre

Kiírunk egy egész számot és egy soremelést (*newline*)

```
printf("%d\n",factorial(10));
```

Bonyolultabb kiírás

```
printf("10! = %d, ln(10) = %f\n", factorial(10), log(10));
```

Ha kicsit összetetteb kimenetet szeretnénk előállítani, akkor lehet a `printf` művelettel ilyet is csinálni.

Típusok

- Kifejezik egy bitsorozat értelmezési módját
- Meghatározzák, milyen értéket vehet fel egy változó
- Megkötik, hogy műveleteket milyen értékekkel végezhetünk el

C-ben

- `int` – egész számok egy intervalluma, pl. $[-2^{63} .. 2^{63} - 1]$
- `float` – racionális számok egy részhalmaza
- `char` – karakterek a *kiterjesztett ASCII* jelkészletben
- `char[]` – szövegek, karakterek tömbje
- `int[]`– egész számok tömbje
- `int*` – mutató (pointer) egy egész számra

stb.

Típus szerepe

- Védelem a programozói hibákkal szemben
- Kifejezik a programozók gondolatát
- Segítik az absztrakciók kialakítását
- Segítik a hatékony kód generálását

Típusellenőrzés

- A változókat, függvényeket a típusuknak megfelelően használtuk-e
- A nem típushelyes programok értelmetlenek

Statikus és dinamikus típusrendszer

A C fordító ellenőrzi *fordítási időben* a típushelyességet

Erősen és gyengén típusos nyelv

- Gyengén típusos nyelvben automatikusan konvertálódnak értékek más típusúra, ha kell
  - Eleinte kényelmes
  - De könnyen írunk mást, mint amit szerettünk volna
- A C-ben viszonylag szigorúak a szabályok (elég erősen típusos)

A programozási nyelvek egy részénél a fordítóprogram már a fordítási időben minden egyes részkifejezésről el tudja dönteni, hogy az milyen típusú. Ezeket a nyelveket statikus típusrendszerrel rendelkezőnek nevezzük. Ennek vannak előnyei, hiszen a nyelv alaposabb ellenőrzéseket tud végrehajtani és optimálisabb kódot is tud generálni. Ilyen nyelv a Haskell, Fortran, Algol, C, Pascal, C++, Java, C#, Go.

Más nyelveknél, legtöbbször az interpretált nyelveknél, egy változó idővel más típusú értékekre is hivatkozhat. Ilyenkor a fordító futási időben kezeli a típusinformációkat. Ezt dinamikus típusrendszer-nek nevezzük. Ilyen nyelv pl. a Python, a SmallTalk, a JavaScript, az Erlang.

Mindez nem jelenti, hogy a dinamikus típusrendszer nem ellenőrizheti a típusok alkalmazását, sőt helytelen alkalmazás hibát okozhat. Azokat a nyelveket, ahol ilyen hibák előfordulnak erősen típusos-nak nevezzük, szemben a gyengén típusos nyelvekkel.

A C erősen típusos statikus típusrendszerrel rendelkező nyelv, a Python erősen típusos dinamikus típusrendszerű.

Alprogramok (subprograms)

- Több lépésből álló számítás leírása
- Általános, paraméterezhető, újrafelhasználható
- A program strukturálása – komplexitás kezelése
  - egy képernyőoldalnál ne legyen hosszabb
- Különböző neveken illetik
  - rutin (routine vagy subroutine)
  - függvény (function): kiszámol egy értéket és “visszaadja”
  - eljárás (procedure): megváltoztathatja a program állapotát
  - metódus: objektum-orientált programozási terminológia

Főprogram (main program)

Ahol a program végrehajtása elkezdődik

C

Egy megfelelő nevű alprogram: `main`

```
int main()
{
    int half = 21;
    printf("%d\n",2*half);
    return 0;                /* sikeres végrehajtás */
}
```

Megjegyzés

```
int main()
{
    int half = 21;
```

```
    printf("%d\n",2*half);
    return 0;                 /* itt így írok megjegyzést */
}
```

Modul

Modularitás: egységbe zárás, függetlenség, szűk interfészek

- Újrafelhasználható programkönyvtárak
  - pl. a nyelv szabványos könyvtára (standard library)
- A program nagyobb egységei
- Absztrakciók megvalósítása

A modularitás azt jelenti, hogy a programot felbonthatjuk részekre, melyek viszonylag függetlenek egymástól, a kapcsolataik pedig szabályozottak: egy szűk interfészen keresztül történik. A modul egységbe zárja (encapsulation) a logikailag összetartozó, erősen összefüggő dolgokat, és a külvilág felé minimalizálja a függőségi kapcsolatokat. A modulok azért fontosak, mert a programkód komplexitását (bonyolultságát) tudjuk a segítségükkel kordában tartani. A rendszer megfelelő tagolása, logikus, strukturált felépítése, a szűk interfészek mind csökkentik a komplexitást.

A legszigorúbb értelemben azt a rendszerfelbontást szokták modulárisnak nevezni, amelyben a modulok egymásra épülnek, így például körkörös függés nem is alakulhat ki közöttük.

Egy másik értelmezése a szónak arra utal, hogy egy moduláris rendszerben egy modult könnyű kicserélni egy vele azonos szolgáltatásokkal rendelkező másik modulra.

Mi itt most egyelőre a lehető legáltalánosabb értelmezését vesszük a modul szónak: a program (alprogramoknál nagyobb) egységekre bontását, újrafelhasználható egységek, könyvtárak kialakítását, absztrakciók (például objektum-orientált nyelvekben osztályok) megvalósítását.

Modulokra bontás

Újrafelhasználható `factorial`

- `factorial.c` – a `factorial` függvényt
- `tenfactorial.c` – a főprogramot

tenfactorial.c

```
#include <stdio.h>

int factorial( int n );  /* deklaráljuk ezt a függvényt */

int main()
{
    printf("%d\n", factorial(10));
    return 0;
}
```

Ha a `factorial` függvényt több programban is szeretném használni, praktikus lehet szétbontani a programot több "modulra". Ezeket C-ben két külön forrásfájlba szervezem.

- A `factorial.c` tartalmazza a `factorial` függvényt.
- A `tenfactorial.c` tartalmazza a főprogramot.

A kettő közötti kapcsolat a `factorial` függvény deklarációjában testesül meg a `tenfactorial.c` állományban. Ezzel a deklarációval jelzem, hogy valahol máshol (egy másik modulban) meg van adva ennek a függvénynek a definíciója.

4 Programok fordítása és futtatása

Forráskód

- Programozási nyelven írt kód
- Számítógép: gépi kód
- Végrehajtás
  - interpretálás
  - fordítás, futtatás
- Forrásfájl, pl: `factorial.c`

Itt beszélhetünk a Javáról is, miszerint fordítás plusz interpretálás...

5

Parancsértelmező (interpreter)

- Forráskód feldolgozása utasításonként
  - Ha hibás az utasítás, hibajelzés
  - Ha rendben van, végrehajtás
- Az utasítás végrehajtása: beépített gépi kód alapján

Hátrányok

- Futási hiba, ha rossz a program (ritkán végrehajtott utasítás???)
- Lassabb programvégrehajtás

Előnyök

- Programírás és -végrehajtás integrációja
  - REPL = Read-Evaluate-Print-Loop
  - Prototípus készítése gyorsan
- Kezdők könnyebben elsajátítják

Forrásfájl C-ben

factorial.c

```c
#include <stdio.h>

int factorial( int n ){
    int result = 1;
    int i;
    for(i=2; i<=n; ++i){
        result *= i;
    }
    return result;
}

int main(){
    printf("%d\n", factorial(10));
    return 0;
}
```

Fordítás és futtatás szétválasztása

- Sok programozási hiba kideríthető a program futtatása nélkül is
- Előre megvizsgáljuk a programot
- Ezt csak egyszer kell (a *fordítás* során)
- Futás közben kevesebb hiba jön elő
- Cél: hatékony és megbízható gépi kód!

"Fordítási idő" és "futási idő"

Fordítás

- forráskód (source code) forrásfájlban (source file)
  - `factorial.c`
- fordítóprogram (compiler)
  - `gcc -c factorial.c`
- tárgykód (target code, object code)
  - `factorial.o`

Egy programozási nyelv egy fordítóprogramja a forráskódot (több lépésben) ún. tárgykóddá (object code) fordítja. A tárgykód már az adott hardvernek megfelelő gépi kódú utasításokat tartalmazza, optimalizált, de még tartalmaz(hat) fel nem oldott hivatkozásokat, pl. globális változókra vagy meghívott, de máshol implementált függvényekre. A tárgykód már nyelvfüggetlen formátum, akár különböző nyelvekből készült tágykódok (C, Pascal, Fortran) is együttműködhetnek.

Fordítási egység

(compilation unit)

- a forráskód egy része (pl. egy modul)
- egyben odaadjuk a fordítónak
- tárgykód keletkezik belőle

Egy program több fordítási egységből szokott állni!

C-ben

Egy forrásfájl tartalma

Szerkesztés, végrehajtható kód

- tárgykódok (target code, object code)
  - `factorial.o` stb.
- szerkesztőprogram (linker)
  - `gcc -o factorial factorial.o`
- végrehajtható kód (executable)
  - `factorial`
  - alapértelmezett név: `a.out`

Sok tárgykódból lesz egy végrehajtható kód!

Végrehajtás

```
./factorial
```

A hivatkozásokat a szerkesztő (linker) oldja fel, más tárgykódokból, vagy könyvtárakból. A könyvtár (library) lényegében szerkesztésre optimalizált tárgykódok halmaza (tárgykódból is készítjük el). Ilyen szerkeztéskor alkalmazott könyvtár a nyelv szabványos könyvtára (standard library).

A szerkesztés történhet statikusan, amikor a végrehajtható (executable) állományba belekerül a hivatkozott kód. A másik mód, a dinamikus szerkesztés, ekkor a végrehajtandó állományba csak egy kis kódrészlet kerül be, és a hivatkozott kód a program futási idejében kerül feloldásra.

Unix rendszerekben a tárgykódok konvencionálisan .o kiterjesztésűek (Windows-on .obj), a statikus könyvtárak .a (archive) (Windows-on .LIB), a dinamikus könyvtárak .so (shared object) (Windows-on .DLL) kiterjesztésűek.

Több fordítási egység

factorial.c

```
int factorial( int n )
{
    int result = 1, i;
    for(i=2; i<=n; ++i)
    {
        result *= i;
    }
    return result;
}
```

tenfactorial.c

```
#include <stdio.h>

int factorial( int n );

int main()
{
    printf("%d\n", factorial(10));
    return 0;
}
```

Fordítás, szerkesztés, futtatás

```
gcc -c factorial.c tenfactorial.c
gcc -o factorial factorial.o tenfactorial.o
./factorial
```

A két lépés összevonható egy parancsba

- forráskód forrásfájl(ok)ban
  - `factorial.c` és `tenfactorial.c`
- fordítóprogram és szerkesztőprogram végrehajtása
  - `gcc -o factorial factorial.c tenfactorial.c`
- végrehajtható kód (executable)
  - `factorial`

Fordítási hibák

- Nyelv szabályainak megsértése
- Fordítóprogram detektálja

factorial.c

```
int factorial( int n )
{
    int result = 1;
    for(i=2; i<=n; ++i)
    {
        result *= i;
    }
    return result;
}
```

gcc -c factorial.c

```
factorial.c: In function ‘factorial’:
factorial.c:6:9: error: i undeclared (first use in this function)
     for(i=2; i<=n; ++i)
         ^
```

Ha a fordítási egység nem felel meg a nyelv szabályainak, és a fordítóprogram ezt kiszúrja, akkor fordítási hibát (compilation error) kapunk. Előnye: szigorú nyelvben nehéz rossz programot írni. Az itt látható fordítási egységben elfelejtettük definiálni az `i` változót, kaptunk is egy fordítási hibát. Nem mindig ilyen könnyű megfejteni a hibaüzenetet. Az a minimum, hogy tudunk angoluk. Ezek a fordítási hibák és a hibaüzenetek elég általánosak, sok nyelvben hasonlítanak egymásra, de azért elég nyelvspecifikusak is tudnak lenni. Sokszor bele kell lapozni a nyelv dokumentációjába is, hogy megértsük, mi a baj.

Szerkesztési hibák

factorial.c

```
int factorial( int n )
{
    int result = 1, i;
    for(i=2; i<=n; ++i)
    {
        result *= i;
    }
    return result;
}
```

tenfactorial.c

```
#include <stdio.h>

int faktorial( int n );

int main()
{
    printf("%d\n", faktorial(10));
    return 0;
}
```

Fordítás, szerkesztés, hiba

```
$ gcc -c factorial.c tenfactorial.c
$ gcc -o factorial factorial.o tenfactorial.o
tenfactorial.o: In function `main':
tenfactorial.c:(.text+0xa): undefined reference to `faktorial'
collect2: error: ld returned 1 exit status
```

Fordítási egységek között lehet inkonzisztencia, amit a szerkesztőprogram észrevesz, így szerkesztési hibát (linkage error) kapunk. Például: az egyik fordítási egységben definiáljuk a `factorial` függvényt, és a másikban `faktorial` névvel próbáljuk meghívni.

Fordítási és futási idejű szerkesztés

Statikus szerkesztés

- Még a program futtatása előtt
- A tárgykódok előállítása után "egyből"
- Előnye: szerkesztési hibák fordítási időben

Dinamikus szerkesztés

- A program futtatásakor
- Dinamikusan szerkeszthető tárgykód
  - Linux *shared object*: `.so`
  - Windows *dynamic-link library*: `.dll`
- Előnyei
  - kisebb végrehajtható állomány
  - kevesebb memóriafogyasztás

Előfeldolgozás

C preprocessor: (forráskódból) forráskódot állít elő

Makrók

```
#define WIDTH 80
...
char line[WIDTH];
```

Deklarációk megosztása

```
#include <stdio.h>
...
printf("Hello world!\n");
```

Feltételes fordítás

```
#ifdef FRENCH
printf("Salut!\n");
#else
printf("Hello!\n");
#endif
```

Programok C-ben

Fordítási idő

- Forrásfájlok (`.c` és `.h`)
- Előfeldolgozás
- Fordítási egységek
- Fordítás
- Tárgykódok
- Statikus szerkesztés
- Futtatható állomány

Futási idő

- Futtatható állomány, tárgykódok
- Dinamikus szerkesztés
- Futó program

6 Programozási nyelvek definíciója

6.1 Szabályok

Programozási nyelv szabályai

- Lexikális
- Szintaktikus
- Szemantikus

Egy programozási nyelv definiálása során le kell rögzítenünk a nyelv szabályait, melyek meghatározzák, hogy mik a „jó programok". A nyelv szabályait három fő kategóriába szokták osztani.

Lexikális szabályok

Milyen építőkockái vannak a nyelvnek?

- Kulcsszavak: `while`, `for`, `if`, `else` stb.
- Operátorok: `+`, `*`, `++`, `?:` stb.
- Zárójelek és elválasztó jelek
- Literálok: `42`, `123.4`, `44.44e4`, `"Hello World!"` stb.
- Azonosítók
- Megjegyzések

Case-(in)sensitive?

A nyelv lexikális szabályai azt határozzák meg, hogy a nyelvben mik a morfémák, azaz a legkisebb, már önálló jelentéssel bíró építőkockák. Ezek az építőkockák egy programozási nyelvben a kulcsszavak, az operátorok, a zárójelek, vesszők/pontosvesszők, literálok, azonosítók és megjegyzések.

Például C-ben az alábbi kulcsszavak vannak:

Egy nagyon fontos lexikális kérdés még az, hogy a nyelv érzékeny-e a kis- és nagybetűkre. Vannak *case-insensitive* nyelvek, például az Ada, amelyek nem tesznek különbseget a kis- és nagybetűk között, így például elfogadják kulcsszónak a `while` és a `wHiLE` karaktersorozatot is. A C nyelv viszont *case-sensitive*.

Literál: egész szám

- decimális alak: `42`
- oktális és hexadecimális alak: `0123`, `0xCAFE`
- előjel nélküli ábrázolás: `34u`
- több biten ábrázolt: `99L`
- és kombinálva: `0xFEEL`

Literál: lebegőpontos szám

- triviális: 3.141593
- exponenssel: 31415.93E-4
- több biten ábrázolt: 3.14159265358979L
- és kombinálva: 31415.9265358979E-4L

Literál: karakter és sztring

- karakterek: `'a'`, `'9'`, `'$'`
- sztringek: `"a"`, `"almafa"`, `"1984"`
- escape-szekvenciák: `'\n'`, `'\t'`, `'\r'`, `"\n"`, `"\r\n"`
- több részből álló string: `"alma" "fa"`
- több sorba írt string:

```
"alma\
fa"
```

Azonosító

- Alfanumerikus
- Ne kezdődjön számmal
- Lehet benne `_` jel?

Jó

- `factorial`, `i`
- `computePi`, `open_file`, `worth2see`, `Z00`
- `__main__`

Rossz

- `2cents`
- `fifty%`
- `nőnemű` és $A\theta\acute{\eta}\nu\alpha$ (bár jók pl. Javában)

Az azonosítók (identifier) a legtöbb nyelvben alfanumerikusak lehetnek, azaz betűkből és számjegyekből épülhetnek fel. Egy fontos szabály szokott lenni, hogy betűvel kell kezdődniük. Ezen kívül sok nyelvben nem használhatunk speciális jeleket vagy nem ASCII betűket azonosítókban.

A dupla aláhúzás jellel kezdődő azonosítókat C-ben speciális célokra használják, így az általunk írt deklarációkban nem szoktunk ilyen azonosítót megadni.

Szintaktikus szabályok

Hogyan építkezhetünk?

- Hogyan épül fel egy ciklus vagy egy elágazás?
- Hogyan néz ki egy alprogram? stb.

Backus-Naur form (Backus normal form) – BNF

```
<statement> ::= <expression-statement>
              | <while-statement>
              | <if-statement>
              | …

<while-statement> ::=  while (<expression>) <statement>

<if-statement> ::=  if (<expression>) <statement>
                  <optional-else-part>
```

```
<optional-else-part> ::= ""
                       | else <statement>
```

Szemantikus szabályok

Értelmes, amit építettünk?

- Deklaráltam a használt változókat? (C)
- Jó típusú paraméterrel hívtam a műveletet?

stb.

6.2 Típus

A típus szerepe

- Védelem a programozói hibákkal szemben
- Kifejezik a programozók gondolatát
- Segítik az absztrakciók kialakítását
- Segítik a hatékony kód generálását

Típusellenőrzés

- A változókat, függvényeket a típusuknak megfelelően használtuk-e
- A nem típushelyes programok értelmetlenek

Statikus és dinamikus típusrendszer

- A C fordító ellenőrzi *fordítási időben* a típushelyességet
- Egyes nyelvekben *futási időben* történik a típusellenőrzés

Erősen és gyengén típusos nyelv

- Gyengén típusos nyelvben automatikusan konvertálódnak értékek más típusúra, ha kell
  - Eleinte kényelmes
  - De könnyen írunk mást, mint amit szerettünk volna
- A C-ben viszonylag szigorúak a szabályok (erősen típusos)

A programozási nyelvek egy részénél a fordítóprogram már a fordítási időben minden egyes részkifejezésről el tudja dönteni, hogy az milyen típusú. Ezeket a nyelveket statikus típusrendszerrel rendelkezőnek nevezzük. Ennek vannak előnyei, hiszen a nyelv alaposabb ellenőrzéseket tud végrehajtani és optimálisabb kódot is tud generálni. Ilyen nyelv a Fortran, Algol, C, Pascal, C++, Java, C#, Go.

Más nyelveknél, legtöbbször az interpretált nyelveknél, egy változó idővel más típusú értékekre is hivatkozhat. Ilyenkor a fordító futási időben kezeli a típusinformációkat. Ezt dinamikus típusrendszer-nek nevezzük. Ilyen nyelv pl. a Python.

Mindez nem jelenti, hogy a dinamikus típusrendszer nem ellenőrizheti a típusok alkalmazását, sőt helytelen alkalmazás hibát okozhat. Azokat a nyelveket, ahol ilyen hibák előfordulnak erősen típusos-nak nevezzük, szemben a gyengén típusos nyelvekkel.

A C erősen típusos statikus típusrendszerrel rendelkező nyelv.

Statikus és dinamikus szemantikai szabályok

- Statikus: amit a fordító ellenőriz
- Dinamikus: amit futás közben lehet ellenőrizni
  - Pl. tömbindexelés

Eldönthetőségi probléma…

Összefoglalva

- Lexikális: mik az építőkockák?
- Szintaktikus: hogyan építünk struktúrákat?
- Szemantikus: értelmes az, amit felépítettünk?
  - statikus szemantikai szabályok
  - dinamikus szemantikai szabályok

6.3 Kitekintés későbbi tárgyakra

A fordítóprogramok részei

- Lexer: tokenek sorozata
- Parser: szintaxisfa, szimbólumtábla
- Szemantikus (pl. típus-) ellenőrzés

(vagy különböző szintű fordítási hibák)

Formális nyelvek

- Lexikális szabályok: reguláris nyelvtan
- Szintaktikus szabályok: környezetfüggetlen nyelvtan
- Szemantikus szabályok: környezetfüggő, vagy megkötés nélküli nyelvtan

Program szemantikája

A (nyelv szabályainak megfelelő) program jelentése

6.4 Pragmatika

A nyelv definíciója

- Lexika
- Szintaktika
- Szemantika
- Pragmatika

Pragmatika

Hogyan tudjuk hatékonyan kifejezni magunkat?

- Konvenciók
- Idiómák
- Jó és rossz gyakorlatok

stb.

Konvenció

általános vagy fejlesztői csoportra (cégre) specifikus

- kapcsos zárójelek elhelyezése
- névválasztás (pl. setter/getter)
- azonosítók írásmódja, nyelve
- kis- és nagybetűk

7 Kifejezések

Példák

```
n + 1                                   3.14 * r * r

                  3 * v[0]

      x < 3.14
                                  3 * (r1 + r2) == factorial(x)
```

Lexika

- literálok
- operátorok
- azonosítók
- zárójelek
- egyéb jelek, pl. vessző

7.1 Számábrázolás

Számok ábrázolása

- egész számok (integer) – egy intervallum $\mathbb{Z}$-ben
  - előjel nélküli (unsigned)
  - előjeles (signed)
- lebegőpontos számok (float) $\subsetneq \mathbb{Q}$

(különböző méretekben)

Előjel nélküli egész számok

Négy biten

$1011 = 2^3 + 2^1 + 2^0$

n biten

$b_{n-1} \ldots b_2\ b_1\ b_0 = \sum_{i=0}^{n-1} b_i\ 2^i$

C-ben

```c
unsigned int big = 0xFFFFFFFF;
if( big > 0 ){ printf("it's big!"); }
```

Előjellel: „kettes komplemens” ábrázolás

(two's complement)

- első bit: előjel
- többi bit: helyiértékek

Négy biten

```
0000     0
0001     1          1111    -1
0010     2          1110    -2
0011     3          1101    -3                 0011
0100     4          1100    -4                +1101
0101     5          1011    -5                -----
0110     6          1010    -6                10000
0111     7          1001    -7
                    1000    -8
```

Előjeles egész C-ben

```
int big = 0xFFFFFFFF;
if( big > 0 ){ printf("it's big!"); }
```

Aritmetika előjeles egészekre

- Aszimmetria: eggyel több negatív érték
- Természetellenes
  - „két nagy pozitív szám összege negatív lehet”
  - „negatív szám negáltja negatív lehet”
- Példa: két szám számtani közepe?

Egész típusok mérete

- `short`: legalább 16 bit
- `int`: legalább 16 bit
- `long`: legalább 32 bit
- `long long`: legalább 64 bit (C99)

```
sizeof(short) <= sizeof(int) <= sizeof(long)
```

Haskell

- `Int`
- `Integer` – „tetszőlegesen nagy” abszolút értékű szám

Lebegőpontos számok

$1423.3 = 1.4233 \cdot 10^3$

$14.233 = 1.4233 \cdot 10^1$

$0.14233 = 1.4233 \cdot 10^{-1}$

Bináris ábrázolás

$(-1)^s \cdot m \cdot 2^e$

(s: előjel; m: mantissza; e: exponens)

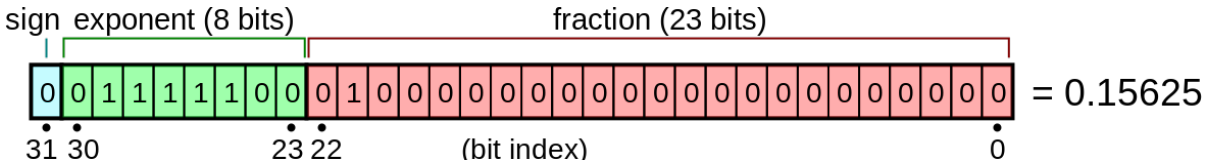
Rögzített számú biten reprezentálandó

- Előjel
- Kitevő
- Értékes számjegyek

IEEE 754

- Bináris rendszer
- A legtöbb számítógépes rendszerben
- Különböző méretű számok
  - egyszeres (32 bites: 1 + 23 + 8)
  - dupla (64 bites: 1 + 52 + 11)
  - kiterjesztett (80 bites: 1 + 64 + 15)
  - négyszeres (128 bites: 1 + 112 + 15)
- Mantissza 1 és 2 közé esik (pl. `1.01101000000000000000000`)
  - implicit első bit

32 bites példa



- előjel: 0 (nem negatív szám)
- „karakterisztika”: `01111100`, azaz 124
  - kitevő = karakterisztika - 127 = -3
- mantissza: 1.01000…0, azaz 1.25

Jelentés: $(-1)^0 \cdot 1.25 \cdot 2^{-3}$, azaz 1.25/8

Lebegőpontos számok tulajdonságai

- Széles értéktartomány
  - Nagyon nagy és nagyon kicsi számok
- Nem egyenletes eloszlású
- Alul- és túlcsordulás
  - Pozitív és negatív nullák
  - Végtelenek
  - NaN
  - Denormalizált számok

Lebegőpontos aritmetika

```
2.0 == 1.1 + 0.9
2.0 - 1.1 != 0.9
2.0 - 0.9 == 1.1
```

Pénzt például sosem ábrázolunk lebegőpontos számokkal!

Komplex számok

Valós és képzetes részből, pl.: $3.14 + 2.72i$ (ahol $i^2 = -1$)

C99

```
float _Complex fc;
double _Complex dc;
long double _Complex ldc;
```

Komplex számok C99-től

```
#include <complex.h>
...
double complex dc = 3.14 + 2*I;
```

Konvertálás típusok között

```
double pi = 3.141592;
int three = (int) pi;
```

7.2 Operátorok

Operátorok

- aritmetikai
- értékadó
- eggyel növelő/csökkentő
- relációs
- logikai
- feltételes
- bitművelet
- `sizeof`

Aritmetikai operátorok

```
+ operand
- operand
left + right
left - right
left * right
left / right
left % right
```

„Valós” és egész osztás

```
5.0 / 2.0 == 2.5
5 / 2 == 2
```

Osztási maradék

```
(left / right) * right + (left % right) == left
```

Eredmény előjele: `left` előjele

Hatványozás

```
#include <math.h>

    pow( 5.1, 2.1 )
```

Értékadás

Értékadó utasítás

```
n = 1;
```

Mellékhatásos kifejezés

```
n = 1
```

Mellékhatásos kifejezés értéke

```
(n = 1) == 1
```

Érték továbbgyűrűzése

```
m = (n = 1)
```

Értékadó operátorok

```
n = 3
n += 3          n = (n + 3)
n -= 3          n = (n - 3)
n *= 3          n = (n * 3)
n /= 3          n = (n / 3)
n %= 3          n = (n % 3)
```

Eggyel növelő/csökkentő operátorok

Mellékhatás

```
c++;          c += 1;       c = (c + 1);
++c;          c += 1;       c = (c + 1);

c--;          c -= 1;       c = (c - 1);
--c;          c -= 1;       c = (c - 1);
```

Érték

```
c++            c
++c            c+1

c--            c
--c            c-1
```

Relációs operátorok

```
left == right
left != right
left <= right
left >= right
left < right
left > right
```

Logikai (boolean) értékűek

Mit jelent ez?

```
3 < x < 7
```

Logikai típus?

ANSI C: nincsen

hamis: `0`, igaz: minden más (de főleg `1`)

```
int right = 3 < 5;
int wrong = 3 > 5;
printf("%d %d\n",right,wrong);
```

C99-től

```
                              #include <stdbool.h>
                              ...
_Bool v = 3 < 5;              bool v = true;
int one = (_Bool) 0.5;
int zero = (int) 0.5;
```

Végtelen ciklus idiómája

```
while(1){
    ...
}
```

Mit csinál ez a kód?

```
while( x = 5 ){
    printf("%d\n", x);
    --x;
}
```

Logikai operátorok

```
left && right
left || right
! operand
```

Feltételes operátor

```
condition ? left : right
```

Bitműveletek

```
int two = 2;
int sixteen = 2 << 3;
int one = 2 >> 1;
int zero = 2 >> 2;

int three = two | one;
int five = 13 & 7;
int twelve = 9 ^ five;
int minusOne = ~zero;
```

7.3 Szintaktika

Függvényhívás szintaktikája

aktuális paraméterlista

```
<function-call> ::= <identifier> ( )
                  | <identifier> (<argument-list>)

<argument-list> ::= <expression>
                  | <expression> , <argument-list>
```

`pi()`, `factorial(n+m)`, `min(0,x*y)`

Operátorok használata

- arítás
  - unáris, pl. `-x`, `c++`
  - bináris, pl. `x-y`
  - ternáris, pl. `x < 0 ? 0 : x`
- fixitás
  - prefix, pl. `++c`
  - postfix, pl. `c++`
  - infix, pl. `x+y`
  - mixfix, pl. `x < 0 ? 0 : x`

7.4 Szemantika

Kifejezések kiértékelése

- teljesen bezárójelezett kifejezés

  `3 + ((12 - 3) * 4)`

- precedencia: a `*` erősebben köt, mint a `+`

  `12 - 3 * 4`

- bal- és jobbasszociativitás
  - azonos precedenciaszintű operátorok esetén
  - `3 * n / 2` jelentése `(3 * n) / 2` (bal-asszociatív op.)
  - `n = m = 1` jelentése `n = (m = 1)` (jobb-asszociatív op.)

Kifejezések kiértékelése (folyt.)

- lustaság, mohóság
  - mohó: az `A + B` alakú kifejezés
  - lusta: az `A && B` alakú kifejezés
- mellékhatás

```
n = 1
i++
++i
i *= j
```

- operandusok, függvényparaméterek kiértékelési sorrendje

```
int i = 2;
int j = i -- - -- i;
```

Szekvenciapont

- Teljes kifejezés végén
- Függvényhívás aktuális paraméterlistájának kiértékelése végén
- Lusta operátorok első operandusának kiértékelése után
- Vessző operátornál

Vessző-operátor

```
<expression> ::= ...
               | <expression> ,  <expression>
```

- Az eredménye a jobboldali kifejezés eredménye
- Alacsony precedenciaszintű

```
int i = 1, v;
v = (++i, i++);   /* nem ugyanaz, mint: v = ++i, i++; */
```

Vessző: operátor vagy elválasztójel

```
int i = 1, v;
if( v = f(i,i), v > i )
    v = f(v,v), i += v;

for( i = f(v,v), v = f(i,i); i < v; ++i, --v ){
    printf("%d %d\n",i,v);
}
```

Értékek

- szám
  - egész (`144L`, `-23`, `0xFFFF`)
  - valós (`123.4`, `314.1592E-2`)
  - komplex (`3.14j`)
- karakter (`'a'`, `'\n'`)
- sorozat
  - szöveg
  - számsorozat

7.5 Szövegek

Karakterek

Valójában egy egész szám!

- Egy bájtos karakterkód, pl. ASCII

```
char c = 'A';       /* ASCII: 65 */
```

Escape-szekvenciák

- Speciális karakter: `\n`, `\r`, `\f`, `\t`, `\v`, `\b`, `\a`, `\\`, `\`, `\"`, `\?`
- Oktális kód: `\0` – `\377`
- Hexadecimális kód, pl. `\x41`

Előjeles és előjel nélküli `char`

```
  signed char a = '\xFF';    /* a < 0          */
unsigned char b = '\xFF';    /* b > 0          */
         char c = '\xFF';    /* platformfüggő */
```

Szélesebb ábrázolás

```
wchar_t w = L'é';
```

- Implementációfüggő!
  - Windows: UTF-16
  - Unix: általában UTF-32
- C99-től „univerzális kód”:, pl. `\uC0A1` és `\U00ABCDEF`

Szövegek

- Nem `string`!
- Karakterek tömbje, `'\0'`-val terminálva
  - Nullától indexelünk

```
char word[] = "apple";
printf("%lu\n", sizeof(word));   /* 6 */

char a = word[0];
word[0] = 'A';

wchar_t wide[] = L"körte";
```

Ékezetes betűk a szövegben

- Platformfüggő ábrázolás
- Egy karaktert több bájton is ábrázolhat
  - pl. UTF-8

```
char word[] = "körte";
printf("%lu\n", sizeof(word));   /* 7 */
```

Karaktertömb lefoglalása

```
char w1[] = "alma";
char w2[8] = "alma";
printf("%lu %s\n", sizeof(w1), w1);   /* 5 alma */
printf("%lu %s\n", sizeof(w2), w2);   /* 8 alma */
```

Veszély: túl kis tömb foglalása

```
char w1[] = "lakoma";
char w2[4] = "alma";
printf("%lu %s\n", sizeof(w1), w1);   /* 7 lakoma */
printf("%lu %s\n", sizeof(w2), w2);   /* 4 almalakoma */
```

Szövegen belül nulla

```
char word[] = "lak\0ma";
printf("%lu %s\n", sizeof(word), word);   /* 7 lak */
printf("%c\n", word[4]);                  /* m */
```

Szövegek manipulálása

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

int main()
{
    char word[100];
    strcpy(word,"alma");
    strcat(word,"lakoma");
    printf("%lu %s\n", sizeof(word), word);  /* 100 almalakoma */
    printf("%lu\n", strlen(word));           /* 10 */
    return 0;
}
```

Kitekintés

```
char w1[]  = "alma";   /* a szöveg benne lesz */
char w2[6] = "alma";   /* a szöveg benne lesz */
char * w3  = "alma";   /* szövegre mutat, nem kellene módosítani */
printf("%lu %s\n", sizeof(w1), w1);   /* 5 alma */
printf("%lu %s\n", sizeof(w2), w2);   /* 6 alma */
printf("%lu %s\n", sizeof(w3), w3);   /* 8 alma */

w1[0] = 'A';
w2[0] = 'A';
w3[0] = 'A';     /* problémás - Segmentation Fault? */
```

7.6 Sorozatok

C tömb

```
double point[3];     /* a méret legyen konstans */
point[0] = 3.14;     /* nullától indexelünk */
```

```
point[1] = 2.72;
point[2] = 1.0;
```

Tömb inicializációja

```
double point[] = {3.14, 2.72, 1. + .1};
             /* az elemek legyenek "konstansok", ha globális */


point[2] = 1.0;   /* módosítható */
```

Feldolgozás

```
#define DIMENSION 3

double sum( double point[] ){
    double result = 0.0;
    int i;
    for( i=0; i<DIMENSION; ++i ){
        result += point[i];
    }
    return result;
}

int main(){
    double point[DIMENSION] = {3.14, 2.72, 1.0};
    printf("%f\n", sum(point));
    return 0;
}
```

Általánosítás

```
double sum( double nums[], int length )
{
    double result = 0.0;
    int i;
    for( i=0; i<length; ++i ){
        result += nums[i];
    }
    return result;
}

int main()
{
    double point[] = {3.14, 2.72, 1.0};
    printf("%f\n", sum(point,3));
    return 0;
}
```

Veszélyforrások

Fordítási hiba

```
double point[DIMENSION] = {3.14, 2.72, 1.0, 2.0};
```

Inicializálatlan elemek

```
double point[DIMENSION] = {3.14, 2.72};
```

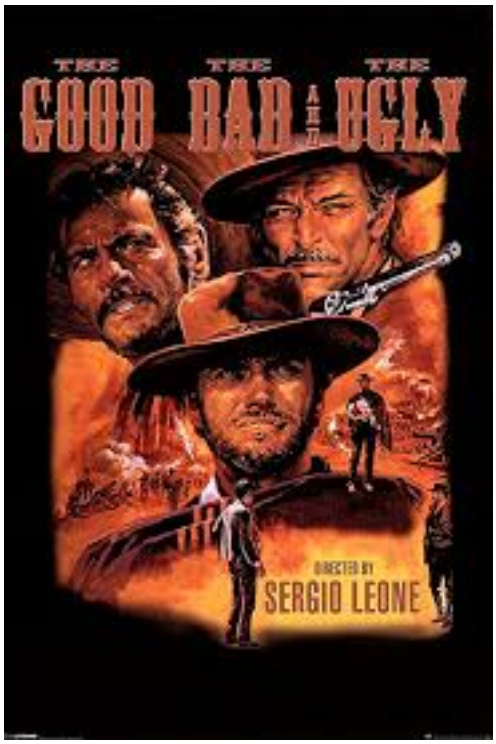
Túlindexelés, illegális memóriaolvasás

```
printf("%f\n", point[1024]);
```

Túlindexelés, illegális memóriaírás (buffer overflow)

```
point[31024] = 1.0;   /* Segmentation fault? */
```

Szövegek megadása tömbként




```
char good[] = "good";
char  bad[] = {'b', 'a', 'd'};
char ugly[] = {'u','g','l','y','\0'};
printf("%s %s %s\n", good, bad, ugly);
```

8 Utasítások

Eddig megismert utasítások

- Egyszerű utasítások
  - Változódeklaráció
  - Értékadás
  - Alprogramhívás
  - Visszatérés függvényből
- Vezérlési szerkezetek
  - Elágazás
  - Ciklus

8.1 Egyszerű utasítások

Változódeklaráció

- Minden változót az első használat előtt létrehozunk
- Érdemes már itt inicializálni

```
double m;
int n = 3;
char cr = '\r', lf = '\n';
int i = 1, j;
int u, v = 3;
```

Kifejezés-utasítás

(Mellékhatásos) kifejezés kiértékelése

```
<statement> ::= <expression> ;
              | ...
```

```
n = 1;
x *= y;
c++;
n > 0 ? --n : ++n;   /* nem idiomatikus! */
```

Tipikus példa: értékadások

Függvények

- Deklarált visszatérési típus, megfelelő `return` utasítás(ok)
- Csak mellékhatás: `void` visszatérési érték, üres `return`

Tiszta függvény

```
unsigned long fact(int n)
{
  unsigned long result = 1L;
  int i;
  for( i=2; i<=n; ++i )
     result *= i;
  return result;
}
```

Csak mellékhatás

```
void print_squares(int n)
{
  int i;
  for( i=1; i<=n; ++i ){
     printf("%d\n",i*i);
  }
  return; /* elhagyható */
}
```

Kevert viselkedés

```
printf("%d\n", printf("%d\n",42));
```

Visszatérés

- Egy függvényben akár több `return` utasítás is lehet
- Nincs `return` ≡ üres `return` (`void`)

```
return 42;
return v + 3.14;

return;
```

Több `return` utasítás

```
int index_of_1st_negative( int nums[], int length ){
    int i;
    for( i=0; i<length; ++i )
        if( nums[i] < 0 )
            return i;
    return -1;    /* extremális érték */
}
```

Üres utasítás

```
;
```

```
int i = 0;
while( i<10 );
    printf("%d\n",++i);
```

```
int i, nums[] = {3,6,1,45,-1,4};
for( i=0; i<6 && nums[i]<0; ++i);

for( i=0; i<6 && nums[i]<0; ++i){
}
```

8.2 Vezérlési szerkezetek

Vezérlési szerkezetek

- Elágazás
- Ciklus
  - Tesztelő
    - Elöltesztelő
    - Hátultesztelő
  - Léptető
- Nem strukturált vezérlésátadás
  - `return`
  - `break`
  - `continue`
  - `goto`

Strukturált programozás

- Szekvencia, elágázás, ciklus
- Minden algoritmus leírható ezekkel
- Olvashatóbb, könnyebb érvelni a helyességéről
- Csak nagyon alapos indokkal térjünk el tőle!

Szekvencia

- Utasítások egymás után írásával
- Pontosvessző
- Blokk utasítás

```
<statement>      ::= { <statement-list> }
                   | ...

<statement-list> ::= ""
                   | <statement> <statement-list>
```

Vezérlési szerkezetek belseje

- egy utasítás
- lehet a blokk-utasítás is

Elágazás

- if-else szerkezet
  - az else-ág opcionális
- csellengő else

Többágú elágazás

```
if( x > 0 )
    y = x;
else if( y > 0 )
    x = y;
else
    x = y = x * y;
```

Többágú elágazás konvencionális tördelése

Idióma

```
if( x > 0 )
    y = x;
else if( y > 0 )
    x = y;
else
    x = y = x * y;
```

Konvencionális tördelés

```
if( x > 0 )
    y = x;
else
    if( y > 0 )
        x = y;
    else
        x = y = x * y;
```

A kapcsos zárójelek nem ártanak

Idióma

```
if( x > 0 ){
    y = x;
} else if( y > 0 ){
    x = y;
} else {
    x = y = x * y;
}
```

Konvencionális tördelés

```
if( x > 0 ){
    y = x;
} else {
    if( y > 0 ){
        x = y;
    } else {
        x = y = x * y;
    }
}
```

switch-case-break utasítás

egész típusú, fordítási idejű konstansok alapján

```
switch( dayOf(date()) )
{
    case  0: strcpy(name,   "Sunday"); break;
    case  1: strcpy(name,   "Monday"); break;
    case  2: strcpy(name,  "Tuesday"); break;
    case  3: strcpy(name,"Wednesday"); break;
    case  4: strcpy(name, "Thursday"); break;
    case  5: strcpy(name,   "Friday"); break;
    case  6: strcpy(name, "Saturday"); break;
    default: strcpy(name,"illegal value");
}
```

Adatban kódolt vezérlés

```
char *names[] = {"Sunday", "Monday", "Tuesday", "Wednesday",
                 "Thursday", "Friday", "Saturday"};
strcpy(name, names[dayOf(date())]);
```

Nem mindig kényelmes adatként

```
switch( key )
{
    case 'i': insertMode(currentRow,currentCol);
              break;
    case 'I': insertMode(currentRow,0);
              break;
    case 'a': insertMode(currentRow,currentCol+1);
              break;
    case 'A': insertMode(currentRow,length(currentRow));
```

```
                break;
    case 'o': openNewLine(currentRow+1);
                break;
    case 'O': openNewLine(currentRow);
                break;
}
```

Átcsorgás

```
switch( month )
{
    case  1:
    case  3:
    case  5:
    case  7:
    case  8:
    case 10:
    case 12: days = 31;
             break;
    case  2: days = 28 + (isLeapYear(year) ? 1 : 0);
             break;
    default: days = 30;
}
```

Nem triviális átcsorgás

```
switch( getKey() )
{
    case 'q': jump = 1;
    case 'a': moveLeft();
              break;
    case 'e': jump = 1;
    case 's': moveRight();
              break;
    case ' ': openDoor();
}
```

A `switch` és a strukturált programozás

Strukturáltnak tekinthető

- Minden ág végén `break`
- Ugyanaz az utasítássorozat többághoz

Nem felel meg a strukturált programozásnak

- Nem triviális átcsorgások
- Pl. ha egyáltalán nincs `break`

Elöltesztelő ciklus

```
while( i > 0 )
{
    printf("%i\n", i);
    --i;
}
```

Olvashatóság

```
while( i > 0 )
{
    printf("%i\n", i);
    --i;
}
```

```
while( i > 0 )
    printf("%i\n", i--);
```

while – szintaxis

```
<while-stmt> ::= while ( <expression> ) <statement>
```

Hátultesztelő ciklus

```
<do-while-stmt> ::= do <statement> while ( <expression> );
```

Jellemző példa

```
char command[LENGTH];
do {
   read_data(command);
   if( strcmp(command,"START") == 0 ){
      printf("start\n");
   } else if( strcmp(command,"STOP") == 0 ){
      printf("stop\n");
   }
} while( strcmp(command,"QUIT") != 0 );
```

Átírás – 1

Milyen feltétel mellett igaz ez?

```
do σ while ( ε );    ≡    σ while ( ε ) σ
```

Átírás – 2

Milyen feltétel mellett igaz ez?

```
do σ while ( ε );
  ≡
int new_var = 1; ... while ( new_var ){ σ  new_var = ε; }
```

Az előző példa átírva

```
char command[LENGTH];
int new_var = 1;
...
while( new_var ) {
   read_data(command);
   if( strcmp(command,"START") == 0 ){
       ...
   } else if( strcmp(command,"STOP") == 0 ){
       ...
```

```
   }
   new_var = ( strcmp(command,"QUIT") != 0 );
}
```

Refaktorálva

```
char command[LENGTH];
int stay_in_loop = 1;
...
while( stay_in_loop ) {
   read_data(command);
   if(        strcmp(command,"START") == 0 ){
       ...
   } else if( strcmp(command,"STOP" ) == 0 ){
       ...
   } else if( strcmp(command,"QUIT" ) == 0 ){
       stay_in_loop = 0;
   }
}
```

Vége jelig való beolvasás idiómája

```
void cat(void)
{
    int c;
    while( (c = getchar()) != EOF )
    {
        putchar(c);
    }
}
```

Léptető ciklus

```
<for-stmt> ::= for ( <optional-expression> ;
                     <optional-expression> ;
                     <optional-expression> )
                   <statement>
<optional-expression> ::= "" | <expression>
```

(inicializáció; feltétel; léptetés)

Végtelen ciklus

```
while(1) ...

for(;;) ...
```

Karaktertábla készítése

```
unsigned char c;
for( c = 0; c <= 255; ++c )
{
    printf( "%d\t%c\n", c, c );
}
```

Célszerű így fordítani

```
gcc -ansi -W -Wall -pedantic ...
```

Átírások

Mindig megtehető

`while ( ε ) σ` ⇒ `for ( ; ε ; ) σ`

Milyen feltétel mellett igaz ez?

`for ( ι ; ε ; λ ) σ` ⇒ `ι; while ( ε ){ σ λ; }`

Egyszerű utasítások

- Változódeklarációs utasítás
- Üres utasítás
- Kifejezés-utasítás
- Értékadás
- Alprogramhívás
- Visszatérés alprogramból (`return`)

Strukturált programozás vezérlési szerkezetei

- Blokk utasítás
- Elágazások
  - `if`–`elif`–`else`
  - `switch`–`case`–`break`
- Ciklusok
  - Tesztelő ciklusok
    - Elöltesztelő (`while`)
    - Hátultesztelő (`do`–`while`)
  - Léptető ciklus (`for`)

8.3 Nem strukturált vezérlésátadás

Nem strukturált vezérlésátadás

- `return`
- `break` és `continue`
- `goto`

break utasítás

- Kilép a legbelső ciklusból (vagy `switch`-ből)

```
while( !destination(x,y) ){
    drawPosition(x,y);
    dx = read(sensorX);
    if( dx == 0 ){
        dy = read(sensorY);
        if( dy == 0 ) break;
    } else dy = 0;
    x += dx;
```

```
        y += dy;
    }
```

continue utasítás

- Befejezi a legbelső ciklusmag végrehajtását

```
while( !destination(x,y) ){
    drawPosition(x,y);
    dx = read(sensorX);
    if( dx == 0 ){
        dy = read(sensorY);
        if( dy == 0 ) continue;
    } else dy = 0;
    if( validPosition( x+dx, y+dy ){
        x += dx;
        y += dy;
    }
}
```

- for-ciklusnál végrehajtja a léptetést

goto utasítás

- Egy függvényen belül a megadott címkéjű utasításra ugrik

```
<statement> ::= ...
              | goto <label>
              | <label> : <statement>
<label> ::= <identifier>
```

Keressünk nulla elemet egy mátrixban

goto-val

```
int matrix[SIZE][SIZE];
...
int found = 0;
int i, j;
for( i=0; i<SIZE; ++i ){
  for( j=0; j<SIZE; ++j ){
    if( matrix[i][j] == 0 ){
      found = 1;
      goto end_of_search;
    }
  }
}
/* --i; --j; */
end_of_search:;
```

szabályosan

```
int matrix[SIZE][SIZE];
...
  int found = 0;
  int i=-1, j;
  while( i<SIZE-1 && !found ){
    j = -1;
```

```
    while( j<SIZE-1 && !found ){
      if( matrix[i+1][j+1] == 0 ){
        found = 1;
      }
      j++;
    }
    i++;
  }
```

8.4 Rekurzió

Rekurzív alprogramok

```
int factorial( int n ){
   if( n < 2 ){
      return 1;
   } else {
      return n * factorial(n-1);
   }
}
```

Másképpen fogalmazva

```
int factorial( int n )
{
   return n < 2 ? 1 : n * factorial(n-1);
}
```

Számítási lépések ismétlése

Imperatív programozás

- Iteráció (ciklus)
- Hatékony

```
int factorial( int n ){
   int result = 1;
   int i = 1;
   for( i=2; i<=n; ++i )
      result *= i
   return result;
}
```

Funkcionális programozás

- Rekurzió
- Érthető

```
int factorial( int n ){
   if( n < 2 ){
      return 1;
   } else {
      return n * factorial(n-1);
   }
}
```

Rekurzió imperatív nyelvben

- A legtöbb nyelvben támogatott
- Ritkán használják a gyakorlatban
  - Hatékonyság
  - Stack overflow

Van, amikor kényelmes

```
int partition( int array[], int lo, int hi );

void quicksort_rec( int array[], int lo, int hi )
{
    if( lo < hi )
    {
        int pivot_pos = partition(array,lo,hi);
        quicksort_rec( array, lo, pivot_pos-1 );
        quicksort_rec( array, pivot_pos+1, hi );
    }
}

void quicksort( int array[], int length )
{
    quicksort_rec(array,0,length-1);
}
```

```
int partition( int array[], int lo, int hi )
{
   int pivot = array[lo], tmp;
   int i = lo+1, j = hi;

   while( 1 )
   {
        while( i <= hi && array[i] <= pivot ) ++i;
        while( array[j] > pivot ) --j;
        if( i >= j ) break;
        tmp = array[i]; array[i] = array[j]; array[j] = tmp;
   }
   array[lo] = array[j]; array[j] = pivot;
   return j;
}
```

Végrekurzív függvény (tail-recursion)

- Vannak eleve végrekurzív módon megadottak
- De mesterségesen is átírhatók (accumulator)

Kézenfekvő

```
int factorial( int n ){
   return n < 2 ? 1 : n * factorial(n-1);
}
```

Végrekurzív

```
int fact_acc(int n, int acc){
   return n < 2 ? acc : fact_acc(n-1,n*acc);
```

```
}

int fact( int n ){
   return fact_acc(n,1);
}
```

A fordítóprogram optimalizálhatja

Végrekurzív

```
int fact_acc(int n, int acc){
   if (n<2) return acc;
   else return fact_acc(n-1,n*acc);
}
```

Optimalizált

```
int fact_acc(int n, int acc){
   START: if (n<2) return acc;
   else {
      acc *= n;
      n--;
      goto START;
   }
}
```

Strukturáltan

```
int fact_acc(int n, int acc){
   while( n>=2 ){

      acc *= n;
      n--;
   }
   return acc;
}
```

A végrekurzív módon megadott alprogramból a fordítóprogram elég könnyen tud olyan kódot fordítani, amiből eltűnik a rekurzív hívás: a formális paramétereknek értékül adjuk a rekurzív hívásban szereplő aktuális paramétereket, majd visszaugrunk az alprogram elejére.

Az így optimalizált kód már megfelel annak, amit `while`-ciklussal mi magunk írhatnánk szépen, strukturáltan. Egy újabb optimalizáció a két függvénydefiníciót egybevonná, és eljutnánk ahhoz a kódhoz, ami az alábbi kézzel írt definíciónak felene meg.

```
int factorial(int n){
   int acc = 1;
   while( n>=2 ){

      acc *= n;
      n--;
   }
   return acc;
}
```

9 Hatókör

Programszerkezet

- Program tagolása – logikai/fizikai
- Programegységek (program units)
  - Pl. alprogramok (függvények)

Mellérendelt szerkezetek

- Fordítási egységek
- Programkönyvtárak
- Újrafelhasználhatóság

Alá-/fölérendelt szerkezetek

- Egymásba ágyazódás
- Hierarchikus elrendezés
- Lokalitás: komplexitás csökkentése

Hierarchikus programfelépítés

- Programegységek egymásba ágyazása
- Ha függvényben függvény: blokkszerkezetes (block structured) nyelv
- Hatókör szűkítése: csak ott használható, ahol használni akarom

Hierarchia nélkül

```
int partition( int array[], int lo, int hi ){ ... }

void quicksort_rec( int array[], int lo, int hi ){
    if( lo < hi ){
        int pivot_pos = partition(array,lo,hi);
        quicksort_rec( array, lo, pivot_pos-1 );
        quicksort_rec( array, pivot_pos+1, hi );
    }
}

void quicksort( int array[], int length ){
    quicksort_rec(array,0,length-1);
}
```

Függvények egymásba ágyazása, lokális fv-definíció?

Nem valid C-kód!

```
void quicksort( int array[], int length )
{
    int partition( int array[], int lo, int hi ){ ... }

    void quicksort_rec( int array[], int lo, int hi ){
        if( lo < hi ){
            int pivot_pos = partition(array,lo,hi);
            quicksort_rec( array, lo, pivot_pos-1 );
```

```
            quicksort_rec( array, pivot_pos+1, hi );
        }
    }

    quicksort_rec(array,0,length-1);
}
```

Függvények egymásba ágyazása tetszőleges mélységben?

Nem valid C-kód!

```
void quicksort( int array[], int length )
{
    void quicksort_rec( int array[], int lo, int hi )
    {
        int partition( int array[], int lo, int hi ){ ... }

        if( lo < hi ){
            int pivot_pos = partition(array,lo,hi);
            quicksort_rec( array, lo, pivot_pos-1 );
            quicksort_rec( array, pivot_pos+1, hi );
        }
    }

    quicksort_rec(array,0,length-1);
}
```

Deklaráció és definíció

Gyakran együtt, de lehet az egyik a másik nélkül!

- Deklaráció: nevet adunk valaminek
  - változódeklaráció
  - függvénydeklaráció
- Definíció: meghatározzuk, mi az
  - a változó létrehozása (tárhely foglalása)
  - függvénytörzs megadása

```
unsigned long int factorial(int n);
int main(){ printf("%ld\n",factorial(20)); return 0; }
unsigned long int factorial(int n){
    return n < 2 ? 1 : n * factorial(n-1);
}
```

Deklaráció hatóköre (scope)

Amíg a névvel elérhető az, amire hivatkozik

- Globális: legkívül van
- Lokális: valamin belül van

```
unsigned long int factorial( int n )    /* globális függvény */
{
    unsigned long int result = 1L;      /*  lokális változó  */
    int i;
    for( i=2; i<n; ++i )
```

```
    {
        result *= i;
    }
    return result;
}
```

Globális és lokális függvény

Ha a C blokkszerkezetes lenne...

```
void quicksort( int array[], int length )  /* global */
{
    void quicksort_rec( int array[], int lo, int hi )  /* local */
    {
        int partition( int array[], int lo, int hi ){ ... }

        if( lo < hi ){
            int pivot_pos = partition(array,lo,hi);
            quicksort_rec( array, lo, pivot_pos-1 );
            quicksort_rec( array, pivot_pos+1, hi );
        }
    }

    quicksort_rec(array,0,length-1);
}
```

Blokk (block)

A (statikus) hatóköri szabályok alapja

- Alprogram
- Blokk utasítás

Törzs (body)

```
void quicksort_rec( int array[], int lo, int hi )
{
    if( lo < hi )
    {
      int pivot_pos = partition(array,lo,hi);
      quicksort_rec( array, lo, pivot_pos-1 );
      quicksort_rec( array, pivot_pos+1, hi );
    }
}
```

Statikus hatóköri szabályok (static/lexical scoping)

a deklarációtól a deklarációt közvetlenül tartalmazó blokk végéig

```
int factorial( int n )
{
    int result = n, i = result-1;  /* nem cserélhető fel */
    while( i > 1 )
    {
        result *= i;
        --i;
    }
```

```
    return result;
}
```

Globális – lokális deklaráció

- Globális: ha a deklarációt nem tartalmazza blokk
- Lokális: ha a deklaráció egy blokkon belül van
  - Lokális a közvetlenül tartalmazó blokkra nézve

Lokális, non-lokális, globális deklaráció

- Lokális egy blokkra nézve: abban a blokkban van
- Nonlokális egy blokkra nézve:
  - befoglaló (külső) blokkban van
  - de az aktuális blokk a deklaráció hatókörében van
- Globális: semmilyen blokkra nem lokális

Lokális, non-lokális, globális változó

```
int counter = 0;                                    /* globális */
int fun(void)
{
    int x = 10;                                /* lokális fun-ra */
    while( x > 0 )
    {
        int y = x/2;           /* y lokális a blokk utasításra */
        printf("%d\n", 2*y == x ? y : y+1);
        --x;                /* nonlokális változó hivatkozható */
        ++counter;    /* nonlokális (globális) v. hivatkozható */
    }
}
```

Globális deklarációk és definíciók

```
int x;

extern int y;

extern int f(int p);    /* elhagyható az extern */

int g(void)
{
    x = f(y);
}
```

Fordítás

Minden fordítási egységben minden használt név deklarált kell legyen

Szerkesztés

Az egész programban minden globális név pontosan egyszer legyen definiálva

Elfedés (shadowing/hiding)

- Ugyanaz a név több dologra deklarálva
- Átfedő (tartalmazó) hatókörrel

```
void hiding(void)
{
    int n = 0;
    {
        int n = 1;
        printf("%d",n);
    }
    printf("%d",n);
}
```

- A „belsőbb” deklaráció nyer
- Láthatósági kör: a hatókör része

Lokális változók deklarációja

ANSI C

- Blokk elején, egyéb utasítások előtt

C99-től

- Keverve a többi utasítással

```
int n = 0;
{
    printf("%d",n);
    int n = 1;
    printf("%d",n);
}
```

- For-ciklus lokális változójaként

```
for( int i=0; i<10; ++i ) printf("%d",i);
```

Nonlokális definíció elérése

```
int n = 0;
{
    printf("%d",n);
    int n = 1;
    printf("%d",n);
}
```

Egy blokkon belül hivatkozás történik egy non-lokális változóra, majd utána ugyanazzal a névvel egy lokális változót is bevezetünk. A lokális változó hatóköre a deklarációjától kezdődik, ezért a deklarációja előtti hivatkozás a C szabályai értelmében a non-lokális deklarációra utal.

Deklarációk sorrendje

Hibás!

```
void g(void){
    printf("%c",f());
}
char f(void){ return 'G'; }
```

Helyes

```
char f(void);    /* forward declaration */
void g(void){
    printf("%c",f());
}
char f(void){ return 'G'; }
```

A hibás C kód azt mutatja, hogy a `g` függvenyben az `f` hívása túl korai: előrevetett deklaráció nélkül ne hivatkozzunk a később definiálásra kerülő `f`-re. A fordító ad erre egy fordítási figyelmeztetést, és készít egy „elképzelt", ún. *implicit deklarációt* az `f` függvényhez. Ebben az implicit deklarációban az `f` visszatérési típusa `int` lesz. Az `f` definíciójánál kiderül, hogy ez az implicit deklaráció nem felel meg az `f` tényleges deklarációjának, így kapunk egy fordítási hibát is. Az egész fordítási figyelmeztetéses, fordítási hibás mizériát elkerülhetjük egy előrevetett deklarációval, illetve nyilván az `f` és a `g` definíciójának felcserélésével is.

Összegzés

- Deklaráció, definíció
- Blokk
- Hatókör
- Statikus hatóköri szabályok
- Lokális, non-lokális, globális deklarációk
- Elfedés

Programentitások (függvények, változók) meghatározását *definiálásnak* neveztük. Azt, amikor nevet adtunk egy programentitásnak, *deklarációnak* hívtuk. Megfigyeltük, hogy sokszor épp a deklaráció az, ami a névvel ellátott programentitást definiálja. Arra is láttunk példát, amikor egy deklaráció nem definiál, mert csak utal egy másik, definiáló deklarációra. Haskellben látunk arra is példát, hogy programentitásokat név (és deklaráció) nélkül definiálunk.

Egy újabb fontos fogalom volt a blokk: ez alatt alprogramokat (függvényeket) és blokk utasításokat értünk. A blokkok egymásba ágyazhatók. Ez a hierarchikus programszerkezet vezet el minket a hatókör fogalmához.

Egy deklaráció hatóköre a program azon szakasza, ahol a deklarációban szereplő név–programentitás összekapcsolódás fennáll. A deklarációban megadott névvel a szóban forgó programentitást csak a hatókörön belül érhetjük el, azon kívül nem.

Láttuk, hogy a C nyelvben *statikus hatóköri szabályok* (static scoping vagy lexical scoping) vannak. Egy deklaráció hatóköre a tartalmazó blokk végéig tart, beleértve a beágyazott blokkokat is.

Lokálisnak nevezzük a deklarációt a közvetlenül tartalmazó blokkban. Egy beágyazott blokkban *non-lokálisnak* nevezzük azt a deklarációt, amelyet a tartalmazó (vagy az azt tartalmazó stb.) blokkban adtunk meg, és amelynek a hatókörében vagyunk. A blokkokon kívüli (legkülső) deklarációkat pedig *globálisnak* nevezzük.

A lokális, non-lokális és globális elnevezéseket változók, függvények kapcsán is használjuk. Például egy adott ponton egy változót *non-lokális változónak* nevezünk, ha a deklarációja non-lokális.

A blokkok egymásba ágyazódása egyes nyelvekben (például a C-ben is) teret ad az *elfedés* jelenségének: ha egy deklaráció hatókörében egy beágyazott blokkban ugyanazzal a névvel egy újabb deklarációt helyezünk el, a belső deklaráció elfedi a külső deklarációt a belső deklaráció hatókörében.

A hatókör kezdetével kapcsolatos szabály, valamint a külön fordíthatóság szabályai miatt a C nyelvben fontos szerep hárul az *előrevetett deklarációkra* (forward declaration). Egy definíció nélküli deklaráció lehetővé teszi azt, hogy a deklarált dologra (a C esetében függvényre, változóra) hivatkozhassunk az utána következő kódrészben, míg a definíciót is tartalmazó/jelentő deklarációt később, vagy akár egy másik fordítási egységben is elhelyezhetjük. Az `extern` kulcsszóval jelezhetjük a C-ben, hogy egy deklaráció nem definiáló (pl. előrevetett) deklaráció. Mivel a függvények definíció nélküli deklarációja szintaktikusan jól megkülönböztethető a definiáló deklarációtól, függvények esetén az `extern` elhagyható.

A C fordítóprogram ellenőrzi, hogy egy programentitásra való hivatkozás legális-e, azaz a hivatkozásban használt név megfelelően deklarálva van-e. Ha a deklaráció hatókörén kívül használjuk a nevet, minimum egy *fordítási figyelmeztetést* kapunk. (A figyelmeztetéseket a C esetén illik komolyan venni. A legtöbb figyelmeztetés azt jelzi, hogy valóban rossz a kódunk, ezért a szokásos megközelítés a C-fejlesztőknél az, hogy igyekeznek az összes fordítási figyelmeztetéstől megszabadítani a kódjukat. Jelen esetben a C egy nem deklarált függvény használatakor „feltételez" egy deklarációt: a használatból kikövetkeztethető típusú paraméterekkel és `int` visszatérési értékkel. Ha a függvényünk nem `int`-et ad vissza, akkor fordítási hibát is kapunk, amikor a fordító elér a definiáló deklarációhoz, és detektálja az implicit deklarációtól való eltérést.)

A C-fordító ellenőrzi, hogy minden használt név deklarálva van-e, és hogy a deklarációnak megfelelően használjuk-e a nevet. A külön fordítás lehetősége miatt azonban arra is szükség van, hogy a más fordítási egységben definiált programentitásokra való hivatkozások érvényességéről is meggyőződjünk. Ezt a szerepet a szerkesztőprogram (linker) tölti be. A linker dolga, hogy ellenőrizze, hogy minden globálisan deklarált dolgot *pontosan egyszer* definiáltunk-e. Ha a definiáló deklarációk és a rájuk hivatkozó nem definiáló (`extern`) deklarációk nincsenek összhangban, *szerkeszési hibát* kapunk.

Ciklusváltozó

C99: ciklusra lokális változó

```
for( int i=0; i<10; ++i )
    printf("%d",i);
printf("%d",i); /* fordítási hiba */
```

C: 012345678910

```
int i;
for( i=0; i<10; ++i )
    printf("%d",i);
printf("%d",i);
```

C: végtelen ciklus

```
signed char i;
for( i=0; i<=127; ++i )
    printf("%c",i);
```

Korábban már szót ejtettünk arról, hogy a C99-től kezdve a C-ben lehet a `for`-ciklus ciklusváltozóját magában a ciklusban delarálni. Ez nagyon hasznos és elegáns megoldás: biztosítja, hogy a ciklusváltozó a ciklusra nézve lokális. A ciklus után a hivatkozás a ciklusváltozóra fordítási hibát ad.

Ha a ciklusváltozót a ciklus előtt deklaráljuk, akkor persze a közvetlenül tartalmazó blokk végéig tart a hatóköre, így a ciklus után is hivatkozható a változó. Ne feledjük el, hogy a ciklusmag utolsó lefutása után a `for`-ciklus „léptető"-része lefut, és a „feltétel" rész ezután kerül kiértékelésre. Ennek megfelelően a ciklusváltozó a fenti példában eggyel nagyobb lesz, mint a ciklusmag utolsó végrehajtása során.

Ugyanez a jelenség figyelhető meg a harmadik példában: a 127 érték elérése után a ciklusváltozót a ciklus megpróbálja eggyel növelni, de persze ez túlcsordulást eredményez, és ez az oka annak, hogy itt egy végtelen ciklus alakul ki.

Amire még érdemes odafigyelni:

```
int i;
for( int i=0; i<10; ++i ){
    printf("%d",i);
}
printf("%d",i);
```

Itt most két `i` változó is van, a belső elfedi a külsőt. A ciklus utáni hivatkozás az `i`-re az (inicializálatlan, esetleg memóriaszemetet tartalmazó) első változót jelenti, és nem a 10 értékűt, ami megszűnt a ciklusból való kilépéssel! A véletlenül duplán deklarált változó jól be tudja csapni az embert…

9.1 Témakörök haladóknak

Definíció deklaráció nélkül

```
double x = x + x
six = double 3
zoo = double 10.0
```

```
six = (\x -> x+x) 3
```

```
double = \x -> x+x
six = double 3
```

Az első kódblokkban egy nevesített függvény definícióját és hívását látjuk. Megfigyelhetjük, hogy a függvény polimorf: működik szám típusokra is és sztringekre is. Ennek az az oka, hogy a `+` művelet is polimorf. (A `+` művelet ilyetén polimorfizmusát szoktuk túlterhelésnek, overloadingnak is mondani. Ez az operátor túl van terhelve, azaz több típusra is értelmezve van.)

A második kódblokk a névtelen függvények használatát szemlélteti. A lambda-kifejezések segítségével létrehozhatunk (definiálhatunk) úgy függvényeket, hogy nem adunk nekik nevet (azaz itt nem történik deklaráció). Az így létrehozott függvényeket meghívhatjuk, átadhatjuk paraméterként (egy magasabbrendű függvénynek), értékül adhatjuk egy változónak (ezt szemlélteti a harmadik kódblokk), vagy bármilyen más módon is felhasználhatjuk egy összetett kifejezésben.

A harmadik kódblokkban olyan változót látunk, mely függvény típusú. A változónak értékül adunk egy függvényt, majd a változóval, mint névvel meg is hívjuk azt.

Magasabbrendű függvények

Funkcionális programozási paradigma

```
filter predicate (x:xs)
  | predicate x
    = x : filter predicate xs
  | otherwise
    = filter predicate xs
filter _ [] = []

filter even [1..10]
filter (\x -> x > 4) [1..10]
filter ( > 4 ) [1..10]
```

A Haskell nyelvben definiálhatunk magasabb rendű függvényeket, olyanokat, mint a példában a `filter`, amely egy függvényt vár első paraméterében.

Mutatunk három lehetséges hívási formát is a `filter`-hez. Az első esetben az `even` függvényt adjuk át paraméterként, a második esetben pedig egy lambda-kifejezést. A harmadik esetben egy részlegesen alkalmazott függvény alkotja azt a kifejezést, amit átadunk a `filter`-nek.

Dinamikus hatóköri szabályok

Bash

```
#!/bin/bash
x=1
function g()
{
   echo $x;
   x=2;
}
function f() {
   local x=3;
   g;
}

f
echo $x
```

C

```
#include <stdio.h>
int x = 1;
void g(void)
{
    printf("%d\n",x);
    x = 2;
}
void f(void){
    int x = 3;
    g();
}
void main(){
    f();
    printf("%d\n",x);
}
```

Eddig azt mondtuk, hogy a C nyelvben statikus hatóköri szabályok vannak. Kitalálható, hogy ezek szerint vannak olyan nyelvek is, amelyekben *dinamikus hatóköri szabályokat* (dynamic scoping) találunk. Például a Perl nyelvben mind a kettő szerephez jut. De mi is ez a dinamikus hatóköri szabályrendszer?

Tekintsük a fenti bash shell-scriptet (forrás: https://en.wikipedia.org/wiki/Scope_(computer_science)). Ez másként fog működni, mint a neki szintaktikusan megfelelő C program.

A bash shell-script először 1-re inicializálja a globális `x` változót. Ezután definiálja az `f` és `g` függvényeket, majd meghívja az `f`-et. Az `f` létrehoz egy lokális `x` változót, 3 értékkel, ezután meghívja a `g` függvényt. A `g` az `echo` paranccsal kiírja az `x` változó értékét. Na de melyik `x` változóról van itt szó? Ha statikus hatóköri szabályok lennének, akkor a globális `x` változó értékét írná ki. A shell-script viszont a futás közben (dinamikusan) legutóbb definiált `x`-et, azaz az `f` függvényben 3 értékkel létrehozott lokális változót írja ki.

Ezután az `x` új értéket kap a `g`-ben. Ez még mindig az `f` lokális változója. Visszatér a vezérlés az `f`-hez, majd a főprogramhoz. A végső `echo` utasítás a globális `x` változót fogja kiírni, amelynek az értéke még mindig 1.

10 Dinamikus programszerkezet

Dinamikus programszerkezet

Hogyan működik a program?

- Információk a programvégrehajtás állapotáról
  - A főprogramból induló alprogramhívások
- A változók tárolása a memóriában

Adunk egy absztrakt modellt!

10.1 Végrehajtási verem

Végrehajtási verem

```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
}
int main()
{
    f();
    h();
    return 0;
}
```

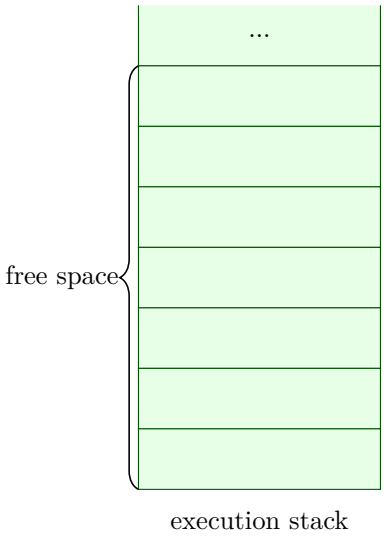
Execution stack

- Alprogramhívások logikája
  - LIFO: Last-in-First-Out
  - Verem adatszerkezet
- Minden alprogramhívásról egy bejegyzés
  - Aktivációs rekord
  - Például információ arról, hova kell visszatérni
- Verem alja: főprogram aktivációs rekordja
- Verem teteje: ahol tart a programvégrehajtás

Alprogramhívások nyilvántartása

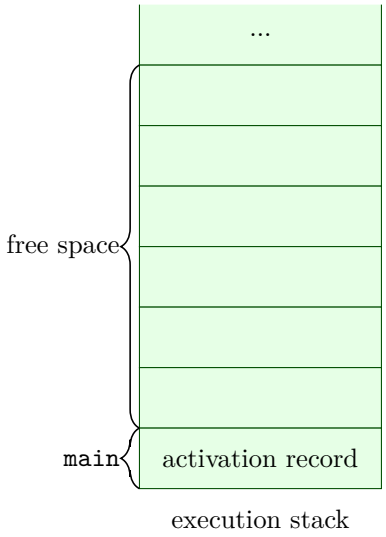
```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
```

```
}
int main()
{
    f();
    h();
    return 0;
}
```



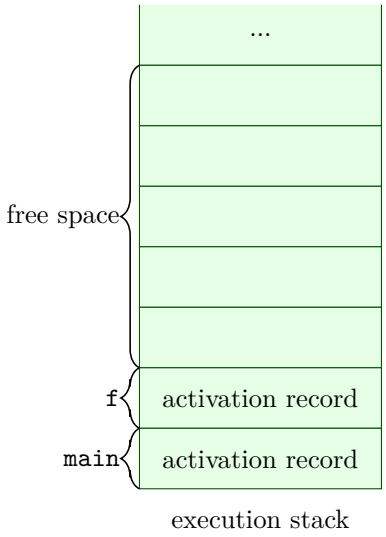
Alprogramhívások nyilvántartása

```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
}
int main()
{
    f();
    h();
    return 0;
}
```



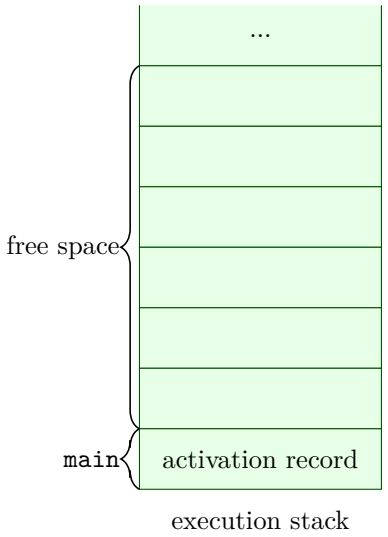
Alprogramhívások nyilvántartása

```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
}
int main()
{
    f();
    h();
    return 0;
}
```



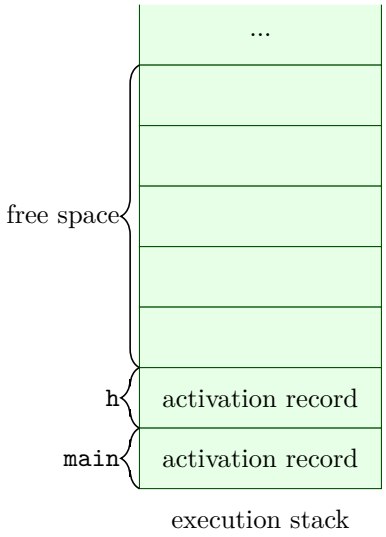
Alprogramhívások nyilvántartása

```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
}
int main()
{
    f();
    h();
    return 0;
}
```



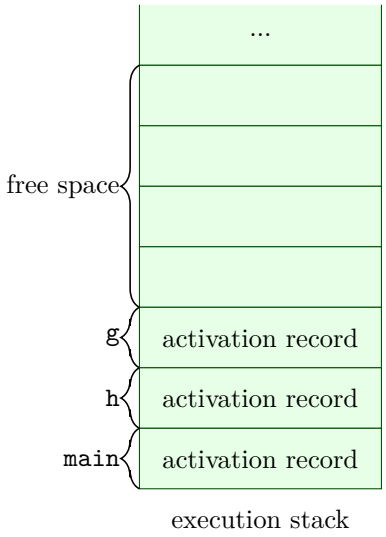
Alprogramhívások nyilvántartása

```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
}
int main()
{
    f();
    h();
    return 0;
}
```



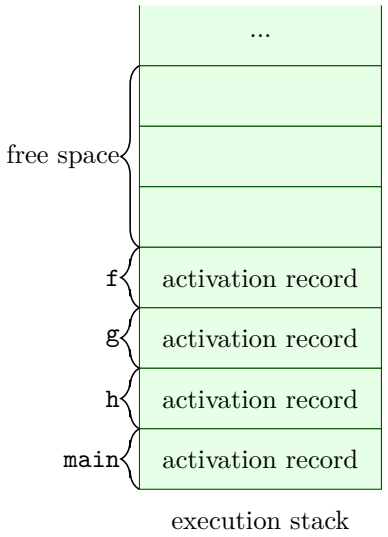
Alprogramhívások nyilvántartása

```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
}
int main()
{
    f();
    h();
    return 0;
}
```



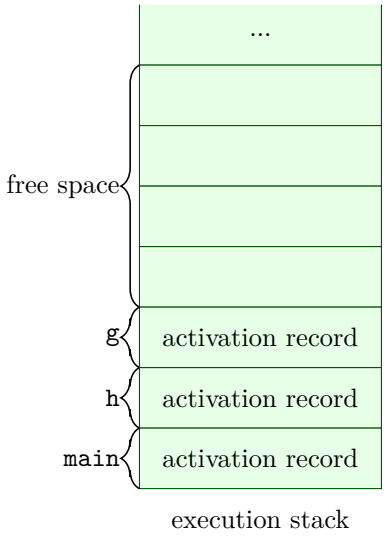
Alprogramhívások nyilvántartása

```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
}
int main()
{
    f();
    h();
    return 0;
}
```



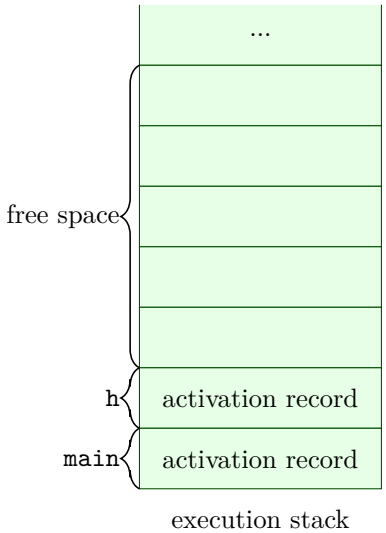
Alprogramhívások nyilvántartása

```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
}
int main()
{
    f();
    h();
    return 0;
}
```



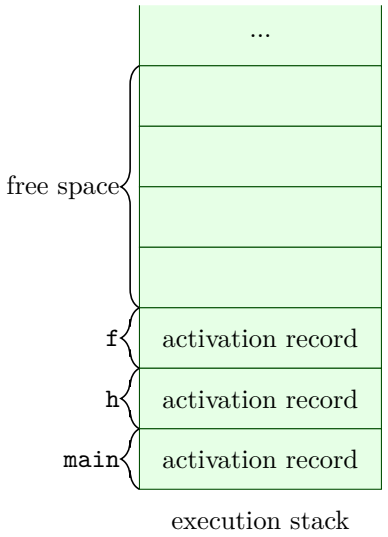
Alprogramhívások nyilvántartása

```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
}
int main()
{
    f();
    h();
    return 0;
}
```



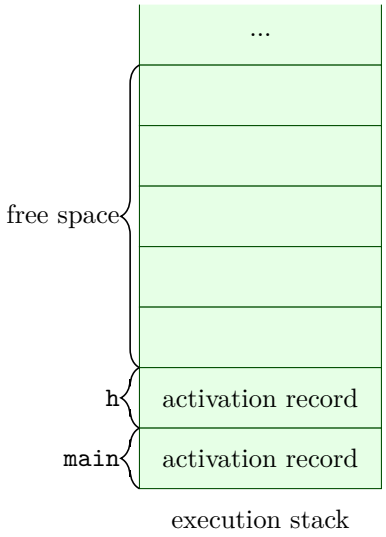
Alprogramhívások nyilvántartása

```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
}
int main()
{
    f();
    h();
    return 0;
}
```



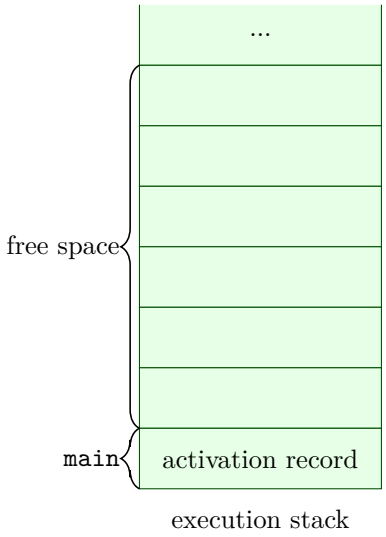
Alprogramhívások nyilvántartása

```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
}
int main()
{
    f();
    h();
    return 0;
}
```



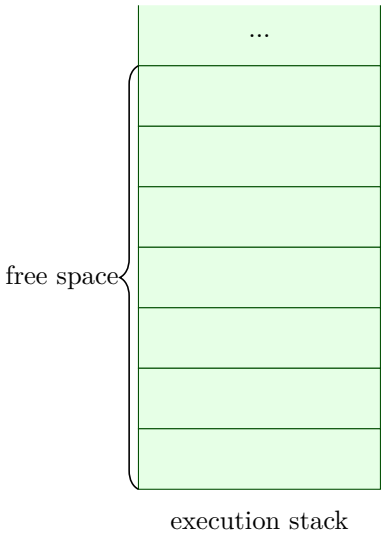
Alprogramhívások nyilvántartása

```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
}
int main()
{
    f();
    h();
    return 0;
}
```



Alprogramhívások nyilvántartása

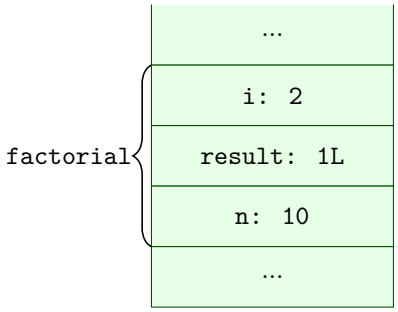
```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
}
int main()
{
    f();
    h();
    return 0;
}
```



Aktivációs rekord

- Mindenféle technikai dolgok
- Alprogram paraméterei
- Alprogram (egyes) lokális változói

```
long factorial( int n )
{
    long result = 1L;
    int i = 2;
    for( ; i<=n; ++n )
        result *= i;
    return result;
}
```



execution stack

Rekurzió

- Egy alprogram saját magát hívja
  - Közvetlenül
  - Közvetve
- Minden hívásról új aktivációs rekord
- Túl mély rekurzió: Stack Overflow
- Költség: aktivációs rekord építése/lebontása

10.2 Változók élettartama és tárolása

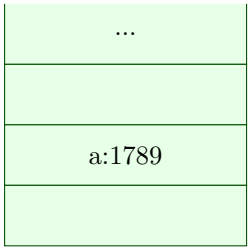
„Változók” tárolása a memóriában

- statikus tárhely → statikus
- végrehajtási verem → automatikus
- dinamikus tárhely (heap) → dinamikus

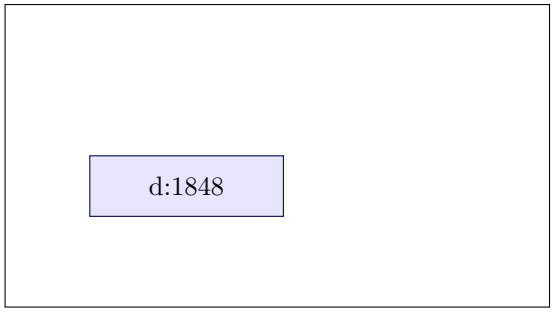
Itt most a *változó* szót nagyon általános értelemben tekintjük. Nem csak azt tekintjük változónak, amit deklarációval hoztunk létre, és aminek nevet adtunk, hanem azt is, amit a dinamikus tárhelyen allokálunk programunk adatainak tárolására.

static - stack - heap

static



stack



heap

}

Statikus tárolású változó

- Statikus tárhely
  - Statikus deklarációkiértékelés
  - A fordító tudja, mekkora tár kell
- Pl. globális változók
- Élettartam: a program elejétől a végéig

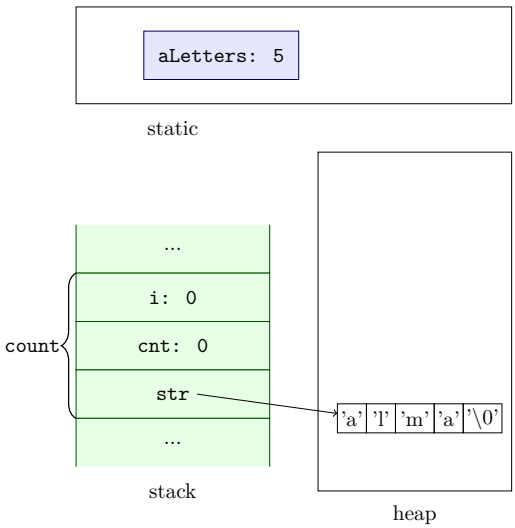
```
int counter = 0;
void signal(void)
{
    ++ counter;
}
```

Automatikus tárolású változó

- Végrehajtási vermen
  - Az aktivációs rekordokban
- A lokális változók *általában* ilyenek
- Élettartam: blokk végrehajtása
  - Automatikusan jön létre és szűnik meg

```
int lnko( int a, int b ){
    int c;
    while( b != 0 ){
        c = a % b;
        a = b;
        b = c;
    }
    return a;
}
```

static - stack - heap



```c
int aLetters = 0;
int count( char *str )
{
    int cnt=0, i=0;
    while (str[i]!='\0')
    {
        if (str[i]=='a')
            ++cnt;
        ++i;
    }
    a_letters += cnt;
    return cnt;
}
```

C - statikus lokális változók

- `static` kulcsszó
- Hatókör: lokális változó
  - Információelrejtés elve
- Élettartam: mint globális változónál

```c
int counter = 0;
void signal(void)
{
    ++ counter;

}
```

```c
int signal(void)
{
    static int counter = 0;
    ++ counter;
    return counter;
}
```

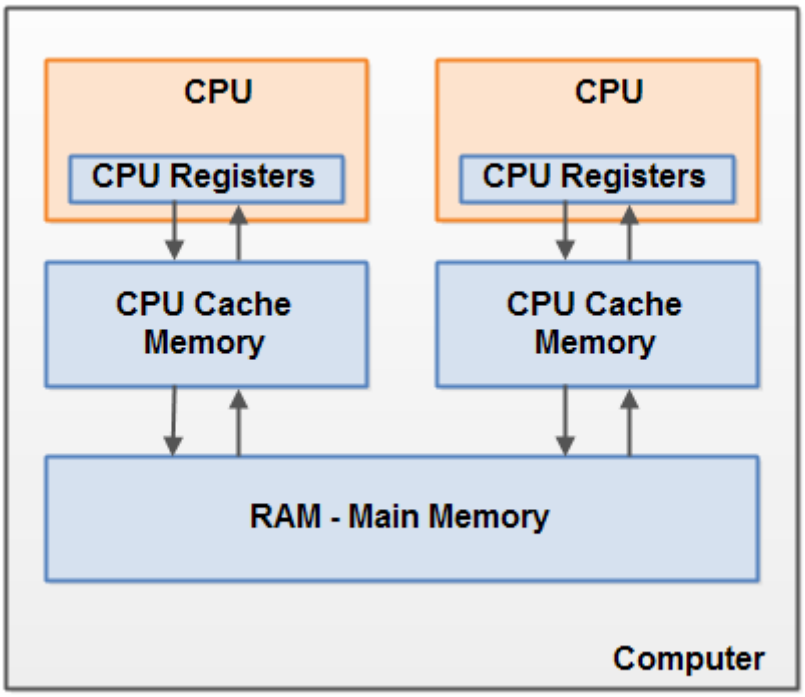
Tárolási mód kifejezése

- `static`
  - lokális
  - globális
- `auto` (nem használjuk)
  - C++ nyelvben az `auto` kulcsszó mást jelent
- `register` (nem használjuk)
  - optimalizáció

```c
int lnko( int a, int b ){
    auto int c;
    while( b != 0 ){
        c = a % b;
        a = b;
```

```
        b = c;
    }
    return a;
}
```

Számítógép memóriája



Optimalizáció: memóriaműveletek emberi skálán

Forrás: David Jeppesen

órajel	0.4 ns	1 sec
L1 cache	0.9 ns	2 sec
L2 cache	2.8 ns	7 sec
L3 cache	28 ns	1 min
DDR memória	~100 ns	4 min
SSD I/O	50-150 microsec	1,5-4 nap
HDD I/O	1-10 ms	1-9 hónap
Internet	65 ms	5-10 év

Globális változók használata

Kerülendő!

Változók definiálása

Deklarációval

- Statikus és automatikus tárolású
  - Statikus tárhely
  - Végrehajtási verem
- Élettartam: programszerkezetből

  - A hatókör
  - Kivéve lokális statikus

Allokáló utasítással

- Dinamikus tárolású
  - Heap (dinamikus tárhely)
- Élettartam: programozható
- Felszabadítás
  - Felszabadító utasítás (C, C++)
  - Szemétgyűjtés (Haskell, Python, Java)

Blokk utasítás

- Új hatókör, lokális deklarációkkal
  - Névtér szennyeződése elkerülhető
- Automatikus tárolású változók
  - Élettartam lerövidíthető

10.3 Paraméterátadás

Alprogram paraméterei

- Definícióban: formális paraméterlista
- Hívásnál: aktuális paraméterlista

Paraméterátadási technikák

- Többféle paraméterátadás van a különféle nyelvekben
  - Érték szerinti (pass-by-value, call-by-value)
  - Érték-eredmény szerinti (call-by-value-result)
  - Eredmény szerinti (call-by-result)
  - Cím szerinti (call-by-reference)
  - Megosztás szerinti (call-by-sharing)
  - Igény szerinti (call-by-need)
  - Név szerinti (call-by-name)
- Végrehajtási verem!

Érték szerinti paraméterátadás

- Formális paraméter: automatikus tárolású lokális változó
- Aktuális paraméter: kezdőérték
- Hívás: az aktuális paraméter értéke bemásolódik a formális paraméterbe
- Visszatérés: a formális paraméter megszűnik

Érték szerinti paraméterátadás – példa

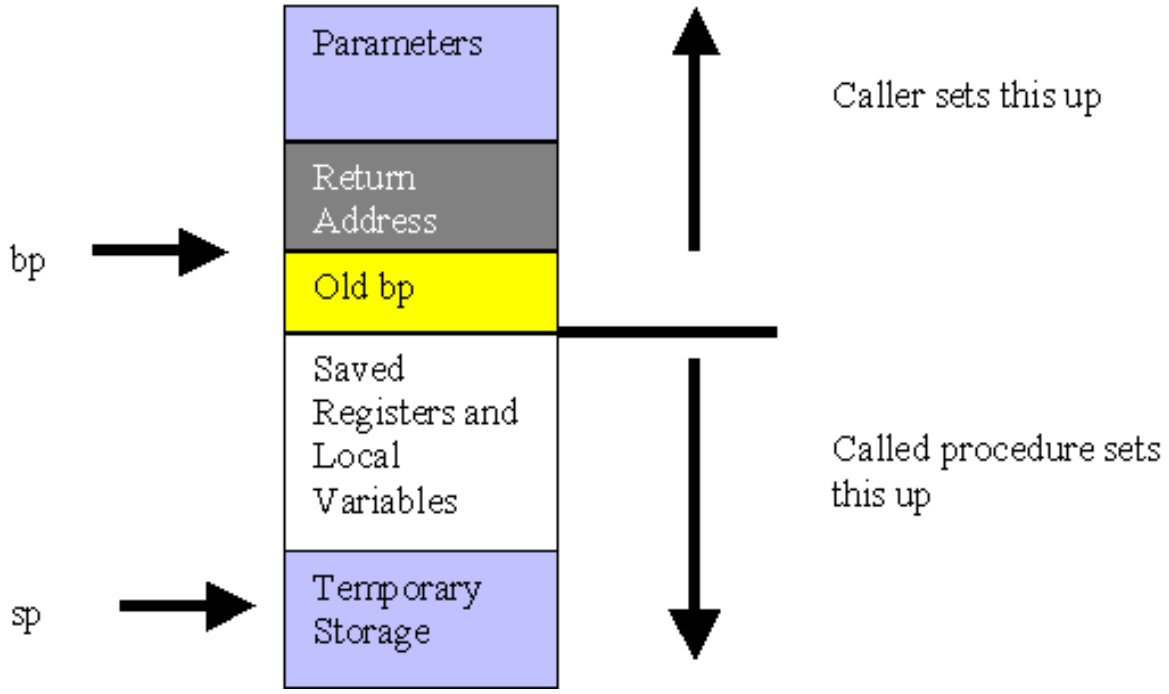
```
int lnko( int a, int b )
{
    int c;
    while( b != 0 ){
        c = a % b;
        a = b;
        b = c;
```

```
    }
    return a;
}
int main()
{
    int n = 1984, m = 356;
    int r = lnko(n,m);
    printf("%d %d %d\n",n,m,r);
}
```

Aktivációs rekord

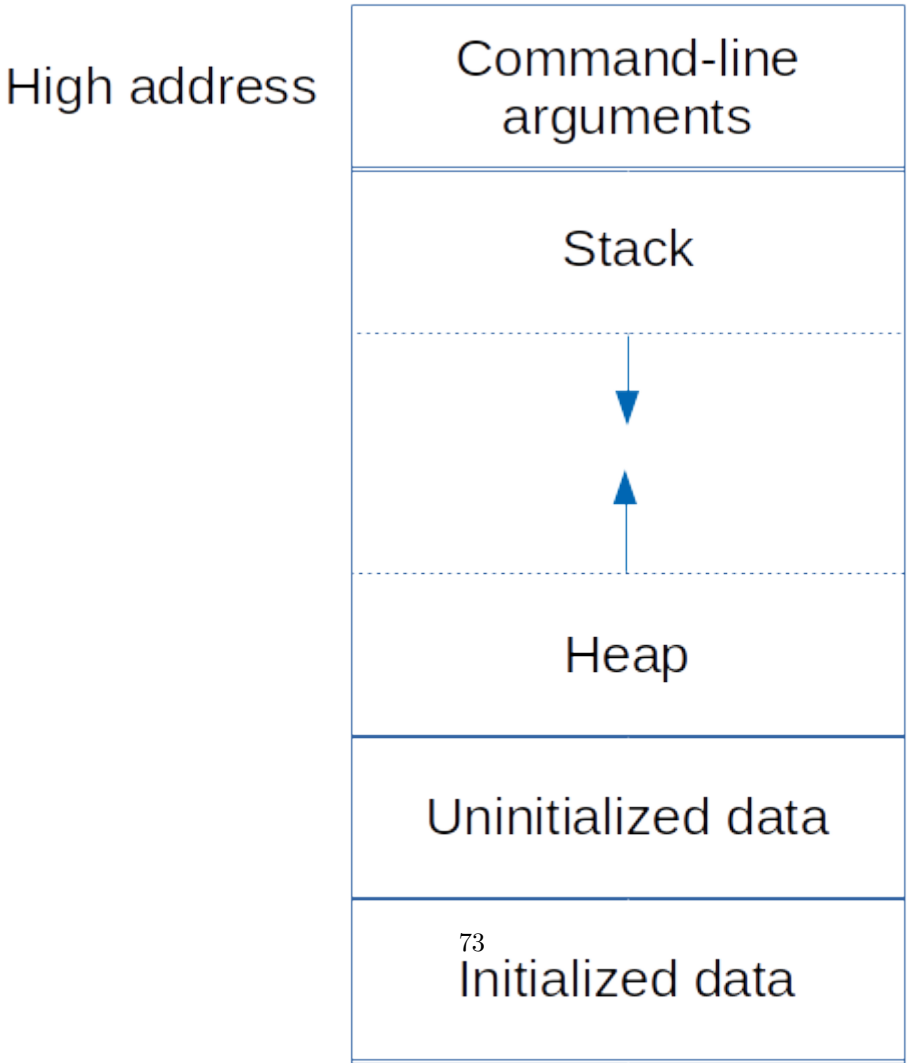
- Mindenféle technikai dolgok
- Alprogram automatikus tárolású változói
  - Pl. az alprogram formális paraméterei
  - Kivéve a regiszterekben átadott paramétereket

Precízebben



11

C programok címtere



  - Heap (dinamikus tárhely)
- Élettartam: programozható
  - Létrehozás: allokáló utasítással
  - Felszabadítás
    * Felszabadító utasítás (C)
    * Szemétgyűjtés – garbage collection (Haskell, Python, Java)
- Használat: indirekció
  - Mutató – pointer (C)
  - Referencia – reference (Python, Java)

Mutatók C-ben

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

int main()
{
    int *p;
    p = (int*)malloc(sizeof(int));
    if( NULL != p )
    {
        *p = 42;
        printf("%d\n", *p);
        free(p);
        return 0;
    }
    else return 1;
}
```

Összetevők

- Mutató (típusú) változó: `int *p;`
  - Vigyázat: `int* p, v;`
  - Hasonlóan: `int v, t[10];`
- Dereferálás (hova mutat?): `*p`
- „Sehova sem mutat”: `NULL`
- Allokálás és felszabadítás: `malloc` és `free` (`stdlib.h`)
  - Típuskényszerítés: `void*` → pl. `int*`

Mire jó?

- Dinamikus méretű adat(-struktúra)
- Láncolt adatszerkezet
- Kimenő szemantikájú paraméterátadás
- …

Dinamikus méretű adatszerkezet

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

int main( int argc, char* argv[] ){
    int *nums = (int*)malloc((argc-1)*sizeof(int));
    if( NULL != nums ){
        int i;
```

```
        for( i=1; i<argc; ++i ) nums[i] = atoi(argv[i]);
        /* TO DO: sort nums */
        for( i=1; i<argc; ++i ) printf("%d\n", nums[i]);
        free(nums);
        return 0;
    } else return 1;
}
```

Kerülendő megoldás

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

int main( int argc, char* argv[] ){
    int nums[argc-1];
    int i;
    for( i=1; i<argc; ++i ) nums[i] = atoi(argv[i]);
    /* TO DO: sort nums */
    for( i=1; i<argc; ++i ) printf("%d\n", nums[i]);
    free(nums);
    return 0;
}
```

- C99: Variable Length Array (VLA)
- Nincs az ANSI C és C++ szabványokban

Láncolt adatszerkezet

- Sorozat típus
- Bináris fa típus
- Gráf típus
- ...

Bejárás közben konstans idejű törlés/beszúrás

Aliasing

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

void dummy(void)
{
    int *p, *q;
    p = (int*)malloc(sizeof(int));
    if( NULL != p ){
        q = p;
        *p = 42;
        printf("%d\n", *q);
        free(p);
    }
}
```

Felszabadítás

Minden dinamikusan létrehozott változót pontosan egyszer!

- Ha többször próbálom: hiba

- Ha egyszer sem: „elszivárog a memória” (memory leak)

Felszabadított változóra hivatkozni hiba!

Hivatkozás felszabadított változóra

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

void dummy(void)
{
    int *p, *q;
    p = (int*)malloc(sizeof(int));
    if( NULL != p ){
        q = p;
        *p = 42;
        free(p);
        printf("%d\n", *q);   /* hiba */
    }
}
```

Többszörösen felszabadított változó

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

void dummy(void)
{
    int *p, *q;
    p = (int*)malloc(sizeof(int));
    if( NULL != p ){
        q = p;
        *p = 42;
        printf("%d\n", *q);
        free(p);
        free(q);      /* hiba */
    }
}
```

Fel nem szabadított változó

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

void dummy(void)
{
    int *p, *q;
    p = (int*)malloc(sizeof(int));
    if( NULL != p ){
        q = p;
        *p = 42;
        printf("%d\n", *q);
    }      /* hiba */
}
```

Tulajdonos?

```
void dummy(void)
{
    int *q;
    {
        int *p = (int*)malloc(sizeof(int));
        q = p;
        if( NULL != p ){
            *p = 42;
        }
    }
    if( NULL != q ){
        printf("%d\n", *q);
        free(q);
    }
}
```

Könnyű elrontani!

```
int *produce( int argc, char* argv[] ){
    int *nums = (int*)malloc((argc-1)*sizeof(int));
    if( NULL != nums ){
        for( int i=1; i<argc; ++i ) nums[i] = atoi(argv[i]);
    }
    return nums;
}
void consume( int *nums ){
    for( i=1; i<argc; ++i ) printf("%d\n", nums[i]);
    free(nums);
}
int main( int argc, char* argv[] ){
    int *nums = produce(argc,argv);
    if( NULL != nums ){ /* TO DO: sort nums */ consume(nums); }
    return (NULL == nums);
}
```

Alias

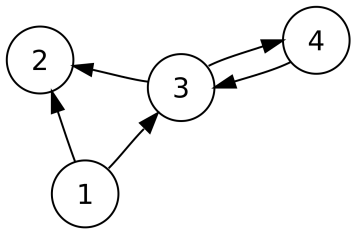
ugyanarra a tárterületre többféle névvel hivatkozhatunk

```
int xs[] = {1,2,3};
int *ys = xs;
xs[2] = 4;
printf("%d\n", ys[2]);
```

Amikor ugyanazt a tárterületet többféle névvel (változóval, kifejezéssel) is elérhetjük, akkor azt mondjuk, hogy ezek a nevek egymás álnevei, *alias*ai. A fenti példában az *ikszek* és az *ipszilonok* (`xs` és `ys`) ugyanazt a számsorozatot jelölik. Amikor értékül adjuk az `ys`-nek az `xs`-t, akkor aliast hozunk létre. Az aliasing nagyon be tudja csapni az embert: ha egy név segítségével megváltoztatjuk a tárhelyen tárolt értéket, akkor ez a változás a másik néven keresztül is megfigyelhetővé válik. Ezért ír ki a fenti példa 4-et. (A mutatók és a tömbök kapcsolata C-ben elég érdekes, erre később még visszatérünk.)

Az aliasing jelenséggel találkozhatunk akkor, amikor dinamikus tárolású változókat kezelünk, mint ebben a példában, de máskor is (például cím szerinti paraméterátadásnál). Könnyű az alias miatt hibát véteni, de maga a jelenség nagyon hasznos is tud lenni. Például, ha irányított gráfokat szeretnénk a programunkban ábrázolni, akkor a gráf éleit hivatkozásokkal (mutatókkal, referenciákkal) adhatjuk meg. Az

aliasing ebben az esetben annak felel meg, hogy a gráf egy csúcsába több más csúcsból vezet él (azaz a csúcs be-foka nagyobb, mint 1).



Mutató gyűjtőtípusa

```
int *p = (int *)malloc(sizeof(int));
if( NULL != p )
{
    *p = 123;
    *p = 12.3;  /* automatikus típuskonverzió */
    printf("%d\n", *p);
    free(p);
}
```

A C-ben egy mutató típusú változóval hivatkozunk a dinamikus tárolású változókra. Ennek a változónak a típusa állandó, mindig csak ugyanolyan típusú adatra hivatkozhat. Ez a statikus típusrendszer következménye. Az `int *p` deklaráció után a `p` egy olyan mutató lesz, amellyel csak `int` típusú tárterületre hivatkozhatunk. Ezt úgy is szoktuk mondani, hogy a `p` *gyűjtőtípusa* az `int`. A fenti példában a `12.3` érték automatikusan konvertálódik `int` típusúra, amikor értékül adjuk a `*p`-nek, mert a fordító tudja, hogy a `*p` típusa `int`.

Mutató gyűjtőtípusa: típuskényszerítés

```
float *q = (float *)malloc(sizeof(float));
if( NULL != q )
{
    int *p = (int *)q;
    *q = 12.3;
    printf("%d\n",*p);
    free(q);
}
```

Kivételt jelent a fenti szabály alól, ha típuskényszerítést, *type cast*ot alkalmazunk. Ez erőszakkal éri el azt, amit a statikus típusrendszer egyébként nem enged meg. Például egy `float *q` változó hivatkozhat ugyanarra a tárterületre, mint a `p`, ha ezt az értékadást végrehajtjuk: `p = (int *)q`. Ha a `*q` hivatkozáson keresztül beállítunk egy `float` értéket a dinamikus változóba, akkor a `*p` hivatkozáson keresztül egy „fura” értéket fogunk látni. Épp emiatt a fura érték miatt várja el a fordító, hogy a típuskényszerítés használatával megerősítsük a döntésünket, miszerint a `p` és a `q` egymás aliasa legyen.

Dinamikus tárhely elérése

- Explicit (mutató)
- Statikus típusellenőrzés
- Erősen típusos
- Felszabadítás

Mutató nem dinamikus változóra

```
int global = 1;

void dummy(void)
{
   int local = 2;
   int *ptr;
   ptr = &global; *ptr = 3;
   ptr = &local;  *ptr = 4;
}
```

Érvénytelen mutató

Értelmetlen

```
int *make_ptr(void)
{

    int n = 42;
    return &n;
}
```

Értelmes

```
int *make_ptr(void)
{
    int *ptr = (int*)malloc(sizeof(int));
    *ptr = 42;
    return ptr;
}



        printf("%d\n", *make_ptr());



int *make_ptr(void)
{
    int *ptr = (int*)malloc(sizeof(int));
    *ptr = 42;
    return ptr;
}

int main(){
    int *ptr = make_ptr();
    if( NULL != ptr ){
        printf("%d\n", *make_ptr());
        free(ptr);
        return 0;
    } else return 1;
}
```

13 Statikus programszerkezet

Statikus programszerkezet

- kifejezés
- utasítás
- alprogram
- modul

Modul

- Nagyobb egység
- Nagy belső kohézió
- Szűk interfész
  - Gyenge kapcsolat modulok között
  - Jellemzően egyirányú

Modulok C-ben

- Fordítási egységek
- Forráskód: `.c` és `.h`
- `#include`
- Szerkesztés: statikus vagy dinamikus

C - statikus globális deklarációk

- Más fordítási egységben nem érjük el
- „Belső szerkesztésű” (internal linkage)
- Az implementációhoz tartozik
- Nem része a modul interfészének
- Információ elrejtés elve

```
int positive = 1;
static int negative = -1;
extern int increment;

static void compensate(void){
    negative -= increment;
}

void signal(void){
    positive += increment;
    compensate();
}
```

Több modulból álló C program

```
gcc -c -W -Wall -pedantic -ansi main.c
gcc -c -W -Wall -pedantic -ansi positive.c
gcc -o main -W -Wall -pedantic -ansi positive.o main.o
```

positive.c

```
int positive = 1;
static int negative = -1;
extern int increment;

static void compensate(void){
    negative -= increment;
}
void signal(void){
    positive += increment;
    compensate();
}
```

main.c

```
#include <stdio.h>

int increment = 3;
extern int positive;
extern void signal(void);

int main(){
    signal();
    printf("%d\n", positive);
    return 0;
}
```

Fejállományok

- „header files“: `.h`
- Modulok közötti interfész
  - `extern`
  - nem `static`
- Modulban és kliensében `#include`
  - típusegyeztetés
- Fordítási egységek közötti függőségek
  - independent compilation
  - szerkesztés feladata
  - fordítási folyamat: `make`

Include guard

vector.h (részlet)

```
#ifndef VECTOR_H
#define VECTOR_H

#define VEC_EOK      0
#define VEC_ENOMEM   1

struct VECTOR_S;
typedef struct VECTOR_S *vector_t;

extern int vectorErrno;

extern void *vectorAt( vector_t v, size_t idx);
extern void  vectorPushBack( vector_t v, void *src);
```

```
#endif
```

14 Paraméterátadás

Függvénydeklarációk és -definíciók

```
int f( int n );
int g( int n ){ return n+1; }

int h();
int i(void);

int j(void){ return h(1); }

int h( int p, int q ){ return p+q; }

extern int k(int,int);

int printf( const char *format, ... );
```

Az `f`-et csak deklaráltuk, nem definiáltuk. A `g`-t definiáltuk is.

A `h`-t úgy deklaráltuk, hogy nem adtuk meg, hogy milyen paramétere(i) van(nak). Az ANSI C előtt még így deklaráltunk függvényeket, és visszafelé kompatibilitási okokból ez a forma még mindig legális – de kerülendő! Ha azt akarjuk kifejezni, hogy egy függvénynek nincsen paramétere, akkor írjuk úgy, mint az `i` esetében: legyen egy `void` a paraméterlistában. (Szokás ezt úgy mondani, hogy a `h`-hoz egy *nem prototípusos* deklarációt adtunk.)

A C++-ban a `h` deklarációja mást jelent, mint a C-ben: paraméter nélküli függvényt deklarál. A C++-ban már nincs meg az a lehetőség, hogy elhagyjuk a deklarációban a paraméterlistát!

A `j` esetén egy paraméter nélküli függvényt definiáltunk. A definícióban a `h`-t egy paraméterrel hívjuk – bár rögtön ez után két paraméterrel definiáljuk a `h`-t. Ebből persze baj lesz: fordítási hibát nem kapunk, de a `h` hibásan, definiálatlanul fog működni a `j`-ből meghívva (definiálatlan memóriatartalmat fog használni).

A `k` deklarációja azt mutatja be, hogy egyrészt a függvénydeklarációk esetén kiírhatjuk az `extern` kulcsszót, másrészt, hogy a prototípusból a paraméterek nevét nyugodtan elhagyhatjuk – azoknak legfeljebb csak dokumentációs szerepük van.

Az utolsó példa a jól ismert `printf` függvény deklarációja. A három pont (angolul *elipsis*) a paraméterlistában azt jelzi, hogy a függvénynek az első, explicit paramétere utan még akárhály további paramétere lehet. Nem könnyű ilyen jellegű függvényeket írni. A trükk itt az, hogy a `format` paraméter tartalmazza azt az információt, hogy mik lesznek a további paraméterek: hány paraméter lesz, és milyen típusúak lesznek.

Paraméterátadási technikák

- **Érték szerinti** (pass-by-value, call-by-value)
- Érték-eredmény szerinti (call-by-value-result) – Ada
- Eredmény szerinti (call-by-result) – Ada
- Cím szerinti (call-by-reference) – Pascal, C++
- Megosztás szerinti (call-by-sharing) – Java, Python
- Igény szerinti (call-by-need) – Haskell
- Név szerinti (call-by-name) – Scala
- Szövegszerű helyettesítés – C-makró

Érték szerinti paraméterátadás

```
int lnko( int a, int b )
{
    int c;
    while( b != 0 ){
        c = a % b;
        a = b;
        b = c;
    }
    return a;
}
int main()
{
    int n = 1984, m = 356;
    int r = lnko(n,m);
    printf("%d %d %d\n",n,m,r);
}
```

Bemenő szemantika

```
void swap( int a, int b )
{
    int c = a;
    a = b;
    b = c;
}

int main()
{
    int n = 1984, m = 356;
    swap(n,m);
    printf("%d %d\n",n,m);
}
```

Mutató átadása érték szerint

```
void swap( int *a, int *b )
{
    int c = *a;
    *a = *b;
    *b = c;
}



int main()
{
    int *n, *m;
    n = (int*) malloc(sizeof(int));
    m = (int*) malloc(sizeof(int));
    if( n != NULL && m != NULL )
    {
        *n = 1984;
        *m = 356;
```

```
        swap(n,m);
        printf("%d %d\n",*n,*m);
        free(n); free(m);
        return 0;
    } else return 1;
}
```

Cím szerinti paraméterátadás emulációja

```
void swap( int *a, int *b )
{
    int c = *a;
    *a = *b;
    *b = c;
}



int main()
{
    int n = 1984, m = 356;
    swap(&n,&m);
    printf("%d %d\n",n,m);
}
```

Cím szerinti paraméterátadás – Pascal

```
program swapping;

   procedure swap( var a, b: integer );  (* var: cím szerint *)
   var
      c: integer;
   begin
      c := a; a := b; b := c
   end;

   var n, m: integer;

begin
   n := 1984; m := 356;
   swap(n,m);
   writeln(n,' ',m)     (* 356 1984 *)
end.
```

Cím szerinti paraméterátadás – C++

```
#include <cstdio>

void swap( int &a, int &b )   /* &: cím szerint */
{
    int c = a;
    a = b;
    b = c;
}
```

```c
int main()
{
    int n = 1984, m = 356;
    swap(n,m);
    printf("%d %d\n",n,m);
}
```

Érték-eredmény szerinti paraméterátadás: Ada

```ada
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Swapping is

   procedure Swap( A, B: in out Integer ) is  -- be- és kimenő
      C: Integer := A;
   begin
      A := B; B := C;
   end Swap;

   N: Integer := 1984;
   M: Integer := 356;

begin
   Swap(N,M);
   Put(N); Put(M);  -- 356 1984
end Swapping;
```

Az érték-eredmény szerinti paraméterátadásról szót ejtettünk már az előző előadáson. Mechanizmusa hasonlít az érték szerinti paraméterátadáshoz, de a hívás végén a formális paraméternek megfelelő lokális változó tartalma visszaíródik az aktuális paraméterbe. Tehát nem csak a híváskor történik információátadás a hívóból a hívottba (bemenő paraméter), hanem a hívás befejeződésekor is a hívottból a hívóba (kimenő paraméter). Az Adában az `in out` kulcsszavak jelzik, hogy be- és kimenő paraméterekről van szó, és az `Integer` típusú ilyen paraméterek esetén érték-eredmény szerinti paraméterátadás történik.

Természetesen ennek a működésnek szükséges feltétele az, hogy az aktuális paraméter értéket tudjon kapni: ne mondjuk egy literál legyen, hanem egy úgynevezett *balérték*-kifejezés, egy olyan kifejezés, amely állhat egy értékadás baloldalán. Nyilván értelmetlen lenne a `Swap(1984,356)` hívás, de az `N` és `M` változók megfelelő aktuális paraméterek.

Megosztás szerinti paraméterátadás

```c
void swap( int t[] )
{
    int c = t[0];
    t[0] = t[1];
    t[1] = c;
}
int main()
{
    int arr[] = {1,2};
    swap(arr);
    printf("%d %d\n",arr[0],arr[1]);
}
```

Ez nem cím szerinti paraméterátadás

```
void twoone( int t[] )
{
    int arr[] = {2,1};
    t = arr;
}

int main()
{
    int arr[] = {1,2};
    twoone(arr);
    printf("%d %d\n",arr[0],arr[1]);
}
```

Automatikus változó visszaadása?

Hibás!

```
int *twoone()
{
    int arr[] = {2,1};
    return arr;
}
```

Igény szerinti paraméterátadás

```
f True  a _ = a
f False _ b = b + b

main = print result
  where result = f False (fact 20) (fact 10)

        fact 0 = 1
        fact n = n * fact (n-1)
```

Ebben a Haskell programban az `f False (fact 20) (fact 10)` kifejezés értékét írjuk ki a képernyőre. Az `f` függvény ezen meghívása során csak azok az aktuális paraméterek értékelődnek ki, amelyekre az eredmény meghatározásához szükség van. Mivel az első aktuális paraméter `False`, a második ág kerül kiértékelésre a definícióban, így a második aktuális paraméterre nincs szükség: a program végrehajtása során `fact 20` nem kerül kiszámításra. A harmadik aktuális paramétert viszont ki kell értékelni, ki kell számolni `fact 10` értékét, hogy a kétszeresét visszaadhassuk.

A név szerinti paraméterátadás egészen hasonló az igény szerintihez, de az `f` második ágában a `b+b` kifejezés értékének meghatározásához a harmadik aktuális paramétert, a `fact 10` értéket kétszer is kiszámítja. Ezt figyelhetjük meg az alábbi Scala kódban. (Scalában érték szerinti és megosztás szerinti paraméterátadás van, hacsak nem helyezzük el a formális paraméter deklarációjában a `=>` jelet.)

```
def f( condition: Boolean, a: => Int, b: => Int ): Int =
   if (condition) a else b + b

def fact(n: Int): Int = n match {
  case 0 => 1
  case _ => n * fact(n-1)
}

f(false, {print("a"); fact(20)}, {print("b"); fact(10)})
```

Ez a Scala szkript kétszer is kiírja a `b` betűt a végeredmény kiírása előtt, viszont az `a` betűt egyszer sem. Ebből látszik, hogy a második aktuális paramétert egyszer sem, míg a harmadik aktuális paramétert kétszer is kiértékeli.

Szövegszerű helyettesítés

```
#define DOUBLE(n) 2*n
#define MAX(a,b) a>b?a:b

int main()
{
    printf("%d %d\n", MAX(10,100), DOUBLE(10));
    {
       int n = 5;
       printf("%d\n", DOUBLE(n+1));
       printf("%d\n", MAX(5,++n));
    }
}
```

A C preprocesszor a fordító előtt fut le, és a forráskódot átalakítja. (Kimenete: fordítási egység.) A preprocesszálás legismertebb lépései az `#include` és a `#define` direktívák végrehajtása, melyek hatására a forráskódba bemásolódnak fájlok (jellemzően *header*-állományok), illetve makrók kifejtésre kerülnek. A `#define` direktívákkal tehát makrók definiálhatók. Korábban említést tettünk paraméter nélküli makrókról: gyakran arra használjuk ezeket, hogy literálokhoz nevet vezessünk be, és ezzel a kód olvashatóságát, karbantarthatóságát növeljük.

Definiálhatunk azonban paraméterrel rendelkező makrókat is, mint azt a fenti példán láthatjuk. Ezek olyan szerepet tölthetnek be, mint a függvények: valamilyen számítás nevesített absztrakcióját adhatjuk meg általuk. A fenti `DOUBLE` makró kiszámítja paramétere kétszeresét, a `MAX` makró pedig két érték közül a nagyobbikat (vagy egy nem kisebbiket) választja ki.

Első ránézésre sok a hasonlóság a függvények és a makrók között. Ez azonban nem szabad, hogy megtévesszen bennünket. Könnyen okozhatnak fura hibákat a programunkban a makrók – oda kell figyelni, hogy jól használjuk őket. Nézzük, hogy mi is történik a fenti példában. Az első két makróhívás nem okoz problémát, itt az történik, amire mindenki gondol, aki rápillant a kódra. A blokk utasításban szereplő makróhívásokról ez már nem mondható el.

Szövegszerű helyettesítés – becsapós

```
#define DOUBLE(n) 2*n
#define MAX(a,b) a>b?a:b

int main()
{
    printf("%d %d\n", MAX(10,100), DOUBLE(10));
    {
       int n = 5;
       printf("%d\n", DOUBLE(n+1)); /* printf("%d\n", 2*n+1); */
       printf("%d\n", MAX(5,++n));
    }
}
```

Makrók esetében nem olyan paraméterátadás történik, mint a függvények esetén. A makrók definíciója szövegszerűen kifejtésre kerül az alkalmazás helyén, és a makrók aktuális paramétere(i) szövegesen behelyettesítésre kerül(nek) ennek során. Például a blokk utasításban szereplő `DOUBLE(n+1)` kifejtődik a `2*n+1` kifejezésre, ami persze nem ekvivalens azzal, amit elsőre gondoltunk volna: `2*(n+1)`. Az ilyen jellegű, a precedenciák miatt bekövetkező bakikkal szemben úgy védekezhetünk, hogy a makrók definíciójában minden „formális paramétert” bezárójelezünk. Sőt, magát a makrótörzset is érdemes bezárójelezni, mert precedenciával kapcsolatos problémát a törzs kifejtésénél is felléphetnek. Például az `INC(x)`

makrót nem egyszerűen `(x)+1` formában érdemes megadni, mert a `3*INC(1)` a szándékaink ellenére a `3*(1)+1` kifejezésre fejtődik ki, nem a `3*(1+1)`-re.

Szövegszerű helyettesítés – zárójelezés

```
#define DOUBLE(n) (2*(n))
#define MAX(a,b) ((a)>(b)?(a):(b))

int main()
{
    printf("%d %d\n", MAX(10,100), DOUBLE(10));
    {
       int n = 5;
       printf("%d\n", DOUBLE(n+1)); /* (2*((n)+1)) */
       printf("%d\n", MAX(5,++n));
    }
}
```

Tehát a makró definíciójában elég sok zárójelet elhelyezve a `DOUBLE` makrót sikerült úgy elkészítenünk, hogy az különféle hívási környezetben is kettővel való szorzás értelemben működjön.

Vannak azonban olyan makródefiníciók is, amelyek esetén még ez az óvintézkedés sem elegendő. Nézzük az alaposan bezárójelezett `MAX(a,b)` makrót.

Szövegszerű helyettesítés – még így is veszélyes

```
#define DOUBLE(n) (2*(n))
#define MAX(a,b) ((a)>(b)?(a):(b))

int main()
{
    printf("%d %d\n", MAX(10,100), DOUBLE(10));
    {
       int n = 5;
       printf("%d\n", DOUBLE(n+1)); /* (2*((n)+1)) */
       printf("%d\n", MAX(5,++n));  /* ((5)>(++n)?(5):(++n)) */
    }
}
```

A kifejtés hatását jól láthatjuk a kommentben. A precedenciák nem okoznak gondot, de az aktuális paraméterként használt kifejezések mellékhatásai elsőre meglepő eredményt hozhatnak. A blokk utasításában olyan kifejezés jön létre a makró alkalmazásának hatására, amely kiértékelése során a `++n` részkifejezés kétszer is kiértékelődik, így nem 6, hanem 7 végeredményt kapunk. A `MAX(a,b)` makrót csak mellékhatásmentes aktuális paraméterekkel célszerű meghívni.

Annak ellenére, hogy a makrók veszélyesek, mert könnyen megtévesztik a programozókat, sokszor találkozunk velük: a programozók előszeretettel írnak (jellemzően egyszerű) számításokat függvénydefiníció helyett makróval. Ennek az az oka, hogy a makrók kifejtése során olyan kód keletkezik, ami közvetlenül tartalmazza a számítást, a függvényhívás nélkül: így a függvényhívás költsége megspórolható. Persze egy jobb fordítóprogram optimalizációval képes arra, hogy függvények törzsét *inline-osítsa*, ezért az az érv, hogy a makró használatából származó hatékonyságnövekedés ellensúlyozza a makrók veszélyességét, egyszerűen nem állja meg a helyét.

15 Típusok

Típuskonstrukciók

- Felsorolási típusok
- Mutató típusok
- Összetett típusok

15.1 Felsorolási típus

Felsorolási típus

Haskell

```
data Color = White | Green | Yellow | Red | Black
```

C-ben: valójában egy egész szám típusra képződik le

```
enum color { WHITE, GREEN, YELLOW, RED, BLACK };

const char* property( enum color code ){
    switch( code ){
        case WHITE:  return "clean";
        case GREEN:  return "jealous";
        case YELLOW: return "envy";
        case RED:    return "angry";
        case BLACK:  return "sad";
        default:     return "?";
    }
}
```

A felsorolási típusok olyan típusok, amelyeknek a típusértékeit egy felsorolással adjuk meg. Ilyen típusokat sok nyelvben lehet definiálni. Találkozhattunk vele például a Haskellben, ahol algebrai adattípusokkal (`data` konstrukcióval) fejezhetjük ki a felsorolási típusokat. (A típusértékek ebben az esetben a típus „adatkonstruktorai” lesznek.)

A C nyelvben is definiálhatunk felsorolási típusokat az `enum` kulcsszóval. A fenti példában a `color` egy felsorolási típus lesz. Egy fontos tulajdonsága a felsorolási típusoknak, hogy használhatjuk őket `switch` utasításhoz. Ez azért van, mert a felsorolási típusok belül egy egész típussal vannak reprezentálva. Ezért aztán minden műveletet, amit egész számokkal végezhetünk, végezhetjük felsorolási típusokkal is. Más nyelvek (pl. Haskell) szigorúbbak szoktak lenni, és nem keverik össze az egész számok típusát a felsorolási típusokkal, és persze nem engedik egymással összekeveredni a különböző felsorolási típusok értékeit sem. A C nyelv túl lazán bánik ezzel a konstrukcióval!

Vegyük még azt is észre, hogy amikor használjuk a felsorolási típust, az `enum` kulcsszót akkor is kiírjuk! Egy kis trükkel ezen segíthetünk, és az `enum color` típust közelebb hozhatjuk a Haskelles megfelelőjéhez.

Felsorolási típus C-ben

```
enum color { WHITE = 1, GREEN, YELLOW, RED = 6, BLACK };

typedef enum color Color;

const char* property( Color code ){ ... }

int main( int argc, char *argv[] )
{
    for( --argc; argc>0; --argc )
```

```
    {
       printf("%s\n", property( atoi(argv[argc]) ));
    }
    return 0;
}
```

Ebben a példában a `typedef` kulcsszó segítségével egy könnyebben használható típusnevet vezetünk be az `enum color` alternatívájaként: `Color`. A `typedef` kicsit félrevezető elnevezés. Nem egy új típust definiálunk vele, hanem egy meglévő típushoz (pl. `enum color`) egy új nevet, szinonímát vezethetünk be a segítségével: azaz *deklarálhatunk*, nevet rendelhetünk egy entitáshoz ezzel a kulcsszóval. (A `typedef` ennélfogva a Haskell `type` konstrukciójának felel meg.)

Ezen a példán azt is megfigyelhetjük, hogy a felsorolási típus értékeit reprezentáló egész számot meg is adhatjuk a típus definíciójában. Ha nem adunk meg értéket, nullától szoktak számozódni a típusértékek, és egyesével növekednek. Ha valamelyik típusértékhez hozzárendelünk egy egész számot, a rá következő típusértékhez eggyel nagyobb egész szám rendelődik. A fenti példában a `YELLOW` ezért 3, a `BLACK` pedig `7` lesz.

A `main` függvényben a parancssori argumentumokat (amelyek sztringek) az `stdlib` könyvtárban definiált `atoi` függvénnyel konvertáljuk át egész számmá, és az így kapott egész számokat feleltetjük meg aktuális paraméterként a `property` függvény `enum color` típusú formális paraméterének. Ez is az egész és felsorolási típusok közötti könnyű átjárhatóságot (és egyben a felsorolási értékek viszonylag gyenge típusozottságát) illusztrálja.

15.2 Összetett típusú értékek

Összetett típusú értékek

- Sorozat
- Direktszorzat
- Unió típus
- Osztály

Sorozat típusok

- C tömbök
- Haskell listák

Sorozat: azonos típusú elemekből álló összetett típus

Sorozat típusnak nevezzük azt a típust, amelynek típusértékei több, (jellemzően) azonos típusba tartozó értékből állnak. Az egyes elemekre indexeléssel szoktunk hivatkozni. Ilyen sorozat típusnak tekintjük a C tömbjeit, valamint a Haskell listáit.

15.3 Tömbök

Tömb fogalma

Azonos típusú (méretű) objektumok egymás után a memóriában.

- Bármelyik hatékonyan elérhető!
- Rögzített számú objektum!

```
int vector[4];
int matrix[5][3];   /* 15 elem sorfolytonosan */
```

Indexelés 0-tól

- `vector[i]` címe: `vector` címe + `i * sizeof(int)`
- `matrix[i][j]` címe: `matrix` címe + `(i * 3 + j) * sizeof(int)`

C tömbök deklarációja

```
int a[4];                          /* 4 elemű, inicializálatlan */
int b[] = {1, 5, 2, 8};            /* 4 elemű */
int c[8] = {1, 5, 2, 8};           /* 8 elemű, 0-kkal feltöltve */
int d[3] = {1, 5, 2, 8};           /* 3 elemű, felesleg eldobva */

extern int e[];
extern int f[10];                  /* méret ignorálva */

char s[] = "alma";
char z[] = {'a','l','m','a','\0'};

int m[5][3];                       /* 15 elem, sorfolytonosan */
int n[][3] = {{1,2,3},{2,3,4}};    /* méret nem elhagyható! */
int q[3][3][4][3];                 /* 108 elem */
```

Tömbök indexelése

- `int t[] = {1,2,3,4};`
- 0-tól indexelünk
- hossz futás közben ismeretlen
- fordítás közben: `sizeof`
  - `sizeof(t) / sizeof(t[0])`
- hibás index: definiálatlanság

15.4 Mutatók

Mutatók

- Más változókra mutat(hat): indirekció
  - dinamikus
  - automatikus vagy statikus
- Típusbiztos

```
int i;
int t[4];
int *p = NULL;  /* sehova sem mutat */

/* dinamikus tárolású változóra mutat */
p = (int*)malloc( sizeof(int) * i ); ... free(p);

/* statikusra vagy automatikusra mutat */
p = &i;    p = t;

*p = 5;         /* dereferálás */
```

Deklarációk mutatókkal

```
int i = 42;
int *p = &i;
int **pp = &p;          /* mutató mutatóra */
int *ps[10];            /* mutatók tömbje */
int (*pt)[10];          /* mutató tömbre */

char *str = "Hello!";

void *foo = str;        /* akármire mutathat */

int* p,q;               /* mutató és int */
int  s,t[5];            /* int és tömb */
int *f(void);           /* int* eredményű függvény */
int (*f)(void);         /* mutató int eredményű függvényre */
```

Tömbök és mutatók kapcsolata

- Tömb: *second-class citizen*
- Tömb → mutató
- Nem ekvivalensek!

```
int t[] = {1,2,3};
t = {1,2,4};  /* fordítási hiba */

int *p = t;
int *q = &t[0];

int (*r)[3] = &t;

printf( "%d%d%d%d\n", t[0], *p, *q, (*r)[0] );
```

Érdekes módon a tömbök nem „teljes jogú állampolgárai” a C nyelvnek. Ez azt jelenti, hogy egy tömb típusú változót nem lehet olyan általánosan használni, mint mondjuk egy skalár típusú változót. Észrevehettük már, hogy a tömböknek nem lehet értéket adni. Ebből a szempontból a tömbök hasonlítanak a függvényekre. (Ha definiálunk egy `f` nevű függvényt, akkor innentől kezdve az `f` azonosító azt a függvényt fogja jelenteni, nem adhatunk neki „más értéket“.)

Sok esetben akkor, amikor tömböket használunk, valójában nem is tömböket használunk, hanem mutatókat. Ugyanis a tömb hivatkozások sokszor automatikusan mutatóvá konvertálódnak. Ez magyarázza meg azt is, hogy egy mutató típusú változónak miért lehet értékül adni egy tömb típusú változót. Valójában ez az értékadás azt jelenti, hogy a tömb legelső elemének a címét (ami egyben a tömb címe is!) kapja meg a mutató típusú változó.

A típusokkal kapcsolatban akadhat itt némi problémánk. A fenti példában az `r` változó egy mutató, ami egy 3 hosszú egész tömbre hivatkozhat. A `t` címével inicializáltuk, így az `r` a `t` tömb elejére mutat. Az értékét tekintve megegyezik tehát a `p` és a `q` mutatókkal, de az `r`-nek más a típusa. A `p = r` értékadás hibás, de a típuskényszerítéssel a típusok különbözősége áthidalható: `p = (int*)r`.

Ahogy látjuk, a tömbök és a mutatók néha ekvivalensnek tűnnek, de nem azok! Később látunk majd olyan példát, amelyből ez kiderül!

Tömb átadása paraméterként?

Valójában mutató típusú a paraméter!

```
double distance( double a[], double *b )
{
   double dx = a[0] - b[0],
          dy = a[1] - b[1],
          dz = a[2] - b[2];
   return sqrt(dx*dx + dy*dy + dz*dz);
}
```

Mutató-aritmetika – léptetések

```
int v[] = {6, 2, 8, 7, 3};
int *p = v;
int *q = v + 3;   /* v konvertálódik */
++p;
*p = 5;           /* v: 6, 5, 8, 7, 3 */
p += 2;
*q = 1;           /* v: 6, 5, 8, 1, 3 */
q -= 2;
*q = 2;           /* v: 6, 2, 8, 1, 3 */
```

Mutató-aritmetika – összehasonlítások

```
int v[] = {6, 2, 8, 7, 3};
int *p = v;
int *q = v + 3;

if ( p == q ) { ... }
if ( p != q ) { ... }
if ( p < q ) { ... }
if ( p <= q ) { ... }
if ( p > q ) { ... }
if ( p >= q ) { ... }
```

Mutató-aritmetika – indexelés

```
char str[] = "hello";

str[ 1 ] = 'o';
*( str + 1 ) = 'o';

printf( "%s\n", str + 3 );
printf( "%c\n", 3[ str ] );
```

Mutató-aritmetika: példa

```
int strlen( char* s )
{
   char* p = s;
   while( *p != '\0' )
   {
      ++p;
   }
   return p - s;
}
```

Mutatók és tömbök közötti különbségek

```
int v[] = {6, 3, 7, 2};
int *p = v;

v[ 1 ] = 5;
p[ 1 ] = 8;

int w[] = {1,2,3};
p = w;  /* ok */
v = w;  /* fordítási hiba */

printf( "%d %d\n", sizeof( v ), sizeof( p ) );
```

Lásd még az utolsó példát itt: http://gsd.web.elte.hu/lectures/imper/imper-lecture-8/.

15.5 Tömbök átadása paraméterként

Tömbök átadása paraméterként: általánosítás?

```
double distance( double a[3], double b[3] ){
   double sum = 0.0;
   int i;
   for( i=0; i<3; ++i ){      /* beégetett érték :-( */
      delta = a[i] - b[i];
      sum += delta*delta;
   }
   return sqrt( sum );
}

int main(){
   double p[3] = {36, 8, 3}, q[3] = {0, 0, 0};
   printf( "%f\n", distance(p,q) );
   return 0;
}
```

Itt a paraméterlistában a méretre adott megkötés elvész, amikor a tömb „átváltozik” mutatóvá. Inkább csak megtévesztő, hogy odaírtuk a tömbök méretét: semmilyen fordítási idejű vagy futási idejű ellenőrzést nem biztosít ezen információ megadása.

Tömbök paraméterként: fordítási időben rögzített méret

```
#define DIMENSION 3

double distance( double a[DIMENSION], double b[DIMENSION] ){
   double sum = 0.0;
   int i;
   for( i=0; i<DIMENSION; ++i ){
      delta = a[i] - b[i];
      sum += delta*delta;
   }
   return sqrt( sum );
}

int main(){
   double p[DIMENSION] = {36, 8, 3}, q[DIMENSION] = {0, 0, 0};
   printf( "%f\n", distance(p,q) );
```

```
   return 0;
}
```

Tömbök paraméterként: futási időben rögzített méret?

```
double distance( double a[], double b[] ){
   double sum = 0.0;
   int i;
   for( i=0; i<???; ++i ){    /* vajon mekkora? */
      delta = a[i] - b[i];
      sum += delta*delta;
   }
   return sqrt( sum );
}

int main(){
   double p[] = {3.0, 4.0}, q[] = {0.0, 0.0};
   printf( "%f\n", distance(p,q) );
   return 0;
}
```

Tömbök paraméterként: hibás megközelítés

```
double distance( double a[], double b[] ){
   double sum = 0.0;
   int i;
   for( i=0; i<sizeof(a)/sizeof(a[0]); ++i ){
      delta = a[i] - b[i];
      sum += delta*delta;
   }
   return sqrt( sum );
}

int main(){
   double p[] = {3.0, 4.0}, q[] = {0.0, 0.0};
   printf( "%f\n", distance(p,q) );
   return 0;
}
```

Tömbök paraméterként: helyesen

```
double distance( double a[], double b[], int dim ){
   double sum = 0.0;
   int i;
   for( i=0; i<dim; ++i ){
      delta = a[i] - b[i];
      sum += delta*delta;
   }
   return sqrt( sum );
}

int main(){
   double p[] = {3.0, 4.0}, q[] = {0.0, 0.0};
   printf( "%f\n", distance(p,q,sizeof(p)/sizeof(p[0])) );
   return 0;
}
```

Bonyolult struktúra átadása paraméterként

```
int main( int argc, char *argv[] ){ ... }
```

- `argc`: pozitív szám
- `argv[0]`: program neve
- `argv[i]`: parancssori argumentum ($1 \leq i <$ argc)
  - karaktertömb, végén `NUL` (`'\0'`)
- `argv[argc]`: `NULL`

```
int main( void ){ ... }
int main( int argc, char *argv[], char *envp[] ){ ... }
int main(){ ... }
```

Több dimenziós tömbök paraméterként

```
double m[4][4] = {{1,2,3,4},{1,2,3,4},{1,2,3,4},{1,2,3,4}};

transpose(m);

{
   int i,j;
   for( i=0; i<4; ++i ){
      for( j=0; j<4; ++j ){
         printf("%3.0f", m[i][j]);
      }
      printf("\n");
   }
}
```

Túl merev megoldás

```
void transpose( double matrix[4][4] ){  /* double matrix[][4] */
   int size = sizeof(matrix[0])/sizeof(matrix[0][0]);
   int i, j;
   for( i=1; i<size; ++i ){
      for( j=0; j<i; ++j ){
         double tmp = matrix[i][j];
         matrix[i][j] = matrix[j][i];
         matrix[j][i] = tmp;
      }
   }
}

double m[4][4] = {{1,2,3,4},{1,2,3,4},{1,2,3,4},{1,2,3,4}};
transpose(m);
```

Sorfolytonos ábrázolás: egybefüggő memóriaterület

```
void transpose( double *matrix, int size ){  /* size*size double */
   int i, j;
   for( i=1; i<size; ++i ){
      for( j=0; j<i; ++j ){
         int idx1 = i*size+j,  /* matrix[i][j] helyett */
             idx2 = j*size+i;  /* matrix[j][i] helyett */
         double tmp = matrix[idx1];
         matrix[idx1] = matrix[idx2];
```

```
            matrix[idx2] = tmp;
        }
    }
}

double m[4][4] = {{1,2,3,4},{1,2,3,4},{1,2,3,4},{1,2,3,4}};
transpose( &m[0][0], 4 );  /* transpose( (double*)m, 4 ) */
```

Alternatív reprezentáció: mutatók tömbje

```
void transpose( double *matrix[], int size ){
   int i, j;
   for( i=1; i<size; ++i ){
      for( j=0; j<i; ++j ){
         double tmp = matrix[i][j];
         matrix[i][j] = matrix[j][i];
         matrix[j][i] = tmp;
      }
   }
}

double m[4][4] = {{1,2,3,4},{1,2,3,4},{1,2,3,4},{1,2,3,4}};
double *helper[4]; for( i=0; i<4; ++i ) helper[i] = m[i];
transpose(helper,4);
```

15.6 Magasabbrendű függvények

Magasabbrendű függvények

függvénymutatók segítségével

```
/* mutató int eredményű, paraméter nélküli függvényre */
int (*fp)(void);

/* int->int függvényt és int-et váró függvény int eredménnyel */
int twice( int (*f)(int), int n );
```

C: függvénymutatók

```
int twice( int (*f)(int), int n )
{
    n = (*f)(n);
    n = f(n);
    return n;
}

int inc( int n ){ return n+1; }


printf(  "%d\n",  twice( &inc, 5 )  );
```

Házi feladat: tömbök rendezése az `stdlib`-ben definiált `qsort` függvénnnyel.

Függvénymutatók - néhány észrevétel

```
int inc( int n ){ return n+1; }

int (*f)(int) = &inc;
f = inc;
f(3) + (*f)(3);

int (*g)() = inc;
g(3,4);  g();
```

15.7 Konstansok

Konstansok

Definíciók

```
const int i = 3;
int const j = 3;

const int t[] = {1,2,3};


const int *p = &i;

int v = 3;
int * const q = &v;
```

Hibás használat

```
i = 4;
j = 4;

t[2] = 4;
t = {1,2,4};

*p = 4;


q = (int *)malloc(sizeof(int));
```

Mit is jelent ez a deklaráció?

```
int const * a, b;
```

Nem teljes a biztonság

```
const int i = 3;
int * const q = &i;     /* csak warning */
int * p = &i;           /* csak warning */

*p = *q = 4;            /* i megváltozik */
```

const-ra polimorf megoldás nincs

```
char *strchr( const char *str, int c ){
    while( *str != 0 && *str != c ) ++str;
    return str;
```

```
}
```

```
char *p = strchr("Hello",'e'),      q[] = "Hello";
*p = 'o';    /* hibás! */           char *r = strchr(q,'e');
                                    *r = 'o';   /* ok */
```

Élettartammal kapcsolatos hibák

http://gsd.web.elte.hu/lectures/imper/imper-lecture-5/

(legvégén)

Érdemes elolvasni az „Élettartammal kapcsolatos hibák” szakaszt a fenti dokumentumból. A szakasz a dokumentum végén található.

15.8 Direktszorzat típusok

Direktszorzat típusok

(Potenciálisan) különböző típusú elemekből konstruált összetett típus

- tuple
- rekord
- struct

C struct

```
struct month { char *name, int days };   /* típus létrehozása */

struct month jan = {"January", 31};      /* változó létrejozása */

/* three-way comparison */
int compare_days_of_month( struct month left, struct month right )
{
    return left.days - right.days;
}
```

A sorozatok mellett az összetett típusok egy másik nagy családját alkotják azok az összetett típusok, amelyeknek az értékei több, potenciálisan különböző típusú elemekből állnak össze. A Haskell nyelvből megismert *rendezett n-es* (tuple) egyértelműen ide sorolható. Az egyes elemekhez a Haskell nyelvben mintaillesztéssel, valamint könyvtári függvényekkel férhetünk hozzá.

Sok programozási nyelv a direktszorzat típusok létrehozásához a rekord konstrukciót ajánlja. A rekord annyiban más, mint a rendezett n-es, hogy az egyes elemekhez *szelektorokkal* férünk hozzá. Amikor a rekord típust definiáljuk, megadjuk, hogy milyen nevű és típusú *mezőkből* épül fel. Az egyes mezőkhöz (elemekhez) a mezőnév használható szelektorként. A Haskell nyelv is támogatja a rekord konstrukciót.

A C nyelvben a `struct` kulcsszóval vezethetünk be rekordokat. A fenti `struct month` rekord két mezője sztring, illetve `int` típusú, a mezők neve `name`, illetve `days`.

A rekord típus leggyakoribb implementációja, hogy az egyes mezők egymás után helyezkednek el a memóriában. Minden mező a rekord elejéhez képest saját távolsággal (offset) rendelkezik. Esetenként azonban a mezők között lehetnek „lyukak” (gap, padding) is, itt nem tárolunk információt. Lyukak amiatt lehetnek, mert egyes fordítók bizonyos típusokat csak adott (például néggyel osztható) bájtcímekre helyeznek el (gépi szó határaira igazítják). Ebből következően a rekord mérete nagyobb vagy egyenlő a mezők méreteinek összegével.

```
struct month { char short_name; char *name; int days; };
printf("%ld\n", sizeof(struct month));   /* nálam 24-et írt ki */
```

A rekord típussal rendszerint csak a legegyszerűbb műveleteket végezhetjük el, pl.

- az értékadást, ide értve az érték szerinti paraméterátadást és függvényvisszatérést is,
- a rekord címének lekérdezését, és
- az egyes mezők elérését.

Miután a rekord egy mezőjét elértük, az adott mező típusának megfelelő műveleteket végezhetjük el rajta.

C struct

```
struct month { char *name; int days; };
struct month jan = {"January", 31};

struct date { int year; struct month *month; char day; };
struct person { char *name; struct date birthdate; };

typedef struct person Person;

int main(){
   Person pete = {"Pete", {1970,&jan,28}};
   printf("%d\n", pete.birthdate.month->days);
   return 0;
}
```

Nézzünk egy picit összetettebb példát. A korábban már látott `struct month` után hozzuk létre a `struct date` és a `struct person` struktúrákat is. Az utóbbi tartalmaz mezőként egy `struct date` típusú értéket, mutatva, hogy az összetett típusok tetszőleges mélységben egymásba is ágyazhatók.

A `struct date` struktúrát lehetett volna három `int` típusú mezővel is definiálni – hiszen a rekordoknál az is megengedett, hogy a mezők ugyanolyan típusúak legyenek. Igazából koncepcionális kérdés az, hogy egy három `int`-ből álló adatot tömb vagy struktúra segítségével írunk le. Az előbbi elemeit indexeléssel, az utóbbi elemeit a mezőnevekkel érhetjük el. Egy dátum típus esetében a struktúra jobb megoldásnak tűnik.

Azt figyelhetjük meg a `struct date` struktúrában, hogy a második mező egy mutató egy `struct month` struktúrára. Amikor a `main`-ben létrehozzuk a `pete` változót, akkor az inicializáció során a `jan` változóra (struktúrára) mutató pointert használjuk a születési idő hónapjaként. A gyakorlatban sokszor előfordul, hogy egy struktúrát mutatókon keresztül érünk el. A mutató dereferálása és a struktúra egy mezőjének kiválasztása nagyon kényelmes a `->` operátor használatával.

```
struct month *next_month = &jan;
char        *name = (*next_month).name;
char *alternative =   next_month->name;
```

A kétfajta hivatkozás ekvivalens. Vegyük észre, hogy a `(*next_month).name` kifejezésből a zárójel nem hagyjató el!

A `typedef` konstrukcióval ugyanúgy új nevet vezethetünk be a struktúra típusokhoz, mint a felsorolási típusokhoz. Ezzel a trükkel ismét elérhetjük, hogy a típusunkhoz a `struct` kulcsszó használata nélkül is hozzáférjünk.

Paraméterátadás

```
void one_day_forward( struct date *d ){
    if( d->day < d->month->days ) ++(d->day);
    else { ... }
}

struct date next_day( struct date d ){
    one_day_forward(&d);
    return d;
```

```
}

int main(){
   struct date new_year = {2019, &jan, 1};
   struct date sober;
   sober = next_day(new_year);
   return ( sober.day != 2 );
}
```

Struktúrákat érték szerint adhatunk át paraméterként, és érték szerint kapjuk vissza őket visszatérési értékként. Ez nagyon más, mint a tömbök esetén. Egy tömb átadása a kezdőcímének (egy mutató) átadását jelenti: erre mondtuk azt, hogy *megosztás-szerinti paraméterátadás*. Tömböket visszaadni pedig nem is lehet a C-ben. (Mutatókat természetesen visszaadhatunk.)

A fenti `one_day_forward` függvény egy struktúrára mutató pointert vár paraméterként, ezzel szimuláljuk a cím-szerinti paraméterátadást. A függvény meg szeretné változtatni azt a dátum objektumot, amelynek címét paraméterként kapta. Ha nem mutatót adtunk volna át paraméterként, akkor a függvény az érték-szerinti paraméterátadás miatt csak egy lokális változót módosított volna, nem a hívás helyén elérhető dátum objektumot. Nyilván nem ez a cél ebben az esetben, hiszen itt egy be- és kimenő szemantikájú paraméterátadást akarunk megvalósítani: szeretnénk, ha a `one_day_forward` függvénnyel egy dátumot meg tudnánk változtatni. (Struktúrákat akkor is szoktunk mutatón keresztül átadni paraméterként, ha a hívott függvény nem akarja megváltoztatni a struktúrát. Akkor szoktunk ilyet csinálni, ha nagy méretű a struktúra, és el szeretnénk kerülni az érték-szerinti paraméterátadással járó másolást.)

A `next_day` függvény kap érték szerint paraméterként egy dátumot (azaz létrejön a függvényben egy lokális `struct date` változó, ami feltöltődik az aktuális paraméter értékével). A formális paraméternek megfelelő lokális változót megváltoztatjuk a `one_day_forward` függvény segítségével, majd a kapott struktúrát visszaadjuk. A kapott függvény használatát szemlélteni a `main`, amelyben a `sober` struktúrát a `next_day` függvény eredményével töltjük fel. A `new_year` struktúra eközben természetesen változatlan marad.

15.9 Unió típus

Unió típus

Típusértékei több típus valamelyikéből

C: union

```
struct       month { char *name; int days; }; /* name and days */
union name_or_days { char *name; int days; }; /* either of them */

union name_or_days brrr = {"Pete"};  /* now it contains a name */
printf("%s\n", brrr.name);   /* fine */
printf("%d\n", brrr.days);   /* prints rubbish */
brrr.days = 42;              /* now it contains a date */
printf("%d\n", brrr.days);   /* fine */
printf("%s\n", brrr.name);   /* probably segmentation fault */
```

Az unió típusok arra valók, hogy több típus típusértékhalmazát egy típusba egyesítsék. Sok procedurális, imperatív nyelvben megjelent ez a fajta típuskonstrukció. Van néhány szép felhasználási módja, de egy komoly problémát is felvet: tudjuk-e, hogy az aktuális időpillanatban milyen típusú értéket tárol a változónk?

A C nyelvben a `union` kulcsszóval vezetünk be unió típust. A fenti, már jól ismert `struct month` struktúra két mezőt tartalmaz: egy `name` *és* egy `days` mezőt. A `union name_or_days` típusú változók azonban a kétfajta érték közül egyszerre mindig csak egyet tartalmaznak! Azt, hogy éppen melyiket, a programnak kell valahogy nyomon követnie. Amikor létrehozzuk a `brrr` változót, egy `char *` értéket teszünk bele, egy karaktermutatót a `"Pete"` sztringre. Hivatkozhatunk a `struct`-oknál megismert szelektorral erre a mezőre – de hivatkozhatunk a másik mezőre is: a C nyelvben sem fordítás közben, sem futás közben

nem lehet kitalálni, milyen típusú értéket hordoz a változónk. A második `printf` ennek köszönhetően valami zagyvaságot ír ki. Meg is változtathatjuk a `brrr`-ben tárolt értéket, akár egy másik típusúra is. A harmadik kiírás rendben van, de a negyedik valószínűleg elszáll, mert a `42` számot próbáljuk karaktermutatóként értelmezni, és kiírni az „általa mutatott" sztringet.

Mi is történik? A `union` típusok belső ábrázolása olyan szokott lenni, hogy az összeuniózni kívánt típusok közül a legnagyobb helyigényűnek megfelelő tárhellyel dolgozik, és ezen a tárhelyen akármilyen értéket tárolhat az összeuniózott típusok akármelyikéből. Tehát a `name_or_days` típusú változóban vagy egy `char *` mutató, vagy egy `int` érték lesz; pontosabban az ott található értéket megpróbálhatjuk értelmezni így is, úgy is. A mi felelőségünk, hogy jól értelmezzük a unióban talált értéket.

Címkézett unió

```
enum shapes { CIRCLE, SQUARE, RECTANGLE };
struct    circle { double radius; };
struct    square { int side; };
struct rectangle { int a; int b; };

struct shape
{
    int x, y;
    enum shapes tag;
    union csr
    {
        struct    circle c;
        struct    square s;
        struct rectangle r;
    } variant;
};
```

A fent látható trükkel biztonságosabbá tehetjük az unió típus használatát. Körbevesszük az unió konstrukciót egy struktúrával, amelynek egyik mezőjét címkeként használjuk: ebből a címkéből fogjuk tudni, hogy milyen típusú értéket tárolunk egy változóban. Természetesen nagyon oda kell figyelnünk `struct shape` értékek konstruálásánál, hogy a címke összhangban legyen a `variant` tartalommal, és arra is oda kell figyelni, hogy nehogy később elrontsuk a tartalmat egy olyan értékadással, amely mondjuk a `variant` belső típusát megváltoztatja, de a címkét nem.

Vannak olyan programozási nyelvek, amelyek eleve csak a címkézett unió konstrukciót biztosítják a programozó számára. Az Ada nyelvben például nem lehet a belső reprezentációt elrontani: a nyelv futtató környezete ellenőrzi, hogy a változó típusú részeket a címkének megfelelően használjuk-e.

A Haskell nyelvben hasonlóképpen szigorú a címkézett unió konstrukció. Ebben a nyelvben az algebrai adattípusokkal fejezzük ki ugyanezt a gondolatot.

Egységes használat

```
struct shape
{
    int x, y;
    enum shapes tag;
    union csr
    {
        struct    circle c;
        struct    square s;
        struct rectangle r;
    } variant;
};

void move( struct shape *aShape, int dx, int dy ){
```

```
    aShape->x += dx;
    aShape->y += dy;
}
```

A `shape` címkézett unió típus lehetővé teszi azt, hogy egységesen használjuk a különböző típusú alakzatokat. Az eltolás műveletet általánosan megírhatjuk az összes alakzatra. Azt, hogy egy művelet többféle típusra is működik, *polimorfizmusnak* (többalakúságnak) nevezzük.

Használat esetszétválasztással

```
struct shape {
    int x, y;
    enum shapes tag;
    union csr {
        struct    circle c;
        struct    square s;
        struct rectangle r;
    } variant;
};

double leftmost( struct shape aShape ){
   switch( aShape.tag ){
       case CIRCLE: return aShape.x - aShape.variant.c.radius;
       default:     return aShape.x;
   }
}
```

Ha olyan műveletet készítünk, amely megvalósítása függ az alakzat milyenségétől, a művelet belsejében megvizsgálhatjuk a *tag*et. Így a polimorfizmus biztonságosan fenntartható.

Biztonságos létrehozás

```
struct shape {
    int x, y;
    enum shapes tag;
    union csr {
        struct    circle c;
        struct    square s;
        struct rectangle r;
    } variant;
};

struct shape make_circle( int cx, int cy, double radius ){
    struct shape c;
    c.x = cx; c.y = cy; c.tag = CIRCLE;
    c.variant.c.radius = radius;
    return c;
}
```

Bevezethetünk olyan függvényeket, amelyekkel a biztonságos létrehozás megkönnyíthető.

Ne felejtsük el azonban azt a tényt, hogy a struktúra belsejébe belelátunk, sőt, bármikor módosíthatjuk a belsejét. Ez a lehetőség továbbra is veszélyessé teszi a címkézett unió típusunk használatát. A belső ábrázolás elrejtése lenne a következő nagy feladat, de ez már egy nagy lépés lesz az objektum-orientált programozás irányába.

Az objektum-orientált progamozásban a fenti példához hasonló konstrukciók helyett az öröklődés mechanizmusát használjuk. Egy bázisosztály és a különböző leszármazottai segítségével a shape, a circle,

a square és a rectangle típusok nagyon kényelmesen (és ami szintén fontos: később bővíthető módon) megfogalmazhatók.

15.10 Osztályok

Osztály

- Objektum-orientált nyelvek
- Osztály: rekordszerű struktúra
  - Adattagok (mezők)
  - Műveletek (metódusok)
- Öröklődés: címkézett unió

Az osztályokkal a következő félévben fogunk megismerkedni. Az osztály olyan rekordszerű konsrukció, amelybe becsomagoljuk az adatokon kívül a rajtuk végezhető alapvető műveleteket is. Az objektumorientált paradigma egy fontos eleme az osztályok közötti öröklődés kapcsolat, amely a címkézett unió típus egyfajta megvalósulása. (Annyiban más a címkézett uniótól, hogy bővíthető újabb típusokkal, nincs „beégetve" a definícióba az összes típus, amit össze kívánunk uniózni.)

16 Láncolt adatszerkezetek

Adatszerkezetek

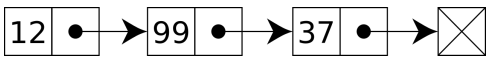
- „Sok" adat szervezése
- Hatékony elérés, manipulálás
- Alapvető módszerek
  - Tömb alapú ábrázolás (indexelés)
  - Láncolt adatszerkezet
  - Hasítás

Az adatszerkezetek célja, hogy jó sok elemi adatot eltároljunk bennük, és ezeket utána kényelmesen és hatékonyan elérhessük, feldolgozhassuk. Nagyon sok féle adatszerkezetet kitaláltak már, a félév során ebből a tárgyból is láttunk párat. Van néhány alapvető fogás abban, hogy hogyan valósíthatjuk meg ezeket az adatszerkezeteket. Van olyan ábrázolás, amely a tömbök már jól ismert tulajdonságára épít: a tömb memóriabeli kezdőcíméből hatékonyan kiszámítható bármelyik adott indexű tömbelem memóriabeli kezdőcíme (a tömbben tárolt elemek méretét felhasználva). Nem csak sorozatot, de például gráfot, fát (kupacot) is lehet tömb segítségével ábrázolni.

Mi most egy másfajta ábrázolással fogunk megismerkedni: a láncolt ábrázolással. Ennek lényegi eleme az, hogy az adatelemekre mutatókkal tudunk majd hivatkozni.

Láncolt adatszerkezetek

- Sorozat: láncolt lista
- Fa, pl. keresőfák
- Gráf



A láncolás során az adatokat olyan „csomópontokban" helyezzük el, amelyeket mutatókkal kötünk össze. Ilyen technikával sorozatok könnyen leírhatók, ahogy az ábra mutatja. Ha egy csomópontban több mutató is van, akkor elágazó struktúrák, például fák is ábrázolhatók. Körkörös hivatkozásokkal akár tetszőleges gráfokat is felépíthetünk.

Sorozatok ábrázolása

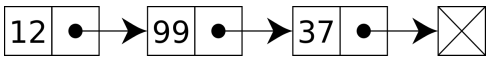
Tömb

- Akárhányadik elem előkeresése, felülírása
- Beszúrás/törlés?
  - Adatmozgatás
  - Újraallokálás

(egy példa: http://gsd.web.elte.hu/lectures/imper/imper-lecture-10/)

Láncolt lista

- Elemek előkeresése és felülírása bejárással
- Beszúrás/törlés bejárás során
- Akárhányadik elem előkeresése, felülírása?



Sorozatok ábrázolására két fő módszert szoktak használni: vagy egy tömbben helyezik el a sorozat elemeit, vagy egy láncolt listát építenek belőlük. A kétfajta reprezentációval más-más műveletek végezhetők el hatékonyan, ezért másfajta algoritmusok során szoktuk használni őket.

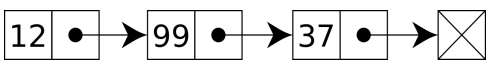
A tömb alapú reprezentáció az indexelésnek köszönhetően nagyon hatékony az elemek előkeresésében. Az adatszerkezetben tárol elemek számától függetlenül (úgy is mondjuk, hogy *konstans időben*), néhány aritmetikai művelettel meghatározható egy adott indexű elem memóriabeli helye. Így az adatelemek kiolvasása vagy felülírása könnyen elvégezhető.

A tömbök akkor használhatók jól, ha előre tudjuk, hogy hány elemet kívánunk használni az adatszerkezetben, vagy legalábbis tudunk egy jó felső korlátot mondani az elemek számára. Ha ugyanis „betelik" a tömb, a kibővítés elég költséges. Ilyen esetben új tömböt kell allokálni, és a régiből átmásolni az adatokat az újba. Ennek költsége a tömbben tárolt elemek számával egyenesen arányos (úgy is mondjuk, hogy *lineáris időben* végezhető el a művelet). A törlés a sorozatból szintén költséges. Ha nem szeretnénk lyukakat a reprezentációban, akkor egy törlés során a törölt elem után álló összes elemet eggyel előre kell mozgatni, ami ismét lineáris idejű művelet.

Ha olyan algoritmust készítünk, amelyben nagyon gyakran kell elemeket beszúrni a sorozatba, vagy törölni a sorozatból, meggondolandó, hogy nem hatékonyabb-e egy láncolt listás ábrázolás. A láncolt lista egy tetszőleges elemének előkeresése egy időigényes művelet, mert végig kell lépkedni a listán, amíg meg nem találjuk. Ha viszont az algoritmusban erre nincs szükség, akkor a láncolt lista nagyon jól használható. Az elemek végigjárása (azaz a lista bejárása) hatékony: minden elem rákövetkezőjét konstans időben megkaphatjuk. Bejárás során a törlés és a beszúrás is konstans idejű.

Láncolt lista

```
struct node
{
    int data;
    struct node *next;
};
```



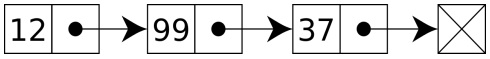
A C-ben a láncolt adatszerkezetek sokkal alacsonyabb szinten programozandók, mint mondjuk Haskellben. A láncolt lista nem egy nyelvbe beépített adatszerkezet, hanem mi magunk kell megvalósítsuk. Az ábrázoláshoz egy struktúrát érdemes használni. A struktúra egy csomópontot fog leírni a listában. Az egyik mezője a csomópontban tárolt adatelem (itt most `data`), a másik mezője pedig a lista következő csomópontjára mutat (`next`). A láncolást tehát mutatók segítségével fejezhetjük ki.

Most nézzük meg, hogyan építhető fel a fenti három elemű lista.

Láncolt lista felépítése

```
struct node
{
    int data;
    struct node *next;
};

struct node *head;
head = (struct node *)malloc(sizeof(struct node));
head->data = 12;
head->next = (struct node *)malloc(sizeof(struct node));
head->next->data = 99;
head->next->next = (struct node *)malloc(sizeof(struct node));
head->next->next->data = 37;
head->next->next->next = NULL;
```



A lista csomópontjait dinamikusan hozzuk létre. Az allokált csomópontokat adatokkal töltjük fel, valamint a láncolást is elvégezzük a mutatók beállításával. Természetesen egy lista feltöltését általában egy ciklussal programozzuk le.

17 Egyenlőségvizsgálat és másolás

Egyenlőségvizsgálat és másolás elemi típusokon

```
int a = 5;
int b = 7;

if( a != b )
{
    a = b;
}
```

Mindenki kapásból megérti a fenti példakódot. Összehasonlítjuk a két `int` típusú változót, és ha nem egyenlőek, akkor az egyiknek értékül adjuk a másikat. Egyszerű. De mi történik, ha nem `int` típusú változókkal, hanem valami bonyolultabbal dolgozunk?

Mutatókkal?

```
int n = 4;
int *a = (int*)malloc(sizeof(int));
int *b = &n;

if( a != b )
{
    a = b;
}
```

A két mutató akkor egyenlő, ha ugyanoda mutatnak. Az értékadás hatására az `a` ugyanoda fog mutatni, mint a `b`, tehát `a` egyenlő lesz `b`-vel, és a `*a` ugyanaz lesz, mint a `*b`. Ezt neveztük aliasingnak.

Tömbökkel?

```
int a[] = {5,2};
int b[] = {5,2};

if( a != b )
{
    a = b;          /* fordítási hiba */
}
```

Ha ugyanezt a kódrészletet tömbökkel vizsgáljuk, azt látjuk, hogy az értékadás nem is lehetséges. Ezt már korábban is megállapítottuk: a C-ben tömböknek nem lehet új értéket adni. Az egyenlőségvizsgálat értelmezett, mégpedig úgy, hogy a tömbök automatikusan mutatóvá alakulnak, így két fizikailag különböző tömb nem lehet egyenlő egymással. (Az *egyenlőség* valójában az *azonosság*.)

Tömbökkel!

```
#define SIZE 3

int is_equal( int a[], int b[] ){
    for( int i=0; i<SIZE; ++i )
        if( a[i] != b[i] ) return 0;
    return 1;
}

void copy( int a[], int b[] ){
    for( int i=0; i<SIZE; ++i ) a[i] = b[i];
}

int a[SIZE] = {5,2}, b[SIZE] = {7,3,0};

...
if( ! is_equal(a,b) ) copy( a, b );
```

Ha tartalmi egyenlőségvizsgálatot szeretnénk a tömbjeinkhez, akkor érdemes lehet egy műveletet definiálni erre a célra. Ha fix méretű tömbjeink vannak (pl. `SIZE`), akkor ez elég könnyű, egyébként viszont külön kell vizsgálni, hogy a tömbök mérete megegyezik-e, és persze a közös méretet át is kell adni az egyenlőséget vizsgáló függvénynek. (Ne feledjük: nem a tömb, hanem a legelső elemére mutató mutató adódik át paraméterként!)

Mivel helyettesíthetjük az értékadást? Ha az a célunk, hogy a két tömb tartalmilag, elemről elemre megegyezzen, akkor írhatunk egy olyan másoló műveletet, ami az egyikből átmásolja a másikba az adatokat. Egyszerűbben is megúszhatjuk a dolgot: használhatjuk a `string` könyvtárban definiált `memcpy` függvényt is erre a célra.

```
void *memcpy( void *dest, const void *src, size_t numbytes );
```

A típusokból látható, hogy ezt akármilyen adatstruktúra másolására használhatjuk: a `dest` memóriacímtől kezdődően tárolja el azt a `numbytes` darab bájtot, amit az `src` memóriacímtől kezdődően kiolvas. Fontos megkötés erre a függvényre, hogy a két tárterület nem fedhet át egymással! A fenti `a` és `b` esetében ez teljesül, de amúgy nem feltétlen könnyű tetszőleges mutatóknál ezt garantálni…

```
#include <string.h>
#define MIN(a,b) ((a)>(b) ? (b) : (a))

int a[] = {5,2};
int b[] = {7,3,4};

memcpy( a, b, MIN(sizeof(a), sizeof(b)) );
```

Struktúrákkal?

```
struct pair { int x, y; };

struct pair a, b;
a.x = a.y = 1;
b.x = b.y = 2;

if( a != b )          /* fordítási hiba */
{
    a = b;
}
```

Térjünk át most struktúrákra. Rájuk meg az összehasonlítás tilos, arra kapunk fordítási hibát. Az értékadás értelmezett, és lényegében pont ugyanaz történik, mint elemi adatok esetén: a `b` változóban tárolt összetett érték átmásolódik az `a` változóba, így a két rekord mezői páronként egyenlően lesznek az értékadás után.

Struktúrákkal!

```
struct pair { int x, y; };

int is_equal( struct pair a, struct pair b )
{
    return (a.x == b.x) && (a.y == b.y);
}

struct pair a, b;
a.x = a.y = 1;
b.x = b.y = 2;

if( is_equal(a,b) )
{
    a = b;
}
```

A megoldás itt is az lehet, hogy írunk egy műveletet, amely a hiányzó nyelvi szolgáltatást megvalósítja, azaz egy olyan műveletet, amely a két struktúrát mezőről mezőre összehasonlítja.

A másoláshoz nem feltétlenül kell műveletet írni, hiszen az értékadás gondoskodik a tartalom átmásolásáról. Ha mégis külön műveletet szeretnénk írni, akkor nem ez a jó megoldás.

```
void copy( struct pair a, struct pair b )
{
    a = b;
}
...
copy(a,b);
```

Az érték szerinti paraméterátadás miatt egy ilyen műveletnek nem lenne semmiféle hatása a hívási környezetben. Helyette természetesen ezt írnánk.

```
void copy( struct pair *a, struct pair *b )
{
    *a = *b;
}
...
copy(&a,&b);
```

Láncolt lista?

```
struct node
{
    int data;
    struct node *next;
};

struct node *a, *b;
...

if( a != b )
{
    a = b;
}
```

Nézzük meg, mint mondhatunk egy láncolt listával megvalósított sorozat egyenlőségvizsgálatáról, illetve másolásáról. Az eddigiek alapján nyilvánvaló, hogy a fenti kód – bár teljesen helyes, értelmes – nem azt vizsgálja, hogy két sorozat tartalmilag megegyezik-e, illetve nem tartalmilag másolja az egyik sorozatot a másikba. Az `a == b` kifejezés az `a` és `b` mutatók egyenlőségét vizsgálja, azaz pontosan akkor igaz, ha `a` és `b` egymás aliasai (azaz ugyanarra a listára hivatkoznak). Az `a = b` értékadás hatása pedig az, hogy az `a` mutató ugyanarra a listára fog hivatkozni, mint a `b` mutató.

Sekély megoldás – nem jó ide

```
struct node
{
    int data;
    struct node *next;
};

int is_equal( struct node *a, struct node *b )
{
    return (a->data == b->data) && (a->next == b->next);
}

void copy( struct node *a, const struct node *b )
{
    *a = *b;
}
```

Ahogy a `struct pair` kapcsán megtettük, megírhatjuk a tartalmi egyenlőségvizsgálatot és másolást a `struct node` típusra is. Értelmezzük a kapott megoldást: két mutató, amelyek `struct node` típusú értékre mutatnak, nem csak akkor lesznek egyenlőek, ha ugyanoda mutatnak (azaz ugyanazt a listát hivatkozzák), hanem akkor is, ha nem azonos, de egyenlő struktúrákra mutatnak. Mit is jelent itt az egyenlőség? Azt, hogy mezőnként egyenlő a két mutatott struktúra: a két listában a legelső elemek megegyeznek, valamint a két listában ugyanaz a struktúra lesz az első csúcspont rákövetkezője. Valóban ezt akartuk? Nem! Még mindig túl szigorú feltétellel próbálunk dolgozni! Azt szeretnénk, ha azon túl, hogy a két listában a legelső elemek megegyeznek, a második csúcsponttól kezdődően ismét *egyenlőség* (és nem *azonosság*) állna fenn! Ezt a megközelítést *mély egyenlőségvizsgálatnak* (angolul *deep equality*) nevezzük, szemben azzal, amit ezen a dián látunk (melynek neve *sekély egyenlőségvizsgálat*, azaz *shallow equality*).

Ugyanilyen problémát figyelhetünk meg a `copy` művelet kapcsán is. A fenti megoldás egy *sekély másolást* (*shallow copy*) végez: belemásolja az `a` mutató által hivatkozott lista legelső csúcspontjába a `b` mutató által hivatkozott lista legelső csúcspontját, aminek az lesz a következménye, hogy a két lista valójában nem két független lista lesz, hanem a második csúcsponttól kezdve egybeesnek. A másolás esetében is a *mély másolás* (*deep copy*) lesz itt a jó megoldás.

Mély egyenlőségvizsgálat

```
struct node
{
    int data;
    struct node *next;
};

int is_equal( struct node *a, struct node *b )
{
    if( a == b ) return 1;
    if( (NULL == a) || (NULL == b) ) return 0;
    if( a->data != b->data ) return 0;
    return is_equal(a->next, b->next);
}
```

Az egyenlőségvizsgálatot könnyen megfogalmazhatjuk rekurzióval és esetszétválasztással. A fenti `is_equal` definíció ugyan rekurzív, de – mivel a rekurzív hívás a definíció legutolsó utasítása, azaz a definíció *végrekurzív* (*tail-recursive*), a fordító könnyen optimalizálja ciklussá.

Mély másolás

```
struct node {
    int data;
    struct node *next;
};

struct node *copy( const struct node *b ){
    if( NULL == b ) return NULL;
    struct node *a = (struct node*)malloc(sizeof(struct node));
    if( NULL != a ){
        a->data = b->data;
        a->next = copy(b->next);
    } /* else hibajelzés! */
    return a;
}
```

A mély másolást is könnyű rekurzívan definiálni. Hosszabb listák esetén azonban ez okozhat problémákat (megtelik a végrehajtási verem, vagy egyszerűen csak nagyon lassú lesz az implementáció). Ezért érdemes lehet ciklusra átírni.

```
struct node *copy( const struct node *b ){
    if( NULL == b ) return NULL;
    struct node *a = (struct node*)malloc(sizeof(struct node));
    if( NULL != a ){
        a->data = b->data;
        struct node *p = a;
        b = b->next;
        while( NULL != b )
        {
            p->next = (struct node*)malloc(sizeof(struct node));
            if( NULL != p->next )
            {
                p->next->data = b->data;
                p = p->next;
                b = b->next;
            } /* else hibajelzés! */
        }
    } /* else hibajelzés! */
```

```
    return a;
}
```

18 Hibakezelés

Hibakezelés

- Ha valami nem várt történik
  - Például sikertelen `malloc`
- Robusztusság
- Nyelvi támogatás?

Egy programot robusztusnak nevezünk, ha fel van készülve futás közbeni hibákra, hibás bemeneti adatokra stb. A professzionális szoftverfejlesztésben ez egy megkerülhetetlen minőségi elvárás. Egy programot helyesnek nevezünk, ha helyes inputokra helyes outputot ad: ha a program környezete az általunk elvárt módon működik, akkor a helyes program helyes eredményt szolgáltat. A robusztusság ellenben azt követeli meg, hogy nem várt környezeti viselkedésre is értelmesen reagáljon a szoftver.

Ha bármilyen problémát detektálunk a program végrehajtása alatt, meg kell próbálni a program értelmes működését fenntartani. Ez lehet egy jó, hasznos hibajelzés a program leállítása előtt, de szerencsés esetben a program tovább tud működni a hiba bekövetkezte ellenére. Például, ha a programunk kliensek kéréseit szolgálja ki, akkor egy adott kliens kérésének teljesítése meghiúsulhat egy hiba miatt, de ettől a többi kliens kérését továbbra is ki tudja szolgálni a programunk.

Az eddigiek során azt láttuk, hogy a dinamikus memória allokálása lehet sikertelen is, és ezt a `malloc` által visszaadott érték vizsgálatával ellenőrizhetjük. Ha `NULL` értéket ad vissza ez a függvény, tudjuk, hogy baj van.

Hibakezelés C-ben

- Függvény visszatérési értéke
  - Hibakód visszaadása (`int`)
  - Speciális („extremális“) érték visszaadása (pl. `NULL`)
- Globális változó: hibakód

A C nyelvben sokszor úgy oldják meg a hibakezelést, hogy egy függvény visszatérési értékébe belekódolják a hibalehetőségek jelzését is. Történhet ez úgy, ahogy a `main` esetében: egy hibakód visszaadásával, vagy történhet a `malloc` stílusában, egy speciális érték visszaadásával.

- A `main` egy egész számot ad vissza. Ha sikeresen futott le programunk, a `0` értéket adjuk vissza, a hibákat pozitív értékekkel jelezzük. Az operációs rendszer felkészülhet a program által visszaadott érték vizsgálatára, a hibák kezelésére. Programunkon belül is használhatjuk ezt a megoldást. Ha egyébként egy `void` függvényt definiálhatnánk, alakítsuk át `int` visszatérési értékűvé, és használjuk a visszatérési értéket a sikeresség, illetve az esetleges hibák jelzésére!
- Egy nem `void` típusú függvénynél a visszaadott érték lehetséges értékkészletét egészítsük ki speciális, hibát jelző értékekkel. Ezek az úgynevezett extremális értékek megkülönböztethetők a normális visszatérési értéktől, így a hívó kódrészlet el tudja dönteni, sikeres volt-e a hívás. Ezt a működési elvet használja a `malloc` is.

Egy másik megoldás az szokott lenni, hogy egy globális (például `int` típusú) változóban tároljuk a legutóbbi művelet sikerességének vagy sikertelenségének megfelelő értéket. Ilyen lehet a közismert `errno`. Egy statikusan tárolt globális változó persze konkurens programozási környezetben nem jól használható, ezért a C újabb szabványaiban felturbózták: C11 felett a változó minden egyes folyamatban külön példányban létrejön.

C hibakezelés hátulütői

- Túl sok elágazás, feltételvizsgálat

  - A hibakezelés akár a kód 30-40%-a is lehet
  - Elvész a kódban a lényeg
- Kispórolt hibakezelés veszélye
- Elfelejtett hibakezelés veszélye

Egy olyan régimódi nyelvben, mint a C, elég nehézkes és kényelmetlen a programfutás közben fellépő hibákat kezelni. A sok extra kód, amit a hibakezelés miatt írni kell, zavarossá, nehezen érthetővé, nehezen karbantarthatóvá teszi a kódbázist. Emiatt a programozók hajlamosak lehetnek kispórolni a hibák kezelését, ami persze komoly veszélyekkel jár a szoftver robusztusságát illetően. Hasonló veszélyeket rejt az, ha egy-egy esetet elfelejt lekezelni a programozó.

A modernebb nyelvekben általában szokott már lenni nyelvi támogatás a hibakezelésre.

Hibakezelés modern nyelvekben

Nyelvi támogatás!

Kivételek

- Kivétel kiváltódása és kiváltása
- Kivétel terjedése
- Kivétel lekezelése

A program futása során észlelt hibákat a modern nyelvek úgynevezett kivételek formájában ábrázolják. A kivételek terjedéséhez és lekezeléséhez, sőt, akár programozó által történő kiváltásához is programnyelvi támogatást nyújtanak.

A kivételkezelés célja, hogy a program a detektált probléma ellenére is tovább tudjon működni, vagy ha ez nem lehetséges, a kivétel továbbterjedése előtt az adott programkomponens a szükséges *clean-up* tevékenységeket elvégezze.

Kivétel terjedése

Vezérlésátadás!

- Alprogramok hívási lánca mentén
- Végrehajtási verem
- Fellépéstől…
  - … lekezelésig
  - … programleállásig

A kivételek kiváltódás (vagy kiváltás) után terjedni kezdenek, amíg egy kivételkezelő utasítás le nem tudja őket kezelni. Így a kivételek terjedése egy vezérlésátadási mechanizmusnak fogható fel: a program egy bizonyos pontján fellép a kivétel, és onnan átadódik a vezérlés a program egy másik (jellemzően távoli) pontjára, a kivételkezelő utasításokra.

Nem igazán lehet a kivételek terjedési mechanizmusát a strukturált programozással összeegyeztetni: az egésznek épp az a lényege, hogy a kivétel kiugrasztja a programot a hibamentes viselkedést leíró struktúrából. A struktúra itt a program szintaktikus szerkezetét, azaz a statikus felépítését jelenti.

Másrészt azonban a kivétel a program dinamikus szerkezetét (az alprogramhívások logikáját) követve terjed. Ha egy függvényben fellép egy kivétel, akkor az a függvényt hívó utasításhoz terjed, ami általában egy másik függvény. Ha ott sem kerül lekezelésre, akkor továbbterjed az azt hívó utasításhoz stb. Tehát a terjedés az alprogramok hívási lánca mellett terjed visszafelé, a főprogram irányába. Ha sehol sem találkozik kivételkezelő utasítással, végső soron a főprogram leállítását okozza. (Többszálú programozást támogató nyelvek esetén a kivétel a kiváltódást kezdeményező esemény végrehajtását végző szál leállását eredményezheti.)

Azt mondhatjuk tehát, hogy a kivételek terjedése a végrehajtási veremben található információk alapján történik. Egy le nem lezelt kivétel a hívási láncon visszafelé haladva dobálja ki az aktivációs rekordokat

a veremből. Ezek a kidobált aktivációs rekordok írák le a kivétel fellépésének körülményeit (a kivétel fellépéséhez vezető alprogramhívások sorozatát) – ezt szokás *stack trace*-nek nevezni.