Homepage of Dr. Zoltán Porkoláb

Home

Archive

Teaching

Timetable

Bolyai College

C++ (for mathematicians)

Imperative programming (BSc)

Multiparadigm programming (MSc)

Programming (MSc Aut. Sys.)

Programming languages (PhD)

Software technology lab

Theses proposals (BSc and MSc)

Research

CodeChecker

CodeCompass

Templight

Projects

Conferences

Publications

PhD students

Affiliations

Dept. of Programming Languages and Compilers

Ericsson Hungary Ltd

Imperatív programozás 1.

Programozási alapfogalmak

Szintaxis (syntax)

A programozási nyelv helyes *nyelvtana*. Például: mik a helyes kulcsszavak, hogyan nézhet ki egy változó neve, hova tegyünk pontosvesszőt, stb.

A *PL/I* programozási nyelv pl. legenerálta a hiányzó **end** utasításokat. Az *Algol68* nyelvben az üres utasítás csak a **skip** utasítással valósult meg. A *Pascal*-ban **else** előtt sohasem szabad **;**-nek szerepelnie, a C-ben ez előfordulhat. A *Go*-ban kötelező kiírni a nyitó-csukó kapcsoszárójeleket a vezérlési szerkezetek esetén, így nem következhet be az ún. goto-fail. A legtöbb programozási nyelvben a whitespace-ek közönbösek, de pl. *Python*-ban a tabulálás (is) azonosítja a programszerkezetet. Az *APL*-ben mind a 256 karakter egy-egy érvényes operátor.

Szemantika (semantics)

A szintaktikailag helyes programok *jelentése*. Például: milyen típusrendszere van a nyelvnek, milyen konverziók történnek, melyik függvényt hívjuk meg egy híváskor, stb.

A C++-ban az **int** és **bool** típusok között automatkus oda-vissza konverzió van. Más nyelvekben szigorúbb a típusosság. Egyes objektum-orientált nyelvekben a meghívott virtuális függvény kiválasztása az objektum dinamikus típusától függ (*dynamic dispatch*). Nagyon ritkán ez *több* objektumtól is függhet (*multiple dispatch*).

A szemantikát különböző nyelvekben eltérően definiálják. Pl. a *Haskell* szemantikáját matematikai formulákkal (*denotational semantics*) adják meg. Az *Algol68* esetében logikai kifejezéseket használtak (*operational semantics*), de van példa axiomatikus és szöveges megadásra is.

Egyes esetekben a szemantika negyon bonyolult tud lenni. Mit jelent C++-ban a *protected abstract virtual base pure virtual private destructor* és mikor volt rá utoljára szükséged? (Tom Cargill, 2009). (Egyébként itt egy példa, hogy mit jelent).

Pragmatika (pragmatics)

A nyelv konstrukcióinak *haszálata*. Pl: hogyan használjuk az egyes nyelvi konstrukciókat a céljaink eléréséért, hogyan alakult a története, milyen fejlődési irányok léteznek.

A legtöbb programozási nyelvben ugyanazt a feladatot sokféleképpen is elvégezhetjük, pl. ciklust *while*, *for*, *range-for* vagy *do-while* segítségével is. Melyik az adott esetben a legjobb konstrukció? Melyik a legolvashatóbb, legkarbantarthatóbb, leghatékonyabb? Melyik a legkönnyebben debuggálhatóbb?

Az informatika fejlődik, folyamatosan újabb és újabb konstrukciók jelennek meg. De ugyanígy fejlődnek az egyes programozási nyelvek is! A C 1971-es, a C++ 1980-tól fejlődik. A C **malloc**/**free** hívását C++-ban felváltotta a **new**/**delete**, ami szükség esetén már végrehajtotta a *konstruktor*-t és *destruktor*-t is. Majd először külső könyvtári alapon, C++11-től a nyelvi szabványban is megjelentek a *smart pointer* osztályok, akik levették a teher nagy részét a programozóról.

A Python programozási nyelv gyökeres változásokon esett át a 2-es és a 3-as verzió között. Vannak programok, amik hibásak, vagy (még rosszabb) nem hibásak, de másképpen működnek az új verzióban.

Programozási paradigmák

Ahhoz, hogy bonyolult informatikai feladatokat megoldjuk, azokat kisebb részekre bontjuk, amíg elég kicsik ahhoz, hogy vagy létezik már megoldás rá, vagy hatékonyan megoldjuk magunk. Az, hogy milyen *elvek* szerint végezzük ezt a felbontást, azt a *programozási paradigma* határozza meg.

Imperatív programozás

Akkor beszélünk imperatív programokról, amikor explicit mi vezéreljük, hogy a program hogyan változtatja meg az állapotát.

Procedurális programozás

A feladatot például felbonthatjuk az elvégzendő feladatok (algoritmusok) szerint. Ezeket *alprogramokként* (függvények, eljárások) valósítjuk meg, köztük pl. paraméterátadással, függvény visszatérő értékkel 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 objetkumelvű (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 szintén kategorizálhatunk:

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. Pl. Lisp, ML, Haskell, Clean.

Logikai programozás

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

Multiparadigma programozás

Természetesen a fentiek némiképp szubjektív kategóriák, ezért más paradigmákról is szoktak beszélni, pl. *matematikai programozás*, *generikus (generic) programozás*, *szándékalapú (intentional) programozás*, stb.

Másrészről az egyes programozási nyelvek is több paradigmára épülnek. Egy objektumelvű programban is procedurálisan implementáljuk a metódusokat. A C++ programozási nyelvben hozhatunk létre osztályokat, de ez nem kötelező, mint pl. a Java-ban (az utóbbira azt mondjuk: tisztán objektumelvű). Emellett alkalmazhatunk funkcionális elemeket (pl. *lambda* objektumokat) és speciális *template* konstrukciókat (generikus programozás). A C++ *multiparadigma* programozási nyelv.

A C programozási nyelv

A procedurális programozás alapjait a C nyelv segítségével fogjuk bemutatni. Miért?

A nyelvek népszerűsége a TIOBE index szerint 2018 szeptemberében:

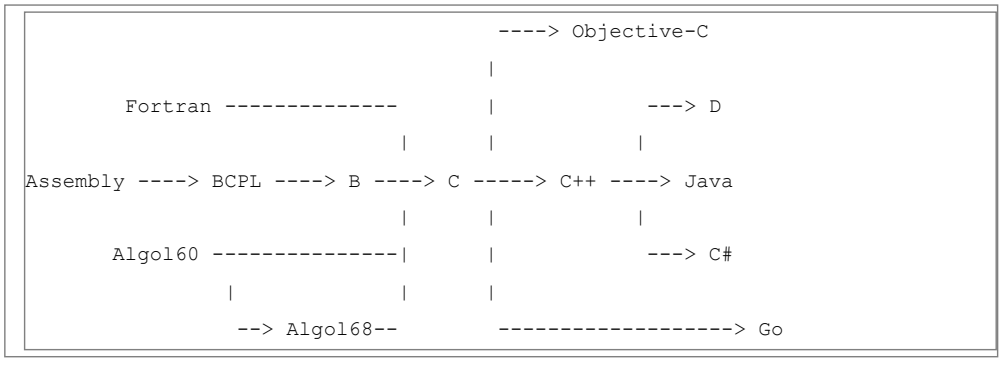
Sep 2018	Sep 2017	Change	Programming Language	Ratings	Change
1	1		Java	17.436%	+4.75%
2	2		C	15.447%	+8.06%
3	5	^	Python	7.653%	+4.67%
4	3	v	C++	7.394%	+1.83%
5	8	^	Visual Basic .NET	5.308%	+3.33%
6	4	v	C#	3.295%	-1.48%
7	6	v	PHP	2.775%	+0.57%
8	7	v	JavaScript	2.131%	+0.11%
9	-	^	SQL	2.062%	+2.06%
10	18	^	Objective-C	1.509%	+0.00%

A C nyelv és története

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 helye a programozási nyelvek között:



Programozási nyelvek hatása

A C idővonal

- 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
- 1988 Brian Kernighan és Dennis Ritchie: The C Programming Language 2nd ed, az ANSI C leírása
- 1989 ANSI C standard (C90) (32 kulcsszó)
- 1999 ANSI C99 standard (+5 kulcsszó)
- 2011 ANSI C11 standard (+7 kulcsszó)

Mi döntően az 1989-es ANSI C-t fogjuk használni.

Fordítás, szerkesztés, végrehajtás

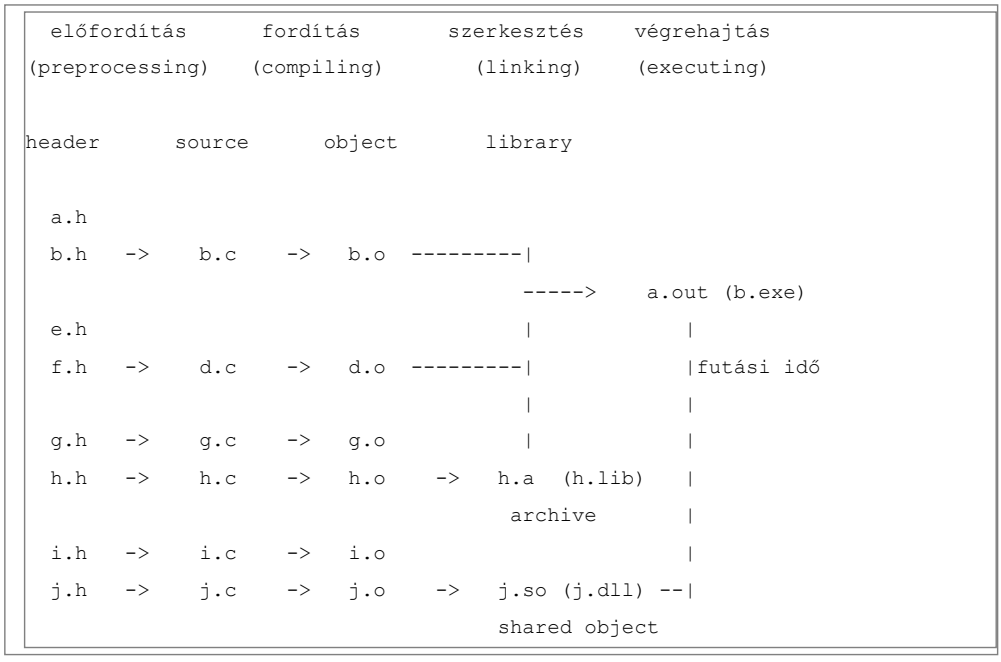
Fordítás vagy intrepretálás

A programozási nyelvek egy jó részét a **fordítóprogram** (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.

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.



Más programozási nyelveket **interpreter** hajt végre. Az interpreter egy önálló program, ami olvassa a végrehajtandó program forrását és lépésenként végrehajtja.

Az ilyen nyelvek sokkal rugalmasabbak (akár önmódosíthatóak) is lehetnek. Gyakran az interpretált nyelvek elő-ellenőrzést vagy előfordítást is alkalmaznak.

A Java nyelv *közbülső* kódra (Java bytecode) fordít, amit a Java Virtuális Gép (JVM, Java Virtual Machine) lényegében intrepreterként hajt végre. Valójában számos hibrid megoldást alkalmaznak, pl. Just In Time compiler-t (JIT), ami a sűrűn végrehajtott bájtkódot gépi kódra fordítja, így azokat sokkal gyorsabban tudja végrehajtani.

Az első C program: hello world

```
$ cat hello.c
```

```
1       #include <stdio.h>
2       int main()
3       {
4         printf( "hello world\n" );
5         return 0;
6       }
```

```
# compile + link
$ gcc  hello.c

# execute
$ ./a.out

# compile + link + set warnings on
$ gcc -ansi -pedantic -Wall -W hello.c

# c11 mode
$ gcc -std=c11 -ansi -pedantic -Wall -W hello.c

# set output name to a.exe
$ gcc -std=c11 -ansi -pedantic -Wall -W hello.c -o a.exe
```

Mindezt külön lépésekben is elvégezhetjük:

```
# compile only
$ gcc -c  hello.c
$ ls
hello.o

# will call the linker
$ gcc hello.o

# calls the compiler for all sources then calls the linker
$ gcc  a.c  b.c  d.o  e.a  f.so
```

Fordítási hibák, figyelmeztetések (warning-ok)

Ha szinaktikus hibát vétünk, a fordító hibaüzenetet ad, nem készül el a tárgykód, a linkelési lépésre nem kerül sor.

```
1       /*
```

```
 2        *  BAD VERSION !!!
 3        *  Missing semicolon
 4        */
 5       #include <stdio.h>
 6       int main()
 7       {
 8         printf("hello world\n") // missing ;
 9         return 0;
10       }
```

```
$ gcc -ansi -pedantic -W -Wall m.c
m.c: In function ‘main’:
m.c:6:28: error: expected expression before ‘/’ token
   printf("hello world\n") // missing ;
                           ^
```

Ha hibát vétünk, de a fordító még képes a forráskódot lefordítani (de elég gyanús az eredmény), a fordító figyelmeztetést (warning-ot) ad:

```
 1         /*
 2          *  BAD VERSION !!!
 3          *  Missing header
 4          */
 5         // #include <stdio.h>
 6         int main()
 7         {
 8           printf("hello world\n");
 9           return 0;
10         }
```

```
$ gcc -ansi -pedantic -W -Wall -std=c11 hello2.c -c
hello2.c: In function ‘main’:
hello2.c:6:3: warning: implicit declaration of function ‘printf’ [-
Wimplicit-function-declaration]
   printf("hello world\n");
   ^
hello2.c:6:3: warning: incompatible implicit declaration of built-in
function ‘printf’
```

A figyelmeztetések komoly dolgok a C-ben, úgy kell kezelnünk őket, mint a fordító által adott szintaktikus hibákat. Csak nagyon kivételes esetekben (és csak amikor teljesen biztosak vagyunk magunkban) szabad őket figyelmen kívül hagyni. A helyes szokás *warning-free* kódot írni.

Előfordulhat hiba a szerkesztési fázisban is. Ha pl. egy olyan függvényt próbálunk meghívni, amit egyetlen összeszerkesztendő állományban sincsen, vagy éppenséggel egynél többször szerepel azokban, szerkesztési hibát kapunk.

Ajánlott feladatok:

1. Hozzon létre egy programot, ami kiírja a nevét. Fordítsa le, szerkessze, futtassa.

2. Vágja ketté az előző programot két forrásfájlra. Az egyik visszaadja a nevét, a másik kiírja. Tipp: a nevet visszaadó függvény szignatúrája legyen **char *my_name(void)**. A printf-ben használja a kiíráshoz a **%s** formátumot.

Homepage of Dr. Zoltán Porkoláb

Home

Archive

Teaching

Timetable

Bolyai College

C++ (for mathematicians)

Imperative programming (BSc)

Multiparadigm programming (MSc)

Programming (MSc Aut. Sys.)

Programming languages (PhD)

Software technology lab

Theses proposals (BSc and MSc)

Research

CodeChecker

CodeCompass

Templight

Projects

Conferences

Publications

PhD students

Affiliations

Dept. of Programming Languages and Compilers

Ericsson Hungary Ltd

Imperatív programozás 2.

Statikus típusrendszer

Statikus vagy dinamikus típusrendszer

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 típusrendszer**-rel 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 ellenőrheti 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.

A második C program: Fahrenheit - Celsius konverzió

(avagy *A jó, a rossz és a csúf* imdb)

A feladat -100 és +400 közötti Fahrenheit értékek Celsius megfelelőinek kiírása 100-as lépésközzel.

```c
/*
 *  BAD VERSION !!!
 *  Convert Fahrenheit to Celsius
 *  between -100F and +400F  by 100F
 */
#include <stdio.h>
int main()
{
  int fahr;

  for ( fahr = -100; fahr <= 400; fahr += 100 )
  {
    printf("Fahr = %d,\tCels = %d\n",fahr,5/9*(fahr-32));
  }
  return 0;
}
```

```
$ gcc -ansi -pedantic -W -Wall -std=c11 fahrenheit.c -o fahrenheit

$ ./fahrenheit
Fahr = -100,    Cels = 0
```

```
Fahr = 0,       Cels = 0
Fahr = 100,     Cels = 0
Fahr = 200,     Cels = 0
Fahr = 300,     Cels = 0
Fahr = 400,     Cels = 0
```

A hiba oka, hogy a statikus típusrendszerben a fordító fordítási időben eldönti, hogy mi az 5/9 részkifejezés típusa. Mivel 5 és 9 típusa is **int**, ez lesz az eredmény típusa is. A konkrét számértékek közömbösek. Az így kapott egész osztás eredménye pedig 0.

Mi más lehetne két egész szám hányadosa? Bizonyos programozási nyelvek más jelölést használnak az egész és a lebegőpontos osztás jelölésére. A Pascal pl. a **div** operátort használja egész, és a **/** operátort lebegőpontos eredmény létrehozásásra. A Python3-ban a **/** lebegőpontos eredményt ad (a Python2-ben még nem!). Viszont ezek a példák sem kivételek: a Pascal és Python3 **/** művelete *mindig* lebegőpontos eredményt ad, akkor is, ha matematikailag a hányados egész lenne, pl. 4/2.

Próbáljuk ki 5/9 helyett az 5./9. kifejezést. (Valójában elég lenne 5./9 is, mert ha az egyik operátor lebegőpontos, akkor a C a másikat is azzá konvertálja).

```c
/*
 *  BAD VERSION !!!
 *  Convert Fahrenheit to Celsius
 *  between -100F and +400F  by 100F
 */
#include <stdio.h>
int main()
{
  int fahr;

  for ( fahr = -100; fahr <= 400; fahr += 100 )
  {
    printf("Fahr = %d,\tCels = %d\n",fahr,5./9.*(fahr-32));
  }
  return 0;
}
```

```
$ gcc -ansi -pedantic -W -Wall fahrenheit.c -o fahrenheit
fahrenheit.c: In function 'main':
fahrenheit.c:17:5: warning: format '%d' expects argument of type
'int', but argument 3 has type 'double' [-Wformat=]
     printf( "Fahr = %d,\tCels = %d\n", fahr, 5./9.*(fahr-32) );
     ^

$ ./fahrenheit
Fahr = -100,    Cels = 913552376
Fahr = 0,       Cels = -722576928
Fahr = 100,     Cels = -722576928
Fahr = 200,     Cels = -722576928
Fahr = 300,     Cels = -722576928
Fahr = 400,     Cels = -722576928
```

Még mindig hibás a program. Most a Celsius értéket helyesen számoltuk ki, a típusa **double**, de a kiíráskor egész számként próbáljuk kiírni a *%d* formátummal. Egy pl. 8

bájtos lebegőpontos számot adunk át paraméterként és az első 4 bájtját próbáljuk egész számkét értelmezni. Ez értelemszerűen hibához vezet.

A C nyelvben paraméterátadáskor csak akkor történik konverzió, ha a hívott függvényt teljes paraméterlistával előzetesen deklaráljuk. A printf esetében a paraméterek feloldása futási időben történik.

```c
/*
 *  UGLY VERSION
 *  Convert Fahrenheit to Celsius
 *  between -100F and +400F  by 100F
 */
#include <stdio.h>
int main()
{
  int fahr;

  for ( fahr = -100; fahr <= 400; fahr += 100 )
  {
    printf("Fahr = %d,\tCels = %f\n",fahr,5./9.*(fahr-32));
  }
  return 0;
}
```

```
$ gcc -ansi -pedantic -W -Wall fahrenheit.c -o fahrenheit

$ ./fahrenheit
Fahr = -100,    Cels = -73.333333
Fahr = 0,       Cels = -17.777778
Fahr = 100,     Cels = 37.777778
Fahr = 200,     Cels = 93.333333
Fahr = 300,     Cels = 148.888889
Fahr = 400,     Cels = 204.444444
```

Most már működik, de az input nem szépen formázott. Ráadásul a program közepén van egy bonyolult képlet. Refaktoráljuk ki ezt a képletet egy önálló függvénybe.

Figyeljük meg, hogy a függvény **szignatúrája** (signature) **double fahr2cels(double)**, ezért az **int** típusú *aktuális* paraméter lebegőpontossá konvertálva adódik át. A **%7.2f** formátum 7 karakter szélességben, 2 tizedesre kerekítve írja ki az eredményt.

```c
/*
 *  OK
 *  Convert Fahrenheit to Celsius
 *  between -100F and +400F  by 100F
 */
#include <stdio.h>
double fahr2cels( double f)
{
  return 5./9. * (f-32);
}
int main()
```

```
12    {
13      int fahr;
14
15      for ( fahr = -100; fahr <= 400; fahr += 100 )
16      {
17        printf("Fahr = %4d,\tCels = %7.2f\n",
18                              fahr, fahr2cels(fahr));
19      }
20      return 0;
21    }
```

```
$ gcc -ansi -pedantic -W -Wall fahrenheit.c -o fahrenheit

$ ./fahrenheit
Fahr = -100,    Cels =  -73.33
Fahr =    0,    Cels =  -17.78
Fahr =  100,    Cels =   37.78
Fahr =  200,    Cels =   93.33
Fahr =  300,    Cels =  148.89
Fahr =  400,    Cels =  204.44
```

A programot még tovább javíthatjuk a kódban szereplő *mágikus konstansok* kiemelésével. Ebben a verzióban **előfordító direktívákat** (preprocessor directive) alkalmazunk, hogy a program konstansait megadjuk.

```
1     /*
2      *  OK, with #define
3      *  Convert Fahrenheit to Celsius
4      *  between -100F and +400F  by 100F
5      */
6     #include <stdio.h>
7     #define LOWER -100
8     #define UPPER  400
9     #define STEP   100
10    double fahr2cels( double f)
11    {
12      return 5./9. * (f-32);
13    }
14    int main()
15    {
16      int fahr;
17
18      for ( fahr = LOWER; fahr <= UPPER; fahr += STEP )
19      {
20        printf( "Fahr = %4d,\tCels = %7.2f\n",
21                              fahr, fahr2cels(fahr) );
22      }
23      return 0;
24    }
```

A következő verzióban névvel ellátott konstansokat alkalmazunk ugyanerre.

```c
/*
 *  OK, with const
 *  Convert Fahrenheit to Celsius
 *  between -100F and +400F  by 100F
 */
#include <stdio.h>
const int lower = -100;
const int upper =  400;
const int step  =  100;
double fahr2cels( double f)
{
  return 5./9. * (f-32);
}
int main()
{
  int fahr;

  for ( fahr = lower; fahr <= upper; fahr += step )
  {
    printf( "Fahr = %4d,\tCels = %7.2f\n",
                          fahr, fahr2cels(fahr) );
  }
  return 0;
}
```

Homepage of Dr. Zoltán Porkoláb

Home

Archive

Teaching

Timetable

Bolyai College

C++ (for mathematicians)

Imperative programming (BSc)

Multiparadigm programming (MSc)

Programming (MSc Aut. Sys.)

Programming languages (PhD)

Software technology lab

Theses proposals (BSc and MSc)

Research

CodeChecker

CodeCompass

Templight

Projects

Conferences

Publications

PhD students

Affiliations

Dept. of Programming Languages and Compilers

Ericsson Hungary Ltd

Imperatív programozás 3.

A C programok szerkezete

A C programozási nyelvű programjainkat különálló **fordítási egységek** (translational unit, TU), lényegében **forrásfájlok** (source file) halmazaként írjuk meg. Ezeket a fájlokat **.c** kiterjesztéssel kell elkészítenünk, a C fordító csak a .c kiterjesztésű fájlokat fordítja le.

A forrásfájlokba háromféle dolgot írhatunk:

- előfordító utasítások (preprocessor directive)
- kommenteket (comment)
- C nyelvi tokeneket (token)

```
1   /*
2    * my first C program    <--- comment
3    *
4    */
5   #include <stdio.h>   <---- preprocessor directvive
6
7   int main()               int    <-- type name: keyword
8                            main   <-- function name:
9   identifier
10                           ()     <-- function call: operator
11  {                        {      <-- block begin: separator
12                           printf <-- function name:
13  identifier
14                           (      <-- function call: operator
15    printf("Hello world"); "Hello world"  <-- string literal,
16  type char[12]
17                           )      <-- function call: operator
18                           ; <-- command-end separator
                             return <-- keyword
      return 0;              0      <-- decimal int literal,
    type int
    }                        }      <-- block end: separator
```

Preprocesszor utasítások

Az előfordító a C/C++ fordítás első logikai lépése. Gyakran ténylegesen egy külön program (cpp) hajtja végre, emiatt akár más programozási nyelvekhez is használhatjuk. Az előfordító feladata a **header** fájlok betöltése, a **makrók** kifejtése, **feltételes fordítás** és a sorok kezelése. Például az előfordító kidobja a forrásfájlból az **<újsor>** karakterpárokat, így a sor végére írt __ segítségével tudunk folytatósorokat írni.

Include utasítás

Az **include** utasítás a sort kicseréli a fájl tartalmára. A legtöbbször a fájl deklarációkat tartalmaz, az *stdio.h* pl. az input-output tevékenységekkel kapcsolatban. Az ilyen

fájlokat nevezzük **header** fájloknak. A header fájlok legtöbbször (de nem kötelezően) **.h** kiterjesztésűek.

```
#include <stdio.h>
#include "filename2"
#include "../relative/filename3.h"
```

A fájlokat a szabványos include keresési úton (include path) keressük, a **""** esetén ez kiegészül a kurrens könyvtárral. A keresési utat mi is kiegészíthetjük, pl. gcc-nél a parancssori **-I/dir1/dir2** kapcsolóval.

```
$ gcc -I/usr/local/include/add/path1 -I/usr/local/include/add/path2
...
```

Makró definíciók

Kétféle makró létezik, a változószerű, amelyiknek nincsen paramétere és a függvényszerű, aminek van. Egy makrót a **#define** paranccsal definiálunk és hatását ki lehet kapcsolni az **#undef** parancssal.

```
#define <identifier>  <token-list>
#define <identifier>(param1, param2, ..., paramN) <token-list>
:
#undef <identifier>
```

Példák makrók definiálására és használatára:

```
1    #define BUFSIZE    1024
2    #define PI         3.14159
3    #define USAGE_MSG  "Usage: command -flags args..."
4    #define LONG_MACRO struct MyType \
5                       {             \
6                         int data;   \
7                       };
8    #define FAHR2CELS(x)  ((5./9.)*(x-32))
9    #define MAX(a,b)  ((a) > (b) ? (a) : (b))
10   :
11   char buffer[BUFSIZE];
12   fgets(buffer, BUFSIZE, stdin);
13   c = FAHR2CELS(f);
14   x = MAX(x,-x);
15   x = MAX(++y,z);
```

Feltételes fordítás

A feltételes fordítás során bizonyos kódrészletek fordítását ki- vagy bekapcsolhatjuk. A feltételes fordítást felhasználhatjuk a kód konfigurálására az #if #ifdef #ifndef #elif #else #endif parancsokkal.

```
1    #if DEBUG_LEVEL > 2
2      fprintf("program was in file %s, line %d\n", __FILE__,
3    __LINE__);
4    #endif
5    :
```

```
#ifdef __unix__ /* __unix__ is usually defined by compilers
for Unix */
#  include <unistd.h>
#elif defined _WIN32 /* _Win32 is usually defined for 32/64
bit Windows */
#  include <windows.h>
#endif
:
#if !(defined( __unix__ ) || defined (_WIN32) )
  /* ... */
#else
  /* ... */
#endif
:
#if RUBY_VERSION == 190
# error 1.9.0 not supported
#endif
```

Az #error parancs hatására a fordítás hibával megáll, és a hibaüzenet jelenik meg.

A feltételes fordítás egyik leggyakoribb esete a **header őrszemek** (header guard) alkalmazása. Ennek az az értelme, hogy megelőzzük a többszörös deklarációkat.

```
#ifndef MYHEADER_H
#define MYHEADER_H
:
/* header content */
:
#endif /* MYHEADER_H */
```

Standard makrók

```
__FILE__
__LINE__
__DATE__
__TIME__
__STDC__
__STDC_VERSION__
__cplusplus
:
#ifdef __cplusplus
extern C {
#endif
/* ... */
#ifdef __cplusplus
}
#endif
```

A **LINE** és **FILE** makrók értékeit szabályozhatjuk a #line paranccsal:

```
#line 1000 "myfile.c"
fprintf("program was in file %s, line %d\n", __FILE__,
```

```
    __LINE__);
```

String műveletek

Stringesítés

```
1   #define str(s) #s
2   #define BUFSIZE 1024
3   // ...
4   str(\n)       -->   "\n"
5   str(BUFSIZE)  -->   1024
```

String konkatenáció

```
1   struct my_int_20_array
2   {
3     int v[20];
4   };
5   struct my_int_30_array
6   {
7     int v[30];
8   };
9   struct my_double_40_array
10  {
11    double v[40];
12  };
13
14  #define DECLARE_ARRAY(NAME, TYPE, SIZE) \
15  typedef struct TYPE##_##SIZE##_array    \
16  {                                       \
17    TYPE v[SIZE];                         \
18                                          \
19  } NAME##_t;
20
21  DECLARE_ARRAY(yours,float,10);
22  yours_t x, y;
```

Egyéb

A #pragma utasítás segítségével fordítófüggő akciókat definiálhatunk. Ilyen akciók lehetnek biznyos warning-ok be/kikapcsolása, stb. A **#pragma once**, amit gyakran látunk használni a header őrszemek helyett *nem szabványos*!

Kommentek

A kommentek nem kerülnek a fordítóprogram által felhasználásra, de fontosak lehetnek a program megértése, későbbi karbantartása, módosítása szempontjából. Mindig törekedjük önmagát magyarázó, világos programozási stílusra, de ezt kommentekkel kiegészíthetjük a kód által nem kifejezhető információkkal.

A klasszikus C kommentek a /* és */ szimbólumok között helyezkednek el, akár több soron át, de nem egymásba ágyazhatóak. A többsoros kommentek, melyek a **//** szimbólumtól a sor végéig tartanak, csak a C99 szabványtól használhatóak.

```
1   /* multi
2      line
3      comments  // hiding single line comments
4   */
5   /************************************\
6   *                                    *
7   *  exist in various style and format  *
8   *                                    *
9   \************************************/
10  :
11  /*
12     /*  but can not be nested */
13  */
```

"Hel /* this is not a comment */ lo": A stringeken belül nem használhatunk kommenteket.

C tokenek

A C forrásfájl a kommenteken és az előfordító utasításokon túl ún. C nyelvi k__token__-eket tartalmaz. A token ebben az értelemben tovább nem bontható elemi nyelvi egység. A legtöbb modern programozási nyelvben a tokenek között tetszőleges üres helyet (whitespace) hagyhatunk: space, tabulátor vagy újsor karakter formájában. A Pythonban ugyanakkor a helyesen elhelyezett indentálás alapján dől el a program struktúrája.

Az imperatív programozási nyelvek token típusai meglehetősen hasonlóak:

- kulcsszavak (keyword)
- azonosítók (identifier)
- konstansok/literálok (literal)
- operátorok (operator)
- egyebek, a C-ben szeparátorok (separators)

Kulcsszavak

Ezek a programozási nyelv "beépített" szavai: pl utasítások nevei (pl. **if**, **while**), gyakran az alaptípusok nevei (pl. C-ben: **int**, **double**), és pár más kulcsszó (pl. C-ben **extern**, **typedef**, stb.)

A C-ben ezek mind csupa kisbetűvel írandóak, és más nyelvekhez képest nagyon kevés van belőlük:

- C89: 32
- C99: +5
- C11: +7

Azonosítók

Azok a nevek, amit mi adunk egyes programelemeknek: változóknak, függvényeknek, új típusoknak, stb.

A C-ben az azonosítók

- betűvel kezdődnek (betűnek számít az '_' alulvonás, underscore karakter is)
- betűkkel és számokkal folytatódhatnak akármilyen hosszan

- de a fordító csak az első 63/31 betűt veszi figyelembe
- tilos kulcsszavakat használni
- a kis és nagybetűket megkülönböztetjük

Okos gondolat a neveket konzisztensen használni és alaposan átgondolni a névválasztást. Minél nagyobb területen használható egy függvény vagy változó neve, annál inkább segít a program megértésében, ha jól választjuk meg. Ugyanakkor egy ciklusváltozót nevezhetünk **i**-nek, mindenki látni fogja, hogy az egy ciklusváltozó.

Vannak bizonyos elterjedt konvenciók:

- camelCaseNotation
- CTypenamesStartsWithUppercase
- under_score_notation

Ezen a honlapon elérhető egy evvel kapcsolatos tanulmány és egy másik cikk.

A MACRO_NEVEK_MINDIG_CSUPA_NAGYBETUSOK az általános C szokások szerint.

Régebben szokásos volt használni C-ben (és néhány más nyelvben) az ún. Hungarian Notation névkonvenciót, ami a névbe belerakta a típussal és használatával kapcsolatos alapvető információkat. Az elnevezés a kitalálójára Charles Simonyira utal.

Konstansok/Literálok

Lényegében a programunkban felhasznált konstansok, értékek. Számok, karakterek, karakterkáncok, amiknek **értéke** és **típusa** van. Az, hogy egy nyelvben mi használható literálként, az összefügg a nyelv céljaival, absztrakciós szintjével.

Egész számok

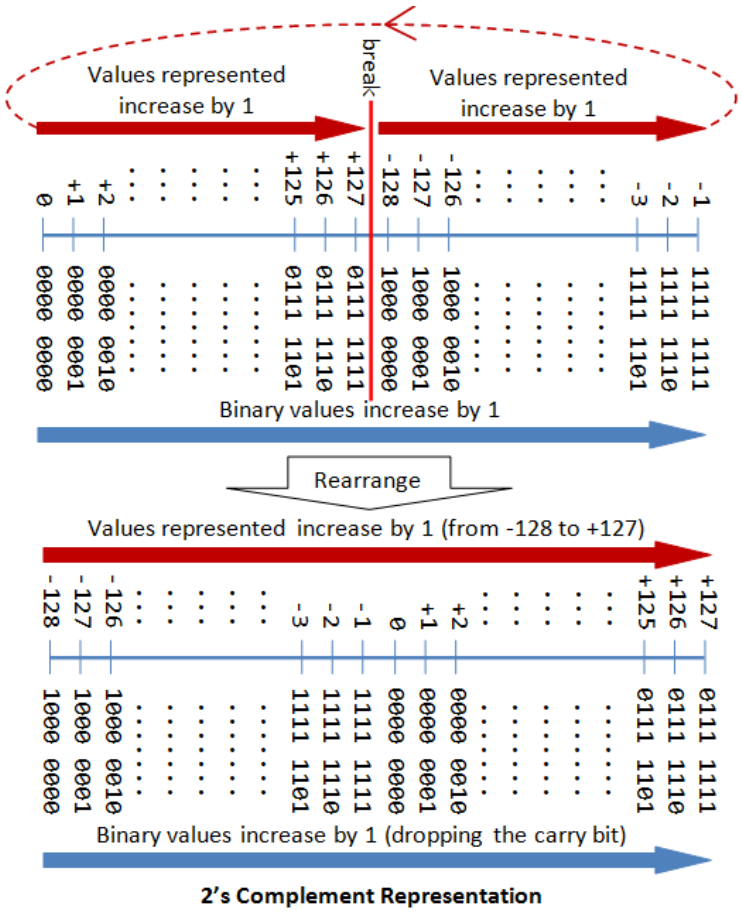
megnevezés	példa	típus	értéke
decimmális egész	25	int	25
oktális egész	031	int	25
hexadecimális egész	0x19	int	25
hosszú egész	12L	long int	12
C99 méghosszabb egész	12LL	long long int	12
előjel nélküli egész	12u	unsigned int	12

Számos programozási nyelv rögzíti az egyes típusok méretét vagy értékhatárát. A Pascal-ban pl. az **integer** típus 2 byte-os, ami azt jelenti, hogy pl. egy nagyobb fájlban lebegőpontos számmal kell pozícionálnunk. A Java ugyancsak rögzíti az egészek méretét, aminek a futási idejű hordozhatóság az oka.

A C nyelv a típusok méretét nem definiálja, csak a számábrázolási minimum értéket adja meg. Viszont a számoknak több méretbeli variánsát is adja. Így pl. egy **short int** legalább két bájt, egy **int** legalább négy bájt méretű. A fordító mindig az adott platformhoz legalkalmasabb méretet választhatja. Azt viszont (fordítási időben) lekérdezhetjük a **sizeof** operátorral, hogy az adott platformon mi egy konkrét típus vagy valamely kifejezés típusának mérete. Egyes típusok mérete között fennállnak relációk:

```
   sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
// at least 16 bit              at least 32 bit    at least 64 bit
```

Az egész jellegű számokat gyakran az ún. **kettes komplemens** kódban ábrázolják.



2's Complement Representation

Az egész számoknak vannak előjeles (**signed**) és előjel nélküli (**unsigned**) verziója. A számoknál (de nem a **char** típusnál) az alapértelmezés az előjeles típus, azaz az **int** az ugyanaz, mint a **signed int**. Az előjel nélküli típusok mérete megegyezik az előjeles típuséval, de az előjel bit is értéket jelöl, így a számábrázolási tartományuk 0-tól az előjeles számok pozitív maximumának (kb) kétszereséig terjed.

Az előjeles egész számok túl- vagy alulcsordulása nem definiált és hibás működést vonhat maga után. Az előjel nélküli típusok esetében viszont a számábrázolási tartomány szerinti moduló alapján értelmezzük ezeket.

```
signed   int  i = 1;  /* ugyanaz, mint int i */
unsigned int ui = 1;  /* ugyanaz, mint unsigned ui */
 i -= 2;              /* i == -1 */
ui -= 2;              /* ui értéke egy nagy pozitív */
```

Karakterek

A karakterekből is több fajta van a C nyelvben. Az egyszerű karakterek egy aposztróf pár között szerepelnek, kivéve a ' (single-quote) , \ (backslash) és az újsor karakter. Ezeknek a karaktereknek típus **char** értéke a megfelelő karakterkód.

Egyes speciális karaktereket, az ún. escape sorozattal tudunk leírni:

```
- '\''    single quote
- '\"'    double quote
- '\?'    question mark
- '\\'    backslash
- '\a'    bell (audio)
- '\b'    backspace
- '\f'    form feed -- new page
```

```
- '\n'    newline
- '\r'    carriage return
- '\t'    horizontal tab
- '\v'    vertical tab
```

Az ezektől eltérő karaktereket is megadhatjuk a kódjukkal:

```
- oktális forma:       '\377'  -> 11111111
- hexadecimális forma: '\xff'  -> 11111111
- univerzális karakter értékek (C99 óta):
  - '\U1234'               típus = char16_t (min 16bit)
  - '\U12345678            típus = char32_t (min 32bit)
```

A karakterek alapértelmezetten **char** típusúak, a C99 óta léteznek 16 és 32 byte-os karakter típusok (**char16_t** ill. **char32_t**). A leghosszabb karakter típus a **wchar_t**.

```
 1 == sizeof(char) < sizeof(char16_t) <= sizeof(char32_t) <=
sizeof(wchar_t)
```

A karakter típusok között is létezik **signed** és **unsigned** típus, de ellentétben az egészekkel, itt a nem minősített **char** típus nem feltétlenül azonos a **signed char** típussal. Az "előjeles" karakterek értelme, hogy ha egészekkel hasonlítjuk össze őket, akkor a 128 feletti ASCII értékek nullánál kisebbek lesznek.

```
         char  ch = '\xff';
unsigned char uch = '\xff';
  signed char sch = '\xff';
:
uch > 0   /* true */
sch < 0   /* true */
 ch < 0   /* true on some platforms, false on others */
```

Boolean

Az ANSI C89-ben nem volt speciális logikai (igaz/hamis) típus, a C99 adta hozzá a nyelvhez a _Bool típust és a **bool** makrót. Klasszikusan az egész értékek közül a nulla hamisnak, minden nem nulla érték igaznak számít. Ezen kívül bizonyos program-környezetekben (pl. elágazásban vagy ciklusban) a pointerek is logikai értékként értékelődnek ki, a NULL pointer hamis, a többi igaz érték.

Amikor C operátorok logikai értékeket készítenek, akkor az igaz értéke **1**, a hamis **0**.

- C99 óta kulcsszó: _Bool
- C99 előtt makró: bool, true, false, használatukhoz kell az <stdbool.h>

A logikai és egész értékek eltérő módon konvertálódnak:

```
1    #include <stdio.h>
2    int main()
3    {
4      printf("_Bool == %d\t int == %d\n",
5                          (_Bool) 0.5, (int) 0.5 );
6      return 0;
7    }
```

```
$ ./a.out
_Bool == 1      int == 0
```

Lebegőpontos számok

A valós számok kezelését a számítástechnikában a **fixpontos** (fixed point) és a **lebegőpontos** (floating point) számábrázolás teszi lehetővé. A fixpontos ábrázolás esetében előre rögzítjük, hogy a rendelkezésre álló memóriaterületen hány biten ábrázoljuk az egész és hányon a tört részt.

A lebegőpontos ábrázolás esetében is két részt tárolunk. A **mantissza** egy előjeles szám, melynek gyakran az abszolút értéke az [1,2] intervallumban van. A **karakterisztika** vagyis az exponenciális rész pedig egy szintén előjeles szám, ami a szám nagyságrendjét adja meg, azaz egy bázis **kitevője**. A legtöbbször mind a mantisza, mind a karakterisztika bináris szám és a bázis értéke is 2.

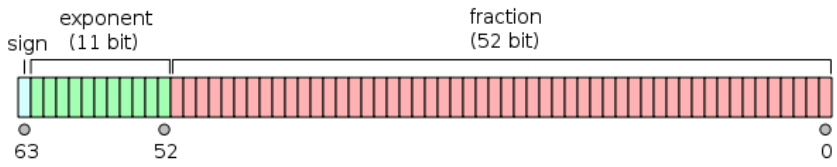
Azaz, ha a lebegőpontos szám formátuma **(m,c)**, akkor értéke **m** * $2^{\mathbf{c}}$. Ha **m** negatív, akkor a lebegőpontos szám negatív. Ha **c** negatív, akkor a lebegőpontos szám abszolút értéke kisebb, mint 1.

A lebegőpontos számok előnye a fixpontossal szemben, hogy nagyon nagy és nagyon pici abszolút értékű számokat is képesek megfelelő pontossággal ábrázolni. Persze, ha műveletet képezünk nygaon nagy és nagyon pici számok között, akkor kerekítési hibák is történhetnek.

A modern számítógépek és programozási nyelvek a szabványos **IEEE 754** lebegőpontos ábrázolást használják.

	bitek	előjel	mantissza	karakterisztika
egyszeres	32	1	23	8
dupla	64	1	52	11
kiterjesztett	80	1	64	15
négyszeres	128	1	112	15

Például a 64 bites lebegőpontos szám így néz ki:



Bonyolultabb lebegőpontos számolásoknél előfordulhat **túlcsordulás** vagy **alulcsordulás**. Az ilyen esetek kezelsére bevezettek pár speciális lebegőpontos értéket:

- A plusz és minusz végtelen
- A plusz és minusz nulla
- A denormalizált számokat
- "NEM SZÁM" **NaN** értéket

Soha ne használjunk lebegőpontos számokat olyan esetekben, amikor precíz, kerekítés nélküli értékekre van szükségünk (pl. fizetések ábrázolására).

Ez programozási nyelvtől független kérdés. A lebegőpontos ábrázolás óhatatlan velejárói a kerekítések. Így pl. 1.03 - .42 eredménye könnyen lehet: 0.6100000000000001 .

A C-ben a lebegőpontos számok típusai:

C típus	Példa	IEEE 754
float	3.14f	egyszeres
double	3.14	dupla
long double	3.14l	kiterjesztett vagy négyszeres

A pontos méreteket a **sizeof** operátorral lehet meghatározni.

```
sizeof(float) <= sizeof(double) <= sizeof(long double)
```

Komplex számok

A komplex számok a Boolean típusokhoz hasonlóan csak a C99 óta része a nyelvnek. A megoldás is hasonló: a _Complex kulcsszót, vagy a használatával a __complex__ makrót lehet használni.

```
                        <complex.h>


float _Complex          float complex
double _Complex         double complex
long double _Complex    long double complex
```

Példa:

```
1   #include <complex.h>
2   #include <stdio.h>
3   int main(void)
4   {
5       double complex z = 1 + 2*I;
6       z = 1/z;
7       printf("1/(1.0+2.0i) = %.1f%+.1fi\n", creal(z),
8   cimag(z));
    }
```

A komplex számok használatához használni kell a matematikai könyvtárat! Ezt a szerkesztésnél (linkelésnél) a **-lm** kapcsolóval adjuk meg.

```
$ gcc  -ansi -pedantic -Wall -W complex.c -lm
$ ./a.out
1/(1.0+2.0i) = 0.2-0.4i
```

String literálok

Egyes programozási nyelvekben a stringek elemi típusok, melyekkel hasonló módon végezhetünk műveleteket, mint pl. számokkal. Más nyelvekben a string nem módosítható, ún. **immutábilis** érték, amikkel lehet műveleteket végezni, de magukat a stringeket nem tudjuk megváltoztatni.

A C nyelvben a stringek nem elemi típusok, nem tudunk közvetlenül műveleteket alkalmazni rájuk. Lényegében karaktertömbök, de a C-ben a tömbökkel sem tudunk elemi műveleteket végrahajtani. A C stringeket a **string.h** headerfájlban deklarált függvényekkel tudjuk majd kezelni.

A string literál egy összefüggő memóriaterületen lefoglalt névtelen karaktertömb, melyek egy **NUL** karakter (‘\0’) zár le. A string literál típusa **karakter tömb**, melynek mérete tartalmazza a lezáró karaktert is.

A string literálok **immutábilisak**, azaz nem módosíthatóak, de felhasználhatóak karakter tömbök vagy karakterre mutató pointerek inicializálására. Amennyiben egy string literált módosítani próbálunk, az *nem definiált viselkedés* és futási idejű hibához vezethet.

A fordító alkalmazhat olyan optimalizálást, hogy két azonos string literált egyetlen egy példányban tárol, vagy akár egy literált egy részét is újra felhasználhatja.

Az egymás mögé írt, csak üreshelyekkel elválasztott string literálokat a fordító egyetlen stringgé ragasztja össze.

```
1  char t1[] = {'H','e','l','l','o','\0'}; /*sizeof(t1) == 6 */
2  char t2[] = "Hello";                    /*sizeof(t2) == 6 */
3  t1[1] = 'a';          /* ok, a tömb t1[1] elemét módosítja */
4  t2[1] = 'o';          /* ok, a tömb t2[1] elemét módosítja */
5  char *p = "Hello";    /* sizeof(p) a pointer mérete  */
6  char *q = "Hello";    /* q ugyanoda mutathat, mint p */
7  char *s = "lo"        /* s mutathat p[3]-ra          */
8  p[1] = 'a'; /* futási idejű hiba lehet: undefined behavior */
9  char *r = "Hi"  " "  "world";  /* ugyanaz, mint "Hi world" */
```

void

A **void** típus az üreshalmaznak felel meg, nem lehet értékeket létrehozni void típussal. A void kulcsszót használjuk a visszatérő érték nélküli függvények deklarációjához és a paraméter nélküli függvények jelölésére is. Ugyanakkor létre tudunk hozni **void *** típusú pointereket, amelyeket általános, típustalan mutatókként tudunk használni.

Imperatív programozás 4.

Operátorok, kifejezések, utasítások.

Kifejezések

A legelső magasszintű programozási nyelvek, mint pl. a Fortran, egyik elsődleges célkitűzése volt, hogy a programokban matematikai kifejezéseket tudjunk használni. A kifejezések - melyek a matematikai egyenletekhez hasonlítottak - változókból (amelyek egy-egy memória-területet azonosítottak) és operátorokból (melyek a matematikai műveleti jeleknek feleltek meg) álltak. Általánosan, a **kifejezéseket** a programozási nyelvekben **operátorok**-ból és **konstans** értékekből vagy **változókból** képezzük.

Az alábbi kifejezés például számos programozási nyelvben érvényes:

```
A + B * C
```

Hasonlóan a matematikai egyenletekhez, a kifejezésekben is fontos, hogy melyik az "erősebb" művelet, azaz hogyan kell értelmeznünk (zárójeleznünk) egy kifejezést. Ebben az egyes műveleteket leíró operátorok **precedenciája** (erőssége) az iránymutató. A szorzás például **magasabb precedenciájú**, mint az összeadás, ezért a fenti kifejezést alábbi módon kell értelmezni:

```
A + (B * C)
```

mivel a szorzás magasabb precedenciájú, mint az összeadás. ha ettől eltérő viselkedést szeretnénk, akkor azt zárójelezéssel jelezhetjük. Ilyen értelemben ez a kifejezés hasonlóan működik, mint a megfelelő matematikai képlet. Azért ez ne tévesszen meg bennünket, *nem matematikai képleteket* írunk a programozási nyelvekben, hanem *kifejezéseket* (expression), melyek egyrészt viselkedhetnek másképpen, mint azt a matematikában megszoktuk, másrészt lehet **mellékhatásuk** (side effect), azaz valami egyéb akciót is végrehajthatnak, miközben **kiértékeljük** (evaluate) a kifejezéseket.

A *funkcionális programozási nyelvekben* pont ezek a mellékhatások hiányoznak, ezért az ott leírt függvények sokkal inkább matematikai *pure* jellegűek.

A FORTRAN77 nyelvi verzióban az azonos precedenciájú műveletek sorrendje nem volt meghatározott. Azaz, ha nem írtunk zárójeleket az alábbi kifejezésbe

```
A * B / C * D
```

akkor az jelenthette az alábbi zárójelezések bármelyikét:

```
((A * B) / C) * D
(A * B * D) / C
(A / C) * B * D
```

Könnyen látható, hogy ha pl. A, B, C, D egész számok (Fortran INTEGER típus), akkor az egész értékű osztás miatt az egyes kiértékelési sorrendek eredménye eltérő lehet. A kerekítési hibák miatt még akkor is kaphatnánk eltérő eredményt, ha az értékek lebegőpontos számok lennének (Fortran REAL vagy DOUBLE PRECISION típus).

A modern programozási nyelvekben az egyes kifejezések értelmét az operátor-__precedencia__ (precedence) mellett az ún. **asszociativitás** (associativity) határozza meg. Az asszociativitás azt definiálja, hogy *azonos precedencia szintű* operátorok esetében hogyan (balról-jobbra vagy jobbról-balra) kell (gondolatban) zárójelezni a kifejezéseket.

A kifejezéseknek **típusa** és **értéke** van. A statikus típusrendszerű programozási nyelvekben (ilyen a C, Java, C#, és sok másik nyelv) a kifejezések *típusát* a fordítási időben megállapítja a fordítóprogram. A kifejezések *értékét* legtöbbször csak futási időben lehet megállapítani, de vannak kivételes esetek, amikor ez az érték fordítási időben ismert. Ezeket a kifejezéseket **konstans kifejezéseknek** (constant expression) nevezzük.

A C nyelv operátorai

A C programozási nyelvre (és leszármazottjaira) jellemző, hogy sok operátort használhatunk, köztük olyanokat is, melyek más nyelvekben utasítások, függvények, vagy egyáltalán nem is léteznek.

Precedencia	Operátor	Leírás	Assoc
Posztfix	++	posztfix növelés	L->R
	– –	posztfix csökkentés	
	()	függvényhívás	
	[]	tömb index	
	.	struct/union tag elérés	
	->	tag elérés mutatóval	
	(type){list}	összetett literál (C99)	
Unáris	++	prefix növelés	**R->L**
	– –	prefix csökkentés	
	+	pozitívlőjel	
	–	negatív előjel	
	!	logikai negáció	
	~	bitenkénti negáció	
	(type)	típus konverzió	
	*	pointer indiekció	
	&	címoperátor	
	sizeof	típus/objektum mérete	
	_Alignof	igazítási követelmény (C11)	
Multiplikatív	* / %	szorzás, osztás, maradék	L->R
Additív	+ –	összeadás, kivonás	L->R
Léptetés	« »	bitenkénti bal/jobb léptetés	L->R
Relációs	< <= > >=	relációs műveletek	L->R
Egyenlőség	== !=	egyenlő, nem egyenlő	L->R
Bitenkénti	&	bitenkénti és (AND)	L->R
	^	bitenkénti kizáró vagy (XOR)	L->R
	|	bitenknti vagy (OR)	L->R
Logikai	&&	logikai és AND	L->R
	||	logikai vagy OR	L->R
Terciális	? :	feltételes kifejezés	**R->L**
Értékadás	=	értékadás	**R->L**
	+= –=	összetett értékadások	
	*= /= %=		

Precedencia	Operátor	Leírás	Assoc
	«= »=		
	&= |= ^=		
Szekvencia	,	vessző (szekvencia) operátor	L->R

Megjegyzések

Nem kiértékelt operátorok

Néhány operátor ún. **nem kiértékelt** (unevaluated), azaz futási időben ténylegesen nem történik velük semmi, Ezek a műveletek fordítási időben felhasználható információt szolgáltatnak. C-ben ilyen az **_Alignof** és a **sizeof**. Ebből leginkább a sizeof-ot használjuk, ami egy típus méretét adja meg bájtokban. Például:

```
1        size_t int_size = sizeof(printf("%d", 42));
```

nem ír ki semmit sem az outputra, de int_size értéke 4 lesz (4 bájtos integer méret esetén).

Bináris vagy/és precedenciája

Figyeljünk arra, hogy pár operátornak nem túl magától értetődő a percedenciája. Például a *bitenkénti és vagy* műveletek "gyengébbek", mint a relációs operátorok. Ebből furcsa hibák következhetnek:

```
1           if ( flag & 0xff == 0 )
```

valójában

```
1           if ( flag & (0xff == 0) )
```

lesz és mindig *hamis*. Az ilyen hibák elkerüléséhez mindig írjuk ki a zárójeleket a kifejezéseinkben:

```
1           if ( (flag & 0xff) == 0 )
```

Értékadás vs. egyenlőségvizsgálat

Hasonlóan figyelni kell az értékadás operátor és az egyenlőségvizsgálat különbségére. Az alábbi esetben

```
1               x = 10;
2               /* ... */
3               if ( x = 0 )
```

nem egyenlőségvizsgálat, hanem értékadás történik. Miután **x** felvette a 0 értéket, a kifejezés értéke 0 és *hamis* lesz. Egy praktikus ötlet: konstanssal való összehasonlításkor írjuk balra a konstanst, így szintaktikus hibát kapnánk, ha elhagyánk egy karaktert:

```
1               if ( 0 = x )
```

Ez utóbbi programozási stílust Yoda conditions-nak nevezik.

Az értékadás operátor és a másolás szemantikája

Az értékadás a programozási nyelvek jó részében *utasítás* és csak a C nyelv óta használják *kifejezésként*. Ennek a C-ben csak annyi hatása van, hogy az értékadásnak van *eredménye*, amit fel lehet használni egy további kifejezésben:

```
1        int a, b;
2        a = 3+(b = 5);  /* a = (3 + (b = 5) )*/
```

Itt **a** értéke 8, **b** értéke 5 lesz. Persze ilyet ritkán csinálunk. Gyakrabban fordul elő, hogy több változónak adunk értéket, de figyeljünk arra, hogy ez *nem párhuzamos értékadás*, hanem jobbról balra haladó 3 különálló értékadás.

```
1        double a, c;
2        int    b;
3        a = b = c = 3.14;  /* a = (b = (c = 3.14) ) */
```

Ami után **c** értéke 3.14, **b** értéke 3 és **a** értéke 3.0 lesz.

Az értékadás működik néhány összetett típusra is, pl. **struct** és **union**, de nem működik *tömbökre*.

```
1    #include <stdio.h>
2
3    struct X
4    {
5      int    i;
6      double d;
7      int   *ptr;
8    };
9    void f()
10   {
11     int      zz = 1;
12     struct X aa;
13     struct X bb;
14
15     aa.i   = 1;
16     aa.d   = 3.14;
17     aa.ptr = &zz;
18
19     bb = aa;  /* 1==bb.i és 3.14==bb.d és *aa.ptr==*bb.ptr */
20     ++*aa.ptr; /* 2==zz  és 2==*aa.ptr  és  2==*bb.ptr !!! */
21   }
```

Ilyenkor tagonkénti értékadás történik (valójában egyszerűen **aa** teljes területe átmásolódik **bb**-be). Az ilyen értékadások azonban lehetnek veszélyesek, ha pl. az egyik tag *pointer*, akkor **aa.ptr** és **bb.ptr** ugyanoda *mutat*, tehát ha az egyik módosítja a mutatott területet, akkor a másik is ezt a módosított értéket fogja látni.

Később, objektum-orientált nyelvekben gyakori lesz, hogy egy osztályt úgy implementálunk, hogy egy objektumból egy pointer mutat valami dinamikusan lefoglalt memóriaterületre. Ilyenkor a pointer által mutatott terület *logikailag* az objektum sajátja, és ha másoljuk az objektumot, akkor nem a pointert, hanem a *mutatott tárterületet* kéne másolni.

Azokban a nyelvekben, ahol az operátorokat *túlterhelhetjük* és az értékadás operátor, írhatunk saját értékadás operátort, ami elvégzi a kívánt tevékenységet. Ilyen a C++ *másoló konstruktora* (copy constructor) és *értékadó operátora* (assignment operator). Ahol ez nem lehetséges, vagy megtiltjuk az értékadás használatát (ADA *private limited* típus) vagy valami "szokásos" függvényt (pl. Java **clone** metódus) hozunk létre. A Java nyelv **Cloneable** és a C# **ICloneable** interfésze ez utóbbi módszert támogatja, de erősen vitatott (Java C#) módon.

Konverziók

A kifejezések kiértékelésekor egyes esetekben az operandusok egyike, vagy mind konvertálódhat más típussá.

- Értékadás, változó inicializálás, paraméterátadás és **return** utasításkor konverzió történik a cél típusra.
- **Aritmetikai konverziók** történnek a *szélesebb* számábrázolású típusok felé:

  char –> short –> int –> long –> long long

  előjeles egészek –> előjelnélküli egészek

  egészek –> float –> double –> long double

  tömb –> első elemre mutató pointer

A konverziók bonyolult és széles skálája a szabványban és a C könyvekben részletesen le van írva.

Kifejezések kiértékelése

Bár a kifejezések *értelmezését* egyértelműen meghatározza a precedencia és az asszociativitás, a kifejezések *kiértékelésének* mikéntjét bizonyos keretek között szabadon meghatározhatja a fordítóprogram.

Mit ír ki az alábbi program:

```
1    #include <stdio.h>
2    int main()
3    {
4      int i = 1;
5      printf( "i = %d, ++i = %d\n", i, ++i );
6      return 0;
7    }
```

```
$ ./a.out
i = 2, ++i = 2    # más platformon i = 1, ++i = 2 is lehet
```

A fenti kifejezés hibás, *nemdefiniált viselkedésű* (undefined behavior) mert **i** és **++i** ugyanazt a memóriaterületet éri el és egyikük módosítja is azt. Ha két kifejezés kiértékelése ugyanazt a memória-területet éri el és legalább az egyik módosítja is azt, akkor **konfliktusban vannak** (conflicting). Erősen leegyszerűsítve, ahhoz, hogy a programok helyes viselkedését biztosítsuk, az ilyen konfliktusban levő kifejezéseket el kell választanunk ún. **szekvencia pontokkal** (sequence point). A szekvencia pont garantálja, hogy az előzőleg elkezdett kiértékelések befejeződjenek a szekvencia pontig és a rákövetkező kifejezések csak a szekvencia pont után kezdődjenek el. Így a kiértékelések nem kerülnek konfliktusba. A precíz leírás a C szabvány 5.1.2.3 pontja alatt olvasható.

Az *utasítások* eleje és vége szekvencia pont. Ezen kívül van néhány *operátor*, amelyik maga is szekvencia pontként viselkedik. Ilyenek

1. a rövidzáras logikai operátorok (&& és ||)
2. a feltételes operátor feltételének a kiértékelése (? :)
3. a vessző operátor (,)

Hasonlóan, amikor egy *függvényt meghívunk*, akkor az összes paramétere kiértékelődik, *mielőtt* a függvény törzsének végrehajtása elkezdődne. Ugyanakkor a paraméterek kiértékelésének egymás közötti sorrendje nem meghatározott.

```
1   #include <stdio.h>
2   int f()
3   {
4     printf("f\n");
5     return 2;
6   }
7   int g()
8   {
9     printf("g\n");
10    return 1;
11  }
12  int h()
13  {
14    printf("h\n");
15    return 0;
16  }
17  void func()
18  {
19    printf("(f() == g() == h()) == %d", f() == g() == h());
20  }
21  int main()
22  {
23    func();
24    return 0;
25  }
```

```
$ gcc -ansi -pedantic -Wall f.c
f.c: In function 'func':
f.c:20:44: warning: suggest parentheses around comparison in operand
of '==' [-Wparentheses]
printf("func: (f() == g() == h()) == %d\n", fpar == gpar == hpar);
                                                    ^
$ ./a.out
f
g
h
func: (f() == g() == h()) == 1
$
```

A fenti példában a kifejezés *jelentését* egyértelműen meghatározza a precedencia és az asszociativitás szabály. Ugyanakkor az egyes függvények meghívási sorrendjéről a fordító szabadon dönthet. Más fordítóprogramok, vagy akár ugyanaz a fordító más platformokon más sorrendet eredményezhet.

A hiányzó szekvencia pont súlyos hibát okozhat a programunkban. A lenti programban a 11. sorban az **i** változó két elérése (köztük az **i++** módosító) konfliktusos akció, ezért ez a program *nemdefiniált viselkedésű* (undefined behavior). A nemdefiniált viselkedésű programok hibásak, még akkor is, ha egyes platformokon lefutnak. Könnyen lehet, hogy a hiba csak akkor jön elő, ha egy másik fordítóval fordítjuk a programot.

```
1   /*
2    * BAD!
3    */
4   #include <stdio.h>
5   int main()
6   {
7     int t[10];
8     int i = 0;
9     while( i < 10 )
10    {
11      t[i] = i++;
12    }
13    for ( i = 0; i < 10; ++i )
14    {
15      printf("%d ", t[i]);
16    }
17    return 0;
18  }
```

```
$ gcc -ansi -pedantic -Wall -W  f.c
f.c: In function ‘main’:
f.c:9:13: warning: operation on ‘i’ may be undefined [-Wsequence-
point]
   t[i] = i++;
            ^
$ ./a.out
613478496 0 1 2 3 4 5 6 7 8
$
```

A helyes megoldás:

```
1   /*
2    * OK
3    */
4   #include <stdio.h>
5   int main()
6   {
7     int t[10];
8     int i = 0;
9     while( i < 10 )
10    {
11      t[i] = i;
12      ++i;
13    }
14    for ( i = 0; i < 10; ++i )
15    {
```

```
16            printf("%d ", t[i]);
17          }
18          return 0;
19        }
```

Utasítások, vezérlési szerkezetek

Az utasítások és vezérlési szerkezetek az imperatív programozási nyelvek alapvető elemei. Ezek segítségével írjuk le, *hogyan* szeretnénk a programot végrehajtani.

Kifejezés utasítás

Egy kifejezés az azt követő pontosvesszővel (;) egy kifejezés utasítást (expression statement) képez. Például a

```
printf("Hello world\n")
```

kifejezés típusa **int** értéke **12** (ugyanis a printf visszatérő értéke a kiírt karakterek száma). Ha pontosvesszőt teszünk utána, akkor utasítást kapunk:

```
printf("Hello world\n");
```

Üres utasítás

Az üres utasítás (null statement) hatás nélküli (bár kaphat cimkét).

```
1         if ( x < 10 )
2           ;
3         else
4           printf("else branch");
```

Összetett utasítás

Az összetett utasítás (compound statement) vagy blokk utasítás arra szolgál, hogy több utasítást összefogjon.

```
1       if ( x < 10 )
2       {
3         ;
4       }
5       else
6       {
7         printf("compound statement");
8         printf("in the else branch");
9       }
```

Sok véletlen hibát elkerülhetünk, ha a vezérlési szerkezetekben mindig kirakjuk a **{ }** kapcsos-zárójeleket, akkor is, ha csak egyetlen utasítást szeretnénk végrehajtani.

Elágazás

Az **if** elágazásnak két formája van.

```
if (expression) statement
if (expression) statement1; else statement2;
```

Az **if** kifejezés feltételét kötelező zárójelbe írni, ahogy azt a **switch while** és **for** esetében is. Az utasítások lehetőleg legyenek összetett utasítások. Az **if** utasítás esetében mindig érdekes kérdés, hogy hova tartoznak a *lógó* (dangling) **else** utasítások. A C-ben és sok más nyelvben az **else** a hozzá szintaktikusan legközelebbi **if**-hez tartozik.

```
1       if ( x < 10 )
2         if ( y > 5 )
3           printf("x < 10 and y > 5");
4       else
5         printf("x < 10 and y <= 5");
```

ekvivalens az alábbival:

```
1       if ( x < 10 )
2       {
3         if ( y > 5 )
4           printf("x < 10 and y > 5");
5         else
6           printf("x < 10 and y <= 5");
7       }
```

és eltér ettől:

```
1       if ( x < 10 )
2       {
3         if ( y > 5 )
4           printf("x < 10 and y > 5");
5       }
6       else
7         printf("x >= 10");
```

A *Pythonban* persze a tabulálás jelöli ki a struktúrát. A C-ben nincsen *elseif* vagy *elif*, de az else ág egyetlen utasításaként írhatunk egy újabb **if** utaítást. Ennek hatása hasonló, mintha elseif-ünk lenne, (kivéve persze, ha az egyik feltétel kiértékelésének olyan mellékhatása van, ami befolyásol egy másik feltételt, de az ilyen konstrukciókat inkább kerüljük).

```
1         if ( x < 10  &&  y > 5 )
2         {
3           printf("x < 10 and y > 5");
4         }
5         else if ( x < 10  &&  y <= 5 )
6         {
7           printf("x < 10 and y <= 5");
8         }
9         else if ( x >= 10  &&  y > 5 )
10        {
11          printf("x >= 10 and y > 5");
```

```
12      }
13      else if ( x >= 10  &&  y <= 5 )
14      {
15        printf("x >= 10 and y <= 5");
16      }
17      else
18      {
19        printf("impossible");
20      }
```

Szelekciós utasítás

A **switch** utasítás egy alternatív elágazái forma, ahol az elágazást egy kifejezés különböző értékei alapján hajtjuk végre. A switch formája:

```
switch (expression) statement
```

Az utasítás szinte mindig egy blokk, melyben **case** címkével ellátott utasítások szerepelnek. A címkék értékének fordítási időben megadottnak és egyedinek kell lennie, és azt a fordító ellenőrzi is.

```
1       int day_of_week;
2       //...
3       switch ( day_of_week )
4       {
5         default: printf("Undefined"); break;
6         case  2: printf("Monday");    break;
7         case  3: printf("Tuesday");   break;
8         case  4: printf("Wednesday"); break;
9         case  5: printf("Thursday");  break;
10        case  6: printf("Friday");    break;
11        case  1: /* fallthrough */
12        case  7: printf("Week-end");  break;
13      }
```

A cimkéket úgy tekinthetjük, mint célpontokat, ahová odaugrik a vezérlés, ha értékük megegyezik a feltételben megadott értékkel. Onnan a vezérlés a megadott utasításoknak megfelelően, szekvenciálisan folytatódik, amíg el nem érünk egy **break** utasításhoz. Onnan a vezérlés a switch-et követő utasítással folytatódik.

Ha nincsen **break** utasítás, akkor a vezérlés *rácsorog* a következő cimkét tartalmazó utasításra. Ez általában nem jó programozási stratégia, de esetenként ezt használjuk a cimkék *csoportosítására*. Ilyenkor ajánlott ezt a szándékunkat pl. kommentben jelezni.

Ha egyetlen címke sem egyezik meg a feltételben megadott értékkel, és van **default** címke, akkor a vezérlés oda adódik át. Ettől eltekintve a default címke viselkedése megegyezik a többi címkéjével. Ha nincsen default címke sem, akkor a vezérlés a switch utáni utasítással folytatódik. Ha egyetlen címkén sem csorgunk túl, akkor az egyes címkék és a default címke sorrendje közömbös.

While ciklus

A C nyelvben többféle módon szervezhetünk ciklust. Az egyik legalapvetőbb konstrukció a **while** ciklus.

```
while ( expression ) statement
```

A while ciklus *először* ellenőrzi a *ciklusfeltétel* kifejezést, és addig hajtja végre a *ciklusmagot*, ameddig a feltétel igaz. A while ciklusban nekünk kell gondoskodni arról, hogy a feltétel előbb vagy utóbb hamissá váljon.

```
1         struct list_type
2         {
3           int       value;
4           list_type *next;
5         };
6         // ...pt-expr
7         list_type *ptr = first;
8         while ( NULL != ptr  )
9         {
10          printf( "%d ", ptr->value);
11          ptr = ptr->next;
12        }
```

Do-while ciklus

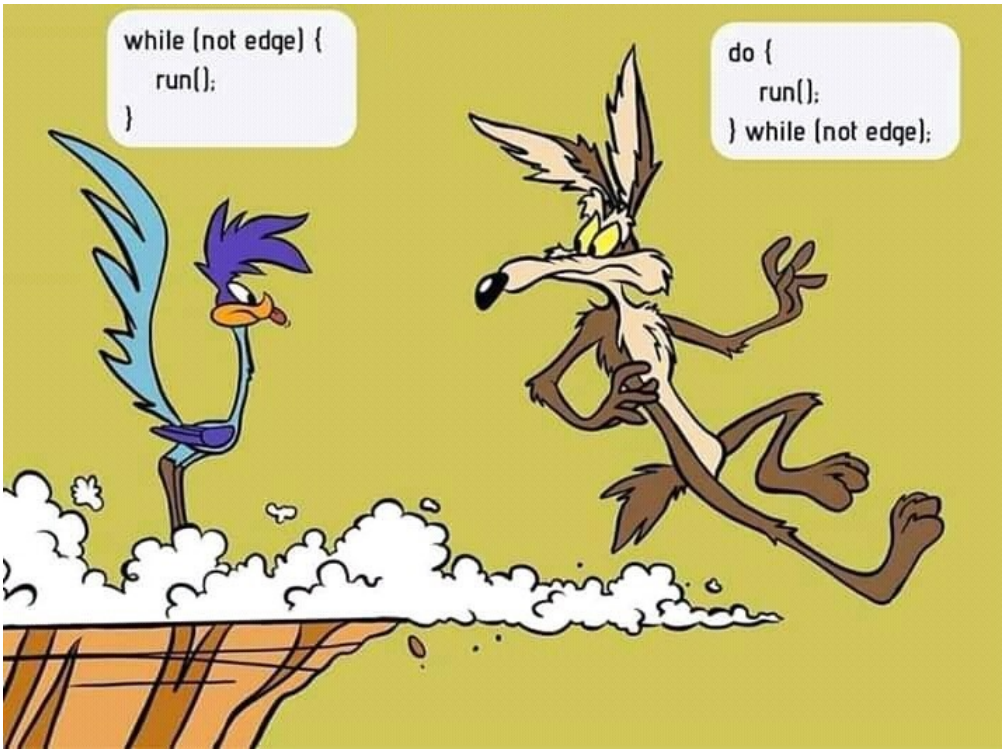
A do-while ciklus ún. hátul-tesztelő ciklus. Ez azt jelenti, hogy a ciklusmagot egyszer mindenképpen végrehajtjuk, és csak utána ellenőrizzük a feltételt.

```
do statement while ( expression ) ;
```

Figyeljük meg a feltétel-kifejezés zárójelét lezáró pontosvesszőt. A do-while utasítás ekvivalens a következő konstrukcióval:

```
statement
while ( expression )
  statement
```

A do-while konstrukciót néha alkalmazzák, amikor az első ciklusvégrehajtás előtti ellenőrzést ki akarják spórolni pl. hatékonysági okokból.



For ciklus

A for ciklus az egyik leggyakrabban előforduló ciklusfajta. Kétféle formája van:

```
for ( opt-expr-1 ; opt-expr-2 ; opt-expr-3 ) statement
for ( declaration; opt-expr-2 ; opt-expr-3 ) statement  (C99 óta)
```

ahol

1. *opt-expr-1* egy opcionális (elhagyható) kifejezés, ami a ciklusváltozó kezdeti értékbeállítására szolgál és a legelső ciklusvégrehajtás előtt hajtódik végre. A C99 verzió óta ezt a kifejezést helyettesíthetjük egy deklarációval. Az itt deklarált ciklusváltozó láthatósága nem terjed túl a cikluson.
2. *opt-expr-2* egy opcionális feltétel, ami minden ciklusmag végrehajtása *előtt* kiértékelődik, és a ciklusmag csak akkor hajtódik végre, ha ennek a kifejezésnek értéke igaz. Ha ezt a kifejezést elhagyjuk, akkor értékét *mindig igaznak* tekintjük.
3. *opt-expr-3* egy opcionális kifejezés, ami mindig kiértékelődik a ciklusmag után. Ez a kifejezés gyakran arra szolgál, hogy a ciklusváltozót módosítsa.

Az alábbi for ciklus

```
for ( e1 ; e2 ; e3 ) s;
```

nagyjából (de nem teljesen) azonos a következő while ciklussal:

```
{
  e1;
  while ( e2 )
  {
    s;
    e3;
```

```
  }
}
```

A három opcionális kifejezés bármelyikét elhagyhatjuk. A középső elmaradása olyan, mintha állandóan igaz kifejezést írnánk. A (látszólag) végtelen ciklus egy alakja:

```
for( ; ; ) statement
```

Ezt a ciklust még mindig elhagyhatjuk a **return** vagy a **break** utasítással.

A C99 óta lehetséges az inicializáló kifejezést helyettesíteni egy ciklusváltozó létrehozásával és inicializálásával.

```
1       for ( int i = 0; i < 10; ++i )
2       {
3         printf( "%d ");
4       }
5       // i is not visible here.
```

A break és a continue utasítások

A **break** utasítást nemcsak a **switch**-ben, hanem bármely cikluson belül is alkalmazhatjuk. Hatására a ciklusból azonnal kilépünk, és a következő utasítással folytatjuk a programot.

```
1     int t[10];
2     // ...
3     for ( int i = 0; i < 10; ++i )
4     {
5       if ( t[i] < 0 )
6       {
7         printf( "negative found");
8         break;
9       }
10      printf("do something with non-negatives");
11    }
12    // break jumps to here
```

A **continue** utasítás átugorja a ciklusmag hátralévő részét és a vezérlés a ciklusmag végére ugrik. Ezután a while és do-while ciklusban a feltétel ellenőrzése, a for ciklusban az *opt-expr-3* majd a feltétel kiértékelése következik.

```
1     int t[10];
2     // ...
3     for ( int i = 0; i < 10; ++i )
4     {
5       if ( t[i] < 0 )
6       {
7         printf( "negative found");
8         continue;
9       }
10      printf("do something with non-negatives");
11      // ...
```

```
12      // continue jumps to here
13    }
```

Return utasítás

A **return** visszatér a kurrens függvény végrehajtásából a hívó függvénybe. A *main* függvény esetében a **return** hatására a program végrehajtása befejeződik.

```
return;
return expr;
```

Egy függvényben több return utasítás is szerepelhet. Ha a függvény visszatérő típusa nem **void** akkor a return argumentuma a függvény visszatérő típusára konvertálódik.

```
1     int find_first_negative( int t[], int length)
2     {
3       for ( int i = 0; i < length; ++i )
4       {
5         if ( t[i] < 0 )
6         {
7           printf( "negative found");
8           return t[i];
9         }
10      }
11      return 0;
12    }
```

Goto utasítás

Feltétel nélküli ugró utasítás. Csak az adott függvényen belülre ugorhatunk.

```
goto label;
/* ... */
label: statement
```

ahol *label* egy azonosító. Ne használjunk **goto** utasítást.

Homepage of Dr. Zoltán Porkoláb

Home

Archive

Teaching

Timetable

Bolyai College

C++ (for mathematicians)

Imperative programming (BSc)

Multiparadigm programming (MSc)

Programming (MSc Aut. Sys.)

Programming languages (PhD)

Software technology lab

Theses proposals (BSc and MSc)

Research

CodeChecker

CodeCompass

Templight

Projects

Conferences

Publications

PhD students

Affiliations

Dept. of Programming Languages and Compilers

Ericsson Hungary Ltd

Imperatív programozás 5.

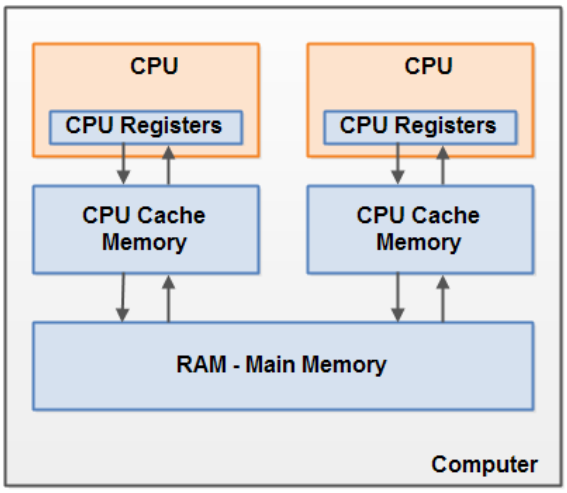
Memóriakezelés, tömbök, pointerek

Az imperatív programozási nyelvekben a programok állapotát módosítjuk utasításainkkal. A program állapota a számítógép memóriájában tárolódik.

Memória

(Egy alapvető cikk Ulrich Dreppertől, amit a programozónak hasznos tudnia a számítógép memóriájáról.)

A mai számítógépek memóriája hierarchikus szerkezetű. Az egyes processzorok saját **gyorsítótárral** (**cache**) rendelkeznek (több szinten is). A modern architektúrák a memória írásakor/olvasásakor felhasználják a gyorsítótárat.



A központi memóriába történő írás/olvasás relatív lassú művelet. Ha a processornak mindig be kéne várni az írás/olvasás eredményét, akkor az értékes processoridő nagyrésze várakozással telne el. A gyorsítótár segítségével a gyakran vagy éppen nemrég használt adatok “kéznél vannak”, így azokkal sokkal gyorsabban lehet műveleteket végezni.

```
Latency Comparison Numbers (~2012)
----------------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5 ns
L2 cache reference                           7 ns
Mutex lock/unlock                           25 ns
Main memory reference                      100 ns
Compress 1K bytes with Zippy             3,000 ns       3 us
Send 1K bytes over 1 Gbps network       10,000 ns      10 us
Read 4K randomly from SSD*             150,000 ns     150 us
Read 1 MB sequentially from memory     250,000 ns     250 us
Round trip within same datacenter      500,000 ns     500 us
Read 1 MB sequentially from SSD*     1,000,000 ns   1,000 us   1 ms
Disk seek                           10,000,000 ns  10,000 us  10 ms
```

```
Read 1 MB sequentially from disk     20,000,000 ns  20,000 us  20 ms
Send packet CA->Netherlands->CA     150,000,000 ns 150,000 us 150 ms
```

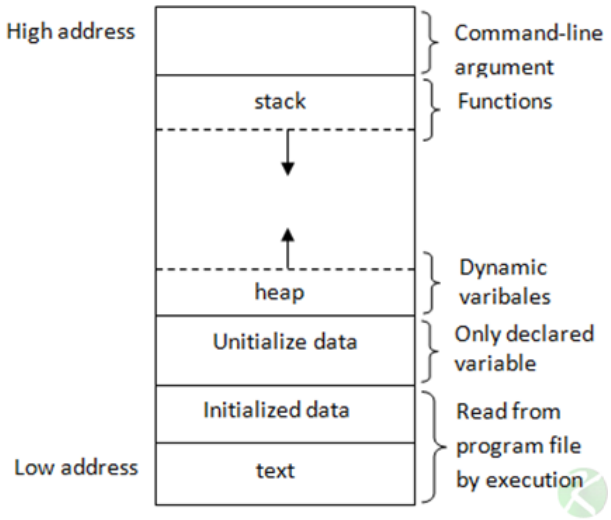
(innen: https://blog.morizyun.com/computer-science/basic-latency-comparison-numbers.html)

A legtöbb operációs rendszer elfedi előlünk a számítógép belső memóriaszerkezetét és egy egyszerűsített modellt ad a programozó számára. Leggyakrabban a memóriát egy összefüggő bájttömbként képzelhetjük el.

Amikor egy program betöltődik a memóriába (az operációs rendszer elkezdi futtatni), akkor az lefoglal bizonyos területet a memóriából. Ez lesz a program **címtere** (**address space**). Az operációs rendszer, ha többfelhasználós, többtaszkos, akkor figyel arra, hogy a program ne érje el más programok címterét. Ha egy program ilyet tenne, akkor az operációs rendszer közbeavatkozik és megszakítja a progam futását.

A program futása során kérhet újabb memóriát az operációs rendszertől vagy vissza is adhat számára területet. A program befejeződésekor az operációs rendszer felszabadítja a teljes program által használt tárterületet.

A C programok címtere általában a következőképpen néz ki:



A memória alján levő részeket a programfájlból hozza létre az operációs rendszer, itt helyezkednek el a **globális változók** (inicializált vagy inicializálatlan értékekkel). A memória tetején a parancssori argumentumok, a **végrehajtási verem** (stack) és a **szabad memória** (heap) helyezkedik el.

Amikor egy változót deklarálunk a programunkban általában úgy tekintünk rá, mint egy adatra, amelynek értéket adhatunk, módosíthatjuk, kiolvashatjuk az értékét. A fordító program a változókat egy kezdő memóriacímként kezeli, és a változó típusából tudja a hosszát ill. hogy milyen műveleteket lehet végezni rajta.

```
1   void f()
2   {
3     int  i = 1; /* egy int típusú változó, tartalma 1 */
4     int  j;     /* egy int változó, tartalma nem definiált */
5
6     j = i;  /* sizeof(int) bájt másolása i címéről j címére */
7   }
```

A változók mérete a típusuktól függ: egy **char** típikusan 1 bájt, egy **short** gyakran 2 bájt, egy **int** mai architektúrákon 4 bájt, de a C nyelvben a méretek nincsen fixen

rögzítve, csak a szükséges minimum méretek. Az adott platformon a méreteket a **sizeof()** operátorral tudjuk lekérdezni.

Pointerek

A **mutató** (**pointer**) egy olyan típus, amelyik egy változó **címét** (address) tárolja. A pointerek **típusosak**, azaz amikor létrehozok egy pointer változót, akkor meg kell mondanom, hogy milyen típusú változó címét akarom eltárolni benne, milyen *típusra mutat* a pointerem.

```
1   void f()
2   {
3     int    *ip;  /* egy int-re mutató pointer */
4     double *dp;  /* egy double-ra mutató pointer */
5     char   *cp;  /* egy char-ra mutató pointer */
6     void   *vp;  /* egy tetszőleges bájtra mutató pointer */
7   }
```

A pointerek egy platformon általában azonos méretűek, pl. 32 bites architektúrákon 4 bájt, 64 biteseken 8 bájt. A pointereket úgy képzelhetjük el, hogy a tartalmuk egy memóriacím, a mutatott változó kezdőcíme. A gyakorlati implementáció ettől eltérhet (pl. intel 286-os sorozat).

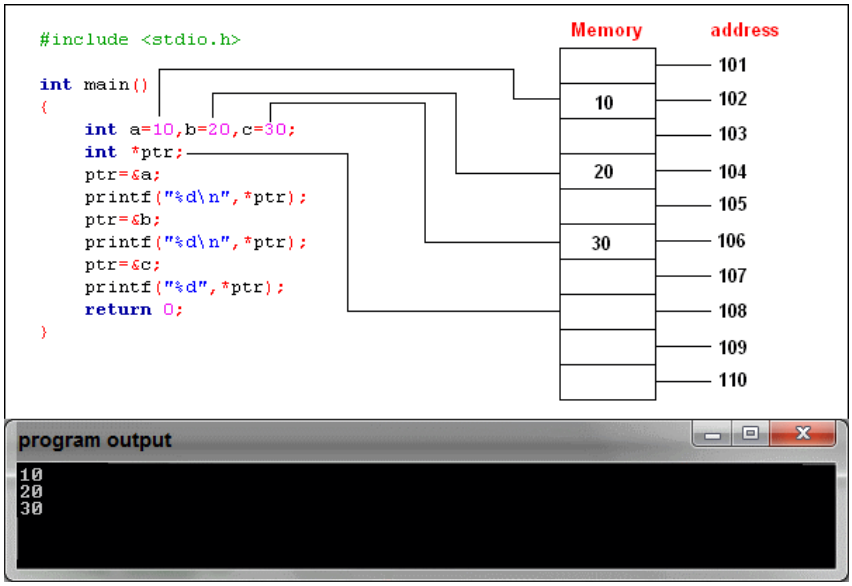
A pointerekkel kapcsolatban két alapvető operátort alkalmazhatunk.

A **cím** (&) operátor. Egy változóra alkalmazva az adott változó tárcímét adja meg, azaz a rá mutató pointer értéket.

Az **indirekció** (*) operátor. Egy pointerre alkalmazva a pointer által mutatott tárterületet jelenti. A *ptr kifejezéssel minden műveletet elvégezhetünk, amit a pointer által mutatott típussal elvégezhetünk.

```
1   void f()
2   {
3     int  i = 1; /* egy int típusú változó, tartalma 1 */
4     int  j;     /* egy int változó, tartalma nem definiált */
5
6     int  *ip;  /* ip egy pointer int-re, inicializálatlan */
7     ip = &i; /* ip most az i változóra mutat */
8     j = *ip; /* sizeof(int) bájt másolása i címéről j címére*/
9   }
```

Egy példa pointereket alkamazó programra:



NULL pointer

A pointer értékek között van egy speciális érték, a **null pointer**, amit a **NULL** makróval fejezünk ki. A null pointer értéke egyetlen valós változó címének sem felel meg. A null pointer azt jelképezi, hogy a pointerünk most éppen sehova sem mutat.

```
1     int t[] = { 1, 3, 5, ..... };
2
3     int *find( int w)
4     {
5
6       for ( int i = 0; i < sizeof(t)/sizeof(t[0]); ++i)
7       {
8         if ( w == t[i] )
9           return &t[i];
10      }
11      return NULL;
12    }
13
14    int main()
15    {
16      int *ptr = find(11);
17
18      if ( NULL == ptr )
19      {
20        printf( "not found\n");
21      }
22      else
23      {
24        printf( "found\n");
25        ++ *ptr;
26      }
27    }
```

Ha a pointer **NULL** értékű, akkor **tilos** az indirekciót alkalmazni rá. A fordítóprogram ritkán tud ilyen hibákat fordítási időben felfedezni, viszont futási időben **futási hiba** fog bekövetkezni, ami gyakran a program elszállását eredményezi.

Műveletek pointerekkel

Pointer változóknak **értékül adhatunk** ugyanolyan típusú pointer értékeket. Ez a gyakorlatban azt jelenti, hogy a két pointer most ugyanarra a tárterületre fog mutatni.

Két pointer értéket össze lehet hasonlítani az *egyenlőség* (**==**) és a *nem egyenlő* (**!=**) operátorral, illetve egy pointer értéket a **NULL** pointerrel. Két pointer érték pont akkor egyenlő, ha a pointerek ugyanoda mutatnak. A **NULL** pointer érték csak saját magával lesz egyenlő.

```
1     void f()
2     {
3       int  i = 1;
4       int  j = 2;
5
6       int  *ip = &i;  /* ip az i változóra mutat */
7       int  *jp = &j;  /* jp a  j változóra mutat */
8
9       if ( ip == jp )   { ... }  /* hamis */
10      if ( ip == NULL ) { ... }  /* hamis */
11      if ( jp == NULL ) { ... }  /* hamis */
12
13      ip = jp;    /* most ip is j-re mutat */
14      if ( ip == jp )   { ... }  /* igaz */
15    }
```

Abban, de csak abban az esetben, ha *mindkét pointer ugyanannak a tömbnek az elemeire mutat*, akkor szabad a *kisebb*, stb. relációs operátorokat is alkalmazni a pointerekre (<, <=, >, >=) és egy pointer akkor nagyobb a másiknál, ha a *nagyobb indexű* elemre mutat. A kisebb-nagyobb összehasonlítás tehát nem a memóriacímeket hasonlítjuk össze, hanem a tömb indexeket.

Különböző tömbök esetében, vagy különböző típusú pointerek esetében nem használhatjuk a relációs műveleteket.

Pointerre mutató pointerek

Mivel a pointer típusú változók is valahol elheyezkednek a memóriában, ezeknek a tárterületeknek is van címe, ezért létezhetnek pointerre mutató pointerek is. A pointerre mutató pointerek is típusosak, azaz nem mindegy, hogy egy int-re vagy egy double-ra mutató pointerre mutatunk.

Ezeknek van gyakorlati haszna a C-ben, például ha egy függvény egy paramérerül kapott pointert szeretne módosítani, akkor a pointer címét adjuk át.

Tömbök

A tömbök szigorúan összefüggő memóriaterületek, ahol ugyanannak a típusnak valahány számú elemét foglaljuk le.

```
1     int it[10];  /* 10 darab int egymás után */
2
3     void f()
4     {
5       double dt[5];  /* 5 darab double egymás után */
```

```
6
7       int it2[6] = {1,2,3,4,5,6};  /* tömb + inicializálás */
8       int it3[]  = {7,8,9};        /* 3 elemű tömb */
9
10      char h1[] = { 'H','e','l','l','o','\0' };  /* 6 char */
11      char h2[] = "hello";  /* ugyanaz, mint h1[] esetében */
12
13      assert( sizeof(h1) == 6 );
14      assert( sizeof(h2) == 6 );
15    }
```

Egy tömböt is inicializálhatunk létrehozásakor. Ilyenkor figyelni kell arra, hogy az inicializációs lista annyi elemű legyen, mint a tömbünk. Ezt egyszerűen elérhetjük, ha az inicializált tömbnek nem adunk méretet: ilyenkor a fordítóprogram megszámolja az elemeket és akkora tömböt foglal le. A tömbökre is működik a **sizeof** operátor, a legfoglalt össz-bájtmennyiséget adja.

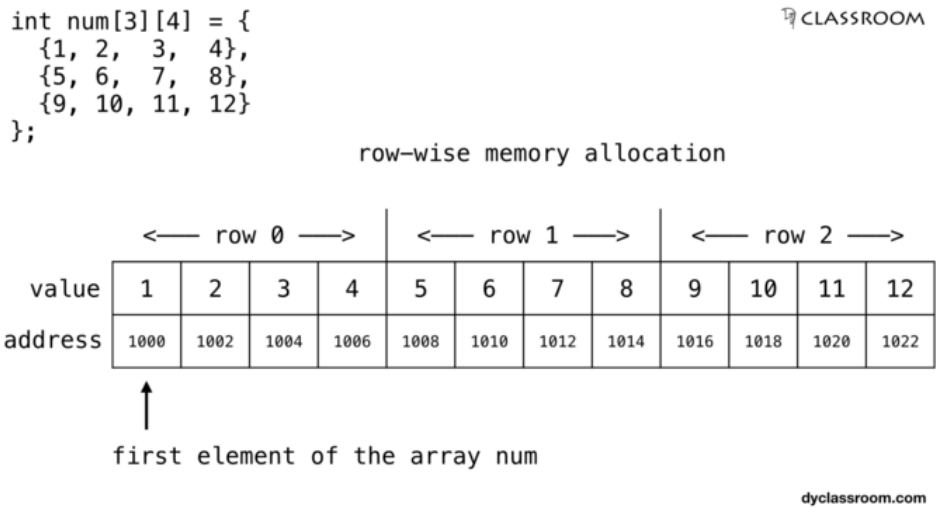
A **tömböket 0-tól indexeljük**. Egy N elemű T tömb elemei: T[0], T[1], … T[N-1]. Szokásos technika a tömbök méretének kinyeréséhez a **sizeof(t) / sizeof(t[0])** alkalmazása, ami típus- és méretfüggetlenül a tömb elemszámát adja meg.

Tömbökkel nem végezhetünk műveleteket, csak tömbelemekkel. Így pl. nem létezik értékadás tömbök között. Ha ilyet szeretnénk csinálni, ciklussal át kell másolni az egyik tömb összes elemét a másikba.

A tömböket az első elemükre mutató pointerként adjuk át függvényhíváskor.

Többdimenziós tömbök

A többdimenziós tömböket *sorfolytonos* módon ábrázoljuk:



A többdimenziós tömböket úgy is felfoghatjuk, mint egy egydimenziós tömb, aminek minden eleme is egy-egy tömb. A többdiemnzió tömbök elemeinek elérése is ez utóbbi filozófiát követi:

```
1     void f()
2     {
3       int num[3][4] = {
4         {1, 2,  3,  4},
5         {5, 6,  7,  8},
6         {9, 10, 11, 12}
```

```
7       };
8
9       for ( int i = 0; i < 3; ++i )
10      {
11        for ( int j = 0; j < 4; ++j )
12        {
13          printf( "%d ", num[i][j]); /* NEM num[i,j] !! */
14        }
15        printf("\n");
16      }
17    }
```

Pointerek és tömbök kapcsolata

A pointerek és a tömbök között logikai kapcsolat van a C nyelvben. Tekintsünk egy tömböt, aminek az első elemére ráállítunk egy pointert.

```
1     void g(double *);
2
3     void f()
4     {
5       double t[] = { 1.0, 2.0, 3.0, 4.0};
6       double *dp;
7
8       dp = &t[0];  /* dp a t tömb első elemére mutat */
9       dp = t;      /* == dp = &t[0] */
10      g(t);        /*˙== g( &t[0] ) */
11
12      assert ( *dp == t[0] );  /* dp t[0]-ra mutat */
13      assert (sizeof(t) == 4*sizeof(double));
14    }
15
16    void g(double par[]) /* valójában g(double *par) */
17    {
18      *par   = -1.0;   /* a t tömb 0-ás indexű eleme */
19      par[1] = -2.0;   /* a t tömb 1-es indexű eleme */
20      assert (sizeof(par) == sizeof(double*));
21    }
```

Ha egy **tömb nevét** egy kifejezésben, pl. értékadásban vagy paraméterátadásban használjuk, akkor a tömb neve automatikusan az **első elemre mutató** pointer értékké konvertálódik. Ez történik függvényhíváskor is, azaz, egy tömböt mindig az első elemére mutató pointerként adunk át függvényhíváskor!

Ha egy **függvényparamétert tömbnek deklarálunk**, azt a fordítóprogram automatikusan **pointernek** tekinti.

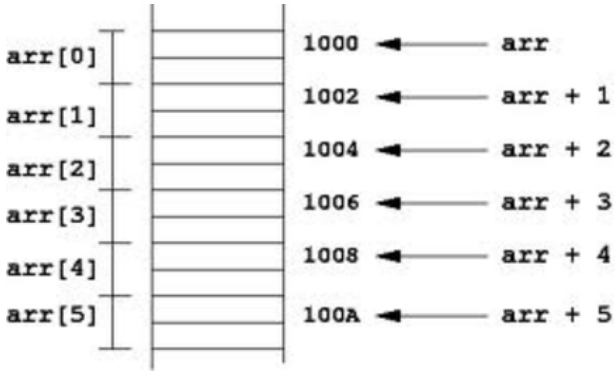
Pointer aritmetika

Ha egy pointer egy tömb valamely elemére mutat, akkor szabad hozzáadni, vagy kivonni belőle egy egész számot (ha az így kapott pointer érték még mindig a tömb valamely elemére mutat).

A művelet definíciója: ha a **ptr** pointer az **i** indexű kombelemre mutatott, akkor **ptr + k** a tömb **i + k** indexű elemére fog mutatni (feltéve, ha van ilyen indexű elem).

```
1   void g(double *);
2
3   void f()
4   {
5     double t[] = { 1.0, 2.0, 3.0, 4.0};
6     double *dp;
7
8     dp = t;       /* == dp = &t[0] */
9
10    assert ( dp+1 == &t[1] );  /* dp+1 t[1]-re mutat */
11    assert ( dp+3 == &t[3] );  /* dp+3 t[3]-ra mutat */
12
13    assert (  dp+k  == &t[k] );  /* 0 <= k < 4 */
14    assert (*(dp+k) ==  t[k] );  /* 0 <= k < 4 */
15
16  }
```

Valójában a C nyelv a tömbök indexelését teljesen visszavezeti a pointer aritmetikára. Mivel a tömb neve maga is mutatóvá konvertálódik, a tömb nevére is alkalmazhatjuk a pointer aritmetikát:



Ennek megfelelően, a pointerekre is alkalmazhatjuk a tömb indexelés operátort. Valójában, akár tömb névre, akár pointerre alkalmazzuk az index operátort, ugyanaz történik: pointer aritmetika és utána indirekció.

```
1     void g(double *);
2
3     void f(int k)
4     {
5       double t[] = { 1.0, 2.0, 3.0, 4.0};
6       double *dp;
7
8       dp = t;       /* == dp = &t[0] */
9
10      assert ( t[k]  == *(t+k)  );
11      assert ( dp[k] == *(dp+k) );
12    }
```

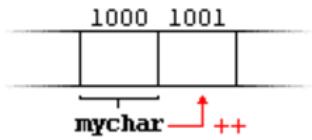
Pointerből kivonás értelemszerűen valamely kisebb indexű elemre fog hivatkozni. Végül, lehetséges két azonos típusú, azonos tömb elemeire mutató pointer

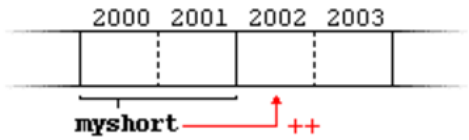
különbségét képezni. Ez a pointerek által mutatott tömbindexek (előjeles) különbsége lesz.

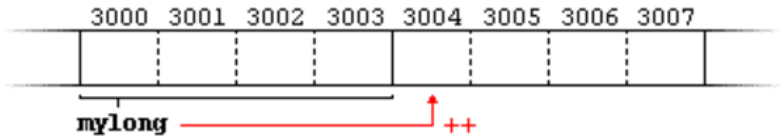
```
1       void g(double *);
2
3       void f()
4       {
5         double t[] = { 1.0, 2.0, 3.0, 4.0};
6         double *dp;
7         double *dq;
8
9         dp = t+1;       /* == dp = &t[1] */
10        dq = t+3;       /* == dq = &t[3] */
11
12        assert ( (dp + 2)  ==  dq );
13        assert ( (dp - dq) ==  -2 );
14      }
```

Fontos tehát észrevenni, hogy a pointer aritmetika nem bájtokban, hanem a mutatott típus darabszámában számol. A fenti példában **dp** és **dq** szinte biztos, hogy nem 2 bájtnyira mutat egymástól, hanem **sizeof(double)*2** bájtra.

Ezért is fontos, hogy a pointereinket a megfelelő típusra definiáljuk. A pointer aritmetika esetében a +1 az nem egy bájt, hanem egy *egység*, amire a pointerünk mutat.

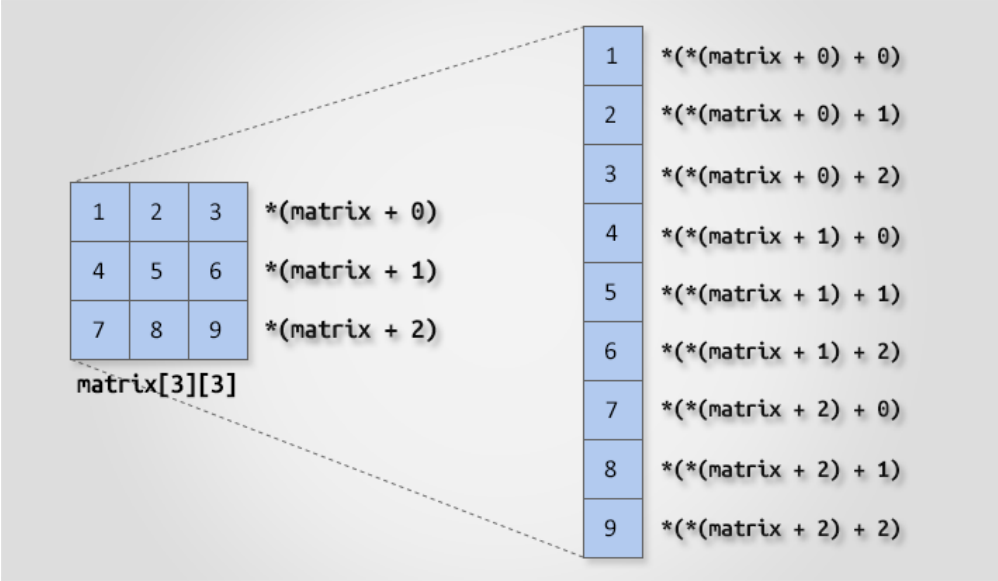






Fontos azt is megjegyezni, hogy a kisebb pointer nem feltétlenül a kisebb memóriacímre mutat, hiszen a tömbök ábrázolása implementációfüggő.

A többdimenziós tömbökre is igaz a pointeraritmetika, de figyeljünk a típusokra: tömbök tömbjéről van szó.



Mint láthatjuk, a C nyelvben a tömbök és a pointerek között van egyfajta logikai kapcsolat:

- A tömbök nevei kifejezésekben az első elemre mutató pointerré konvertálódnak
- A tömbre mutató pointerekre akalmazhatunk pointer aritmetikát
- Az index operátort egyaránt alkalmazhatjuk a tömbnevere és pointerekre

De ettől még **a pointerek és a tömbök nem ekvivalensek!**. A tömb több, azonos típusú elem számára lefoglalt folytonos tárterület. A pointer egyetlen objektum, ami egy másik tárterület hivatkozását (címét) tartalmazza.

Haladó feladat

Az alábbi két forrásfájlból (a.c és b.c) álló program lefordul, összeszerkesztődik, de futási időben elszáll. Mi lehet a hiba? Hogyan kell kijavítani?

```
1   /* a.c */
2   #include <stdio.h>
3
4   int t[] = {1,2,3};
5
6   void g( int *par);
7
8   void f( int *par)
9   {
10    printf("%d\n",par[1]);
11    printf("%d\n",t[1]);
12  }
13  int main()
14  {
15   int *p = t;
16   printf("%d\n",t[1]);
17   printf("%d\n",p[1]);
18   f(t);
19   g(t);
20  }
21
22
```

```c
/* b.c */
#include <stdio.h>

extern int *t;

void g( int *par)
{
 printf("%d\n",par[1]);
 printf("%d\n",t[1]); /* itt száll el a program */
}
```

VLA

Az eddigiekben a tömböket mindig fix, a fordításkor ismert konstans mérettel hoztuk létre. Ez a javasolt megoldás. A C nyelv azonban a C99 verzió óta megengedi a változó méretű tömbök (Variadic Lenght Array, **VLA**) létrehozását is.

```c
/* NE HASZNÁLJUNK VLA_t!!! */
void f()
{
  int n;
  printf("Add meg a tömb méretét: ");
  scanf("%d",&n);
  int t[n];  /* VLA - ne használjuk */
  /* tömbelemek: t[0]...t[n-1] */
}
```

Ez így kényelmesnek tűnik, de **számos érv van a VLA használata ellen**.

- A VLA-k használata bizonyítottan több hibát okoz és biztonsági sérülékenységekhez vezethet.
- A VLA bizonyos futási idejű lassulást okoz a fix méretű tömbök használatához képest.
- Bizonyos esetei (pl. struct-on belüli használata) nem megengedett és egyes fordítók nem támogatják.
- C++-ban sohasem volt és nem is lesz szabványos nyelvi elem.

A Linux kernelt 2018 végére kemény munkával megtisztították a VLA-któl, ahogy ez a cikk ismerteti. **Ti se használjatok VLA-t**.

Homepage of Dr. Zoltán Porkoláb

Home

Archive

Teaching

Timetable

Bolyai College

C++ (for mathematicians)

Imperative programming (BSc)

Multiparadigm programming (MSc)

Programming (MSc Aut. Sys.)

Programming languages (PhD)

Software technology lab

Theses proposals (BSc and MSc)

Research

CodeChecker

CodeCompass

Templight

Projects

Conferences

Publications

PhD students

Affiliations

Dept. of Programming Languages and Compilers

Ericsson Hungary Ltd

Imperatív programozás 6.

Deklarációk, láthatóság, élettartam

Az imperatív programozási nyelvekben két fontos szabályrendszer határozza meg a változók, függvények és típusok használatát: a **láthatóság** (scope) és az **élettartam** (life). A láthatóság helyett szokták a *hatókör* elnevezést is használni. A láthatóságot és élettartamot - hasonlóan más nyelvekhez - a C programozási nyelvben a **deklarációk** formája és helye határozza meg.

Deklaráció, definíció

Amikor egy *nevet* (azonosítót) bevezetünk egy programban, akkor a *statikus típusrendszerű* (lásd 2. előadás) nyelvek elvárják, hogy közöljük a fordítóprogrammal, hogy "mit gondoljon" erről az azonosítóról: pl. mi a **típusa**, vagy hol és mennyi ideig kívánjuk használni.

A deklarációk egy része konkrétan meg is határozza az illető objektumot: ezt nevezzük **definíciónak**. A *változók* esetén a definíció intézkedik a tárterület tényleges lefoglalásáról, a *függvények* esetén a paraméter lista és visszatérő érték típusa megadása mellett a konkrét függvénytörzs meghatározása is megtörténik, *típusok* esetén pedig az adatszerkezetet kell megadnunk.

A **deklaráció** azonban gyakorta nem jár együtt a definícióval. Ha pl. egy változót egy másik fordítási egységben (forrásfájlban) foglalnak le (azaz ott definiálják), de ebben a fájlban is akarjuk használni (írni, olvasni), akkor ebben a fordítási egységben is meg kell mondni a fordítóprogramnak, hogy mit gondoljon felőle. Azaz deklarálni kell. Ilyenkor a változóknak meg kell adni a típusát. A függvényeknél a visszatérő értékét és a paraméterlistáját (hogy pl. konverziók történjenek a paraméterátadáskor vagy visszatéréskor), de nem kell megadni, hogy mely konkrét utasítások lesznek végrehajtva, hiszen a függvény kódját a másik fordítási egység fordítja le.

Deklaráció formája

A deklarációk formája C nyelvben *tárolási-osztály típusnév deklarátor-lista*, ahol a *deklarátor-lista* egyszerűen vesszővel elválasztott *deklarátor*-ok listája. A **tárolási osztály** (storage class) egy olyan kulcsszó, ami a deklaráció jelentését befolyásolja, és az alábbi kulcsszavak egyike:

auto, **register**, **static**, **extern**, **typedef**

Mivel C-ben nem szükséges alkalmazásuk, ezért inkább kerüljük a **register** és **auto** használatát. A **register** egy optimalizációs ajánlás, amit a modern fordítóprogramok enélkül is megtesznek. Az **auto** kulcsszó pedig más jelentéssel bír C++-ban. A többi tárolási osztály használatára látunk majd példákat.

A *deklaráror* rekurzív formában van megadva:

- függvény esetében: *deklarátor (paraméter-lista)*
- mutató esetében: * *deklarátor*
- tömb esetében: *deklarátor [n]*
- egyébként egy azonosító: *azonosító*

Példák definíciókra:

```
1   int  i;      /* egész (int) változó definiálása */
2   int *pi;     /* egész típusra mutató pointer definiálása  */
3   int  t[10];  /* egy 10 egészt tartalmazó tömb definiálása */
```

```
4   int func1(void){...} /* par nélküli, int-el visszatérő fv */
5   int func2(int i, double d){...} /* uaz int és double par. */
```

Ezeket a deklarátorokat rekurzívan is lehet használni. A kivételek: függvények nem térhetnek vissza függvénnyel vagy tömbbel, és tömbök nem tartalmazhatnak függvényeket.

```
1   int **pi; /* egy egészre mutató pointer-re mutató pointer */
2   int  tt[10][20]; /* 10 db 20 elemű egész tömböt tart. tömb */
3   int *func3(void){...}/* par. nélküli int ptr visszatérő fv */
4   int *func4(int i, double d){...} /* int- és double par. fv */
```

Amennyiben kétértelműség állna fenn, akkor az operátorok precedenciája és a zárójelezés dönti el a deklaráció értelmét. Ugyancsak vigyázzunk arra, hogy pl. a mutató * jele a deklarátorhoz tartozik!

```
1   int  *ptr_arr[10];/* 10 elemű tömb int mutatókkal */
2   int (*ptr_to)[10];/* mutató 10 egészet tartalmazó tömbre */
3   int*  ptr1, ptr2; /* ptr1 mutató egészre, ptr2 viszont int */
```

A változóknak a definiciójuknál kezdőértéket is adhatunk, azaz *inicializálhatjuk* őket. Ez erősen ajánlott, hiszen így biztosan azt az értéket tartalmazzák, amit mi adtunk nekik.

```
1    int     i = 1;
2    double pi = 3.14;
3    int  *ptr = &i    /* pointer to int. i-re mutat */
4
5    int arr1[10] = {0,1,2,3,4,5,6,7,8,9};     /* nem ajánlott */
6    int arr2[10] = {0,1,2,3,4,5,6,7,8};  /* mert arr2[9] == 0 */
7    int arr3[] = {0,1,2,3,4,5,6,7,8}; /* ajánlott int arr3[9] */
8    int t[][3] = { {1,2,3}, {4,5,6} };         /* int t[2][3] */
9
10   char str1[] = {'H','e','l','l','o','\0'}; /* char str1[6] */
11   char str2[] = "Hello";                    /* char str2[6] */
12   char  *str3 = "Hello";         /* char * (pointer 'H'-ra) */
```

A globális, statikus élettartamú tárterületek 0-ra inicializáltak alapértelmezésben, más esetekben azonban a változók inicializálás nélküli tartalma valami memória-szennyeződés (ismeretlen érték) lehet.

A fenti példák *definíciók* voltak, azaz változóknál rendelkeztünk a tárterület lefoglalásáról ill. megadtunk a függvények törzsét. A következő példák *deklarációk* lesznek:

```
1    extern int i;        /* egész deklarációja,
2                            valahol máshol van definiálva */
3    extern int *pi;      /* mutató deklarációja,
4                            valahol máshol van definiálva */
5    extern int t[10];    /* egész tömb deklarációja,
6                            a méretet nem vesszük figyelembe */
7    extern int t[];      /* egész tömb deklarációja,
8                            ekvivalens a fentivel */
9    extern int tt[10][20];  /* tömb deklarációja, minden
10                              tömbelem 20 int-ből álló tömb */
11   extern int tt[][20];    /* tömb deklarációja,
```

```
12                          ekvivalens a fentivel */
13  extern int func1(void); /* paraméter nélküli függvény
14                             deklarációja */
15  extern int func2(int i, double d); /* int, double
16                          paraméteres fv deklarációja */
17  int func1(void);    /* függvénydeklarációkor az extern
18                         kulcsszó elhagyható */
19  int func2();        /* csak deklarációkor: semmit sem
20                         tudunk a paraméterekről */
```

A forrásfájlunkat sikeresen lefordíthatjuk, ha a használt változókat, függvényeket deklaráltuk. A futásra kész, összeszerkesztett programunknak azonban rendelkeznie kell a deklarációkhoz tartozó pontosan egy definícióval. A változót, amit több forrásfájlban is használunk, pontosan egy fordítási egységben le kell foglalni. A függvényt, ami több forrásfájlból is hívható, pontosan egy fordítási egységben definiálnunk kell: meg kell adni, milyen utasításokat tartalmaz.

Annak ellenőrzését, hogy egy *másik* forrásfájlban mit csináltunk, nem tudja ellenőrizni a fordító, ami csak a pillanatnyilag fordított fájlt látja. Ezért az evvel kapcsolatos hibákat nem a fordító, hanem a *szerkesztő* (linker) program fogja detektálni. Amennyiben nincsen egyetlen definíció sem, akkor a linker *feloldatlan hivatkozás* (unresolved external) hibát fog jelezni, ha pedig egynél több azonos nevű objektumot definiálunk, akkor *többértelmű hivatkozás* (ambigous reference) hibát kapunk.

Láthatóság

A láthatóság (scope) szabályai határozzák meg, hogy egy *azonosítót* (pl. változó-, függvény-, vagy típusnevet) a program mely részein használhatunk az adott objektum azonosítására. A egy változó láthatóságát szokás a változónév *hatókörének* is nevezni.

A C-ben egy deklaráció lehet **lokális**, ha valamely függvényen belüli blokkban helyezkedik el, vagy **globális**, ha minden függvényen kívül.

A lokális nevek a deklaráló blokkon belül láthatóak, beleértve a belső blokkokat is, kivéve, ha ugyanezt a nevet egy belső blokkban újra deklarálják, azaz *eltakarják* (hide). A globális változók a deklaráció helyétől a forrásfájl végéig látszódnak, hacsak egy blokkban el nem takarják őket. Függvényeket csak globálisként tudunk definiálni, azaz nem léteznek függvénybe beágyazott lokális függvények (mint léteznek pl. Pascal-ban).

A lokális változók **belső szerkesztésű**-ek (internal linkage), azaz a linker számára láthatatlanok. A globálisan deklarált nevek alapértelmezésben **külső szerkesztésűek** (external linkage), a linker számára láthatóak. Globálisoknál a **static** kulcsszó jelenti azt, hogy a név *belső szerkesztésű*. Az ilyen neveket a linker nem látja, azaz ezek a nevek csak az adott forrásfájlban használhatók. Ha ugyanazt a nevet több forrásfájlban belső szerkeszthetőségűnek definiálunk, akkor arra a linker nem jelez hibát.

A C nyelvben a belső szerkesztésű változókat és függvényeket gyakran egy nagyobb kódmodul belső, *implementációs* céljaira használjuk, a külső szerkesztésűeket pedig az illető modul *interfészének*. Íly módon, bár elég primitíven, szimulálni tudjuk az objektum-orientált nyelvek *enkapszuációs* elveit. A **main** mindig külső szerkesztésű kell legyen.

```
1   int i;          /* globális, külső szerkesztésű */
2   static int j;   /* globális, belső szerkesztésű */
3   extern int n;   /* globális, valahol máshol definiált */
4   extern double fahr2cels(double); /* függvény deklaráció */
5
6   void f() /* külső szerkesztésű, hívható más forrásfájlból */
```

```
7   {
8     int i;              /* lokális i, eltakarja (1)-et */
9     static int k = 5;   /* lokális k, statikus élettartam */
10    {
11      int m = n; /* lokális m, globális n (3)-nál deklarálva*/
12      int i = k; /* lokális i, eltakarja (8)-at */
13    }
14    i = 6;     /* ez ismét (8)-ban deklarált */
15  }
16  static void g()  /* statikus függvény: belső szerkesztésű */
17  {
18    extern double aa;     /* deklaráció, máshol definiált aa */
19    extern void h(int);   /* deklaráció, máshol definiált f  */
20    aa = fahr2cels(35); /*(4)-ben deklarált függvény hívása */
21    h(aa);              /* (19)-ben deklarált függvény hívása */
22    ++i; ++j;           /* globális i (1) és j (2) használata */
23  }
```

Az ANSI C-ben (C89) a deklarációk minden blokkban meg kell előzzék a végrehajtható utasításokat. A C99 óta ez már nem szükséges, a C++-hoz hasonlóan tetszőleges helyen deklarálhatunk változókat. Ennek az az előnye, hogy csak akkor hozunk létre új változókat, amikor kezdőértéket tudunk nekik adni, így kevesebb definiálatlan értékű változónk lesz.

Élettartam

Az élettartam szabályok azt határozzák meg, hogy az egyes memóriaterületek melyeket a programunk használ, mettől meddig érvényesek a programunk futása során. Ha olyan tárterületere hivatkozunk, ami már nem érvényes, súlyos futási idejű hiba következhet be.

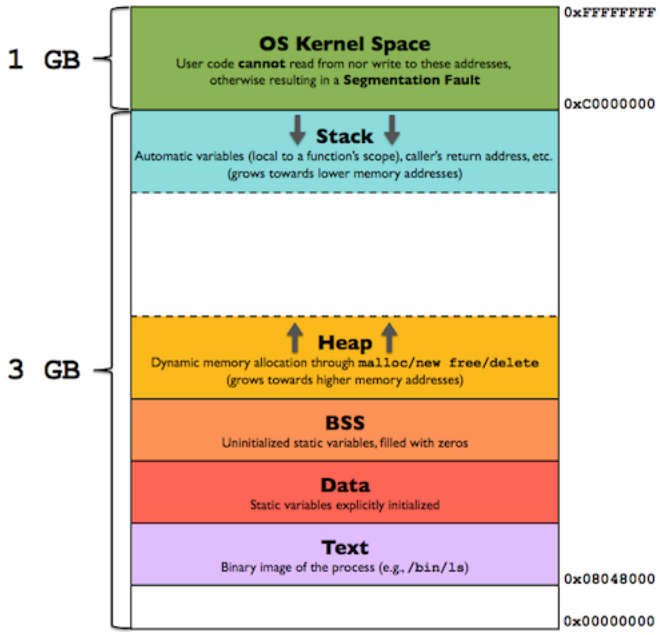
Tipikus élettartam kategóriák

Az első programozási nyelvek az összes változó számára a program elején lefoglalták a tárterületet, amit a program végéig fent is tartottak. Ez a **statikus élettartam** egy "biztonságos" megközelítés, de rendkívül pazarló, hiszen a változók jó részét csak a programunk kis területén (egy függvényen belül, vagy akár csak egy blokkon belül) akarjuk csak használni. Ezen a blokkon/függvényen kívül miért ne használhatnánk másra ugyanazt a memória-területet?

Az Algol 60 nyelvben vezették be a *blokk* fogalmát, ami nemcsak a vezérlés szerkezetét határozta meg, hanem a lokális változók élettartamát is. A lokális változók a blokkba való belépéskor foglalódtak le, és léteztek a belső blokkok, vagy meghívott függvények végrehajtása alatt is (bár esetleg nem voltak névvel elérhetőek, ha a neveiket eltakarták). Ezek a memóriaterületek akkor szabadultak fel, amikor a létrehozó blokkjuk végrehajtása befejeződött. Ez az **automatikus élettartam** képes ugyanazt a memóriaterületet időben máskor más és más változók számára kiosztani.

Végül olyan eset is előfordul, amikor a tárterület létrehozása és megszünése nem kapcsolható egy blokk végrehajtásához. Pl. az egyik függvényben foglaljuk le a tárterületet és egy másikban kell megszüntetni azt. Ez a **dinamikus élettartam**, amikor a programozó vezérli (függvényhívásokkal vagy más módon) a tárterület élettartamát.

Egy tipikus (UNIX) folyamat memória-szerkezete pl. így nézhet ki:



Statikus élettartam

A globális változók, ideértve a belső szerkesztésű, static globálisakat is *statikus élettartamúak*. Tárterületük a program elején létrejön és a program végéig lefoglalva marad. A statikus tárterületek inicializálási sorrendje a forrásfájlon belül a definíciós sorrend, a fordítási egységek közötti sorrendiség nem definiált. A nem inicilaizált statikus memóriaterületek kezdőértéke nulla.

```
1   char buffer[80];/* statikus élettartam, kezdőértéke csupa
2   0*/
3   int k = 42;          /* statikus élettartam, kezdőértéke 42
4   */
5   static double j;    /* statikus élettartam, kezdőértéke 0.0
6                          nem látható másik fordítási egységből
7   */
8
9   int main()
10  {
      /* ... */
    }
    /* az élettartamok vége */
```

A statikus élettartam egy speciális esete, amikor egy lokális változót definiálunk **static** kulcsszóval. Ilyenkor a static nem a szerkesztést, hanem az élettartamot befolyásolja, az ilyen lokális változók statikus élettartamúak. A statikus lokális változók egyetlen egyszer inicalizálódnak.

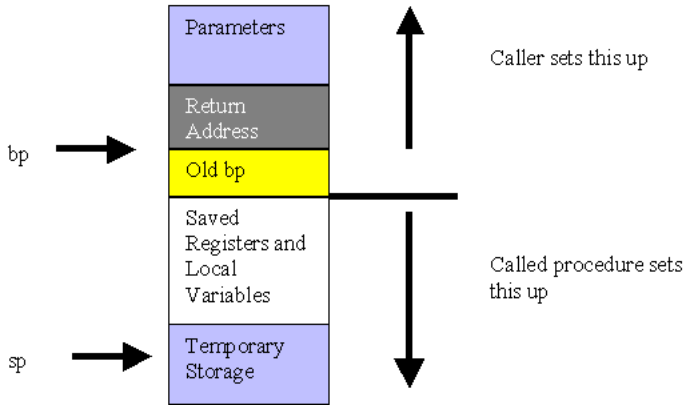
```
1  int count(void)
2  {
3    static int cnt = 0; /* lokális statikus inicializálása
4                           csak első alkalommal hajtódik végre */
5    ++cnt;
6    /* ... */
7    return cnt; /* minden híváskor egyel nagyobbat ad vissza */
8  }
```

A lokális statikusok a sima (automatikus) lokális változókhoz képest a blokkból kilépve is megőrzik a tartalmukat, és a legközelebbi belépéskor "emlékeznek" rá. Olyan globális változóknak gondolhatjuk őket, melyek láthatósága a blokkra korlátozott.

Automatikus élettartam

A (nem statikus) lokális változók a C programozási nyelvben a program **végrehajtási vermében** (program stack) jönnek létre. A verem egyben a függvényhívásoknál a *paraméterátadás* és a *visszatérő értékek* közvetítésére is szolgál.

Az ilyen változók a blokkba való belépéskor foglalódnak le és élettartamuk megszűnik, amikor elhagyjuk a blokkot. Ha van inicializálásuk, akkor az minden egyes alkalommal megtörténik, amikor belépünk a blokkba. Ellenkező esetben a változók értéke nem definiált (valami memóriaszemét, ami a verem korábbi használatából maradt ott). Amikor a blokk végrehajtása befejeződik, a verem állapota visszaáll a blokkba való belépés előttire, azaz az automatikus változóink (és függvényparamétereink) tárterülete felszámolódik, más függvények, blokkok számára felhasználhatóvá válik.



A **bp** az ún. *bázis-pointer*, ami egy adott függvényhívás során a verem által használt területet, az ún. *stack-frame*-et azonosítja. A lokális változók (és az átadott függvényparaméterek) pozíciója a bátzispointerhez képest relatív távolsággal kerül meghatározásra. A stack frame-et szokás *aktivizációs rekordnak* is nevezni.

Ha a blokk végrehajtása során egy függvényt hívunk, akkor annak a függvénynek a számára újabb stack-frame foglalódik le, ezalatt a változóink értéke megőrződik. Ez egyben azt is jelenti, hogy a függvények rekurzívan is hívhatóak: minden hívás saját stack-frame-et hoz létre.

Egy slide-show a program stack és az automatikus változók működéséről.

Dinamikus élettartam

Vannak esetek, amikor a memóriaterület létrehozása és felszámolása nem kapcsolható valamely függvény vagy blokk végrehajtásához. Ilyenkor a programozó manuálisan intézkedik a tárterület lefoglalásáról az ún. **szabad memóriából** (free memory, heap). A tárterület lefoglalva marad, amíg a programozó azt manuálisan fel nem szabadítja. Amennyiben ezt elmulasztja, akkor hosszan futó programok esetében (pl. egy szerver program vagy maga az operációs rendszer) a rendszeres fel nem szabadított allokálások miatt a memória elfogyhat. Ezt a hibajelenséget nevezzük *memória elszivárgásnak* (memory leak).

Számos modern programozási nyelv figyeli, hogy létezik-e még hivatkozás a heap területen lefoglalt memóriaterületekre. Ha az már "elérhetetlen", akkor "begyűjtésre" jelöli meg, és ha szabad memóriára lenne szükség, akkor felszabadítja és újrahasználja azt. Ezt a bonyolult és nem olcsó mehanizmust nevezzük **szemétgyűjtésnek** (garbage collection), illetve az ezt elvégző eszközt szemétgyűjtőnek (garbage collector).

Azok a nyelvek, melyek valamely virtuális futtató környezetet használnak, mint a Smalltalk, Java, C# és Eiffel, alkalmazzák a szemétgyűjtést, más nyelvek, ahol a

hardver közvetlen, hatékony elérésén van a hangsúly, mint a C vagy a C++, azok nem. Ez utóbbi nyelveknél nagyon kell figyelnünk a memória elszivárgás megelőzésére.

C nyelvben a dinamikus memória lefoglalását a **malloc** függvénnyel végezzük, melynek paramétereként a lefoglalandó bájtok számát adjuk meg. Jó ötlet itt a **sizeof** operátor használata. A *malloc* egy **void*** pointert ad vissza, amit a szükséges típusra kell konvertálnunk. Előfordulhat, hogy nincsen elég memória, ilyenkor a malloc **NULL** pointert ad vissza, ezt *soha se felejtsük el ellenőrizni*!

A memória felszabadítását a **free** függvény végzi, aminek a malloc által adott mutatót kell megadnunk. A felszabadított tárterületet tilos tovább használnunk, ez futási idejű hibát okoz. Különösen súlyos hiba a többszöri felszabadítás. A *free* függvény kaphat NULL pointert, ekkor semmit sem csinál.

```
1   #include <stdlib.h>
2   #include <assert.h>
3
4   void g(double *);
5
6   void main()
7   {
8     double *dbls;
9
10    /* megkísérlünk 1024 char-t lefoglalni */
11    char *buffer = (char*) malloc(1024);
12
13    if ( NULL != buffer )  /* sikeres volt a foglalás? */
14    {
15        *buffer = 'x';   /* használhatjuk a területet  */
16        free (buffer );  /* felszabadítjuk a területet */
17    }
18
19    /* megkísérlünk 10 darab double-t lefoglalni */
20    if ( dbls = (double*) malloc( 10*sizeof(double) ) )
21    {
22        /* sikeres volt a lefoglalás */
23        g(dbls);   /* dbls használata */
24        /* a területet itt már nem használhatjuk
25           mert g() felszámolta */
26    }
27  }
28
29  void g( double *dptr)
30  {
31    assert ( NULL != dptr );
32    dptr[2] = 3.14;  /* használjuk a területet */
33
34    /* átméretezzük, a régi terület átmásolódik */
35    if ( dptr = realloc( dptr, 20) )
36    {
37        dbtr[19] = dbtr[2]; /* a régi értékek átmásolódtak */
38    }
39    free(dptr);   /* a 20 double felszabadítása */
40  }
```

Élettartammal kapcsolatos hibák

Az alábbiakban ezgy esettanulmányon keresztül megvizsgáljuk a láthatóság és élettartam kapcsolatát és azt, milyen hibákat kell elkerülnünk.

Az esettanulmány slide-show formátumban.

Legyen feladatunk egy egyszerű *answer* függvény megírása, amelyik kiírja a paraméterként kapott kérdést, beolvassa a választ és azt visszaadja a hívójának. A hívó program (*main*) kiírja a választ a standard outputra.

```
1   /*
2    * Ez nagyon HIBÁS verzió
3    */
4   #include <stdio.h>
5   char *answer( const char *question);
6   int main()
7   {
8     printf( "answer = %s\n", answer( "How are you? ") );
9     return 0;
10  }
11  /* nagyon hibás !! */
12  char *answer( const char *question)
13  {
14    char buffer[80]; /* lokális láthatóság, aut. élettartam */
15    printf( "%s\n", question);
16    gets(buffer);  /* ERR1: buffer-túlcímzés !! */
17    return buffer; /* ERR2: automatikus élettartam vége,
18                      tilos használni! */
19  }
```

Két súlyos hibát követtünk el:

1. A **gets(buffer)** az első újsor karakterig olvassa a karaktereket, így lehet, hogy többet olvasnánk, mint a bufferünk hossza. Ez súlyos hiba, mert felülírjuk a buffer mögötti memóriát. Ez a *buffer-túlcsordulás* hiba ez egyik legkritikusabb C biztonsági hibák egyike.
2. A **return buffer** egy karakterre mutató pointert ad vissza a lefoglalt automatikus élettartamú tömb elejére. Viszont amint visszatérünk a függvényből, a verem ezen része felszabadul és a mutatónk egy invalid területre fog mutatni.

Javítási kísérlet, változtassuk meg a buffer élettartamát és cseréljük ki a beolvasást biztonságosabbra:

```
1   /*
2    * Működik, de nehezen karbantartható
3    */
4   #include <stdio.h>
5
6   #define BUFSIZE 80
7   char buffer[BUFSIZE]; /* globális láthatóság,
8                            statikus élettartam */
9   char *answer( const char *question);
10  int main()
11  {
12    printf( "answer = %s\n", answer( "How are you? ") );
13    return 0;
```

```
}
char *answer( const char *question)
{
  printf( "%s\n", question);
  fgets(buffer, BUFSIZE, stdin); /* legfeljebb BUFSIZE-1
                                    char olvasása */
  return buffer;   /* ok, pointer globálisra */
}
```

Ez így működik, de nehezen karbantartható. A *buffer* feleslegesen globális, neve ütközhet más fordítási egységekkel. Túl sok helyről elérhető. Rejtsük el a függvényen kívüli világ elől.

```
/*
 * Működik, karbantarthatóbb, de nem szál-biztos
 */
#include <stdio.h>
#define BUFSIZE 80

char *answer( const char *question);
int main()
{
  printf( "answer = %s\n", answer( "How are you? ") );
  return 0;
}
char *answer( const char *question)
{
  static char buffer[BUFSIZE]; /* lokális láthatóság,
                                  statikus élettartam */
  printf( "%s\n", question);
  fgets(buffer, BUFSIZE, stdin); /* legfeljebb BUFSIZE-1
                                    char olvasása */
  return buffer;   /* ok, pointer statikus élettartamúra */
}
```

Ebben a környezetben az *answer* függvény már jól működik, a buffer élettartama statikus, tehát a függvény visszatérése után is használható, láthatósága viszont lokális, így lényegében implementációs részletként eltakartuk a külvilág elől.

A statikus élettartamú változóknak is van azonban veszélye. Mivel *egyetlen* példányban léteznek, nem pedig minden egyes függvényhíváskor a veremben jönnek létre, mint az automatikusak, veszélyes, ha egy időben több helyről használjuk őket. Az alábbi program mindig ugyanazt a (második) választ adja vissza:

```
/*
 * Működik, karbantarthatóbb, de nem szál-biztos
 */
#include <stdio.h>
#define BUFSIZE 80

char *answer( const char *question);
int main()
{
  printf( "answer = %s\n%s\n", answer( "How are you?"),
                               answer("Sure?") );
  return 0;
```

```
13  }
14  char *answer( const char *question)
15  {
16    static char buffer[BUFSIZE]; /* lokális láthatóság,
17                                    statikus élettartam */
18    printf( "%s\n", question);
19    fgets(buffer, BUFSIZE, stdin);
20    return buffer; /* ugyanaz a buffer minden hivás esetén */
21  }
```

Minden egyes hívás esetén ugyanoda rakjuk a választ (evvel potenciálisan felülírva a korábbi válaszokat). Ha több hívásunk is van, csak az utolsót fogjuk tudni kiolvasni. Ez a probléma különösen veszélyesen jelentkezik *többszálú* (multithreaded) programok esetében.

Úgy tűnik, minden *függvényhívás* esetében új területre van szükségünk. Próbáljuk meg dinamikus élettartammal.

```
1   /*
2    * Egy ideig működik, de memória elszivárgást okoz
3    */
4   #include <stdio.h>
5   #include <stdlib.h>
6   #define BUFSIZE 80
7
8   char *answer( const char *question);
9   int main()
10  {
11    printf( "answer = %s\n%s\n", answer( "How are you?"),
12                                 answer("Sure?") );
13    return 0;
14  }
15  char *answer( const char *question)
16  {
17    /* új memória lefoglalása minden híváskor */
18    char *buffer = (char *) malloc(BUFSIZE);
19    if ( NULL == buffer )
20        return "error\n"; /* jelezzük a hibát */
21    printf( "%s\n", question);
22    fgets(buffer, BUFSIZE, stdin);
23    return buffer; /* mindig új buffer */
24  } /* de ki fog felszabadítani? */
```

Ez a megoldás egy ideig működik, de közben *memória elszivárgást* okoz. Minden alkalommal újra és újra lefoglaljuk a memóriát, de sohasem szabadítjuk fel. Nem is lenne egyszerű, hol szabadítsuk fel: az *answer* függvényben még korai, a *main*-ben meg nem igazán alkalmas.

A helyes megoldáshoz azt kell eldönteni, hogy végül is, *kinek van szüksége a tárterületre*? Ki legyen a *tulajdonos* (owner), akinek a feladata a memória kezelése?

Mivel a tárterületet a *main* akarja felhasználni, legyen ő a tulajdonos!

```
1   /*
2    * OK
3    */
4   #include <stdio.h>
```

```
5   #define BUFSIZE 80
6
7   char *answer( const char *question, char *buffer, int len);
8
9   int main()
10  {
11    char buffer1[BUFSIZE], buffer2[BUFSIZE]; /* lokális,
12                                                automatikus */
13    printf("answer = %s\n%s\n",
14                     answer("How are you?", buffer1, BUFSIZE),
15                     answer("Sure?", buffer2, BUFSIZE) );
16    return 0;
17  }
18  char *answer( const char *question, char *buffer, int len)
19  {
20    printf( "%s\n", question);
21    fgets(buffer,len,stdin); /*kölcsönkapott területre írunk*/
22    return buffer; /* ok, a hívó függvényben van lefoglalva */
23  }
```

A körülményekhez képest még ez a legstabilabb, karbantarthatóbb megoldás.

Homepage of Dr. Zoltán Porkoláb

Home

Archive

Teaching

Timetable

Bolyai College

C++ (for mathematicians)

Imperative programming (BSc)

Multiparadigm programming (MSc)

Programming (MSc Aut. Sys.)

Programming languages (PhD)

Software technology lab

Theses proposals (BSc and MSc)

Research

CodeChecker

CodeCompass

Templight

Projects

Conferences

Publications

PhD students

Affiliations

Dept. of Programming Languages and Compilers

Ericsson Hungary Ltd

Imperatív programozás 7.

Függvények, paraméterátadás

A *függvények* és *eljárások*, gyakori összefoglaló nevükön **alprogramok** az imperatív nyelvek alapvető építőkövei. Levetővé teszik, hogy a nagyobb, komplex programokat kisebb, könnyebben karbantartható, egyszerűbb részekre bontsuk fel, eltakarva az implementációs részleteket a külvilág elől. Segítségükkel újra fel tudjuk használni a már megírt algoritmusainkat, az esetleges különbségeket a paraméterekkel kifejezve.

Függvények egy csoportját külön fordítási egységekbe szervezhetjük és könyvtárakat is építhetünk belőlük. Ezeket a könyvtárakat statikusan vagy dinamikusan szerkeszthetjük hozzá a programjainkhoz (lásd 1. előadás). A leggyakrabban használt függvnyek a **standard könyvtárban** vannak és onnan használhatóak a megfelelő fejlécállományok (header-ek) include-ja után.

A C nyelvet úgy tervezték, hogy könnyen írhassunk függvényeket és azok futási időben kis költséggel végrehajthatók legyenek.

A függvénydeklarációban nem kell megadnunk a függvény törzsét, csak a hívásához szükséges információkat: a nevet, a paramétereket és a visszatérési érék típusát (ha van). A függvények deklarációjában használt paraméter neveket szokás **formális paraméternek** (parameter) nevezni, míg a függvény meghívásakor ténylegesen átadott értékek az **aktuális paraméter** (argument). A függvény nevét a teljes paraméterlistával a függvény **szignatúrájának** (signature) nevezzük, a visszatérő érték típusát és a szignatúrát együtt pedig a függvény **prototípusának** (function prototype) nevezzük.

C-ben az összes alprogramot egységesen függvénynek nevezzük. Amennyiben nem adnak vissza értéket (eljárás), akkor azt a **void** visszatérő típussal jelöljük.

Függvénydeklaráció C-ben

A visszatérő érték és a paraméterlista típusa része a függvény típusának. Ahol a függvénynek nincsen visszatérő értéke a **void** típust használjuk. Tömbök nem használhatóak visszatérő típusként.

```
int f(void);     /* függvény par. nélkül, int visszatérő típussal  */
int *fp(void);   /* függvény par. nélkül, int* visszatérő típussal */
```

A deklarációkban használt formális paraméternevek csak leíró szerepűek, ténylegesen nem használja fel a fordító, és el is hagyhatjuk őket.

```
1   int f(int *x, int *y); /* az x,y neveknek nincs szerepe,  */
2                          /* elhagyhatók                     */
3   int f(int *, int *);   /* ekvivalens a fentivel */
```

A C nagyon régi, ANSI szabvány előtti verziójában nem használtunk prototípusos deklarációkat. Az evvel való kompatibilitás miatt az ANSI C-ben is elhagyhatjuk a paraméterek specifikációját:

```
1   int f(void);     /* függvény pontosan nulla paraméterrel   */
2   int g();         /* függvény paraméter-specifikáció nélkül */
```

Itt f() és g() különböznek. Tudjuk, hogy f() pontosan nulla paraméterrel rendelkezik, de semmit sem tudunk g() paramétereinek számáról vagy típusáról. A g() deklarációja reverz kompatibilis az ANSI előtti C-vel, de ilyet új kód írásakor ne használjunk.

Megjegyzés C++-ban *mindig* alkalmaznunk kell a prototípusos deklarációt. Ott a g() jelölés ekvivalens a g(void) jelöléssel és a pontosan nulla paramétert jelenti.

Abban az esetben, ha nem tudjuk a paraméterek számát, vagy típusát, mint pl a *printf* esetében, használjuk az *ellipsis* jelölést:

```
1    int printf(const char *format, ...);
2    int fprintf(FILE *stream, const char *format, ...);
```

Ennek a jelölésnek az a jelentése, hogy *nulla vagy több további paraméter ismeretlen típussal*. Ilyen függvényeket nem egyszerű implementálni, az ilyen változó paraméterlistájú (variadic parameter) függvényeket a <*stdarg.h*> headerfájl **va_** makróival írhatunk.

Függvényhívás C-ben

Amikor egy *f* függvényt meghívunk, a hívásnak meg kell felelnie a deklaráció prototípusának:

- A paraméterek száma ugyanannyi.
- A hívás minden aktuális paramétere olyan típusú, hogy értékül adható legyen a deklaráció formális paraméter típusának.

Ha a paraméterek száma nem felel meg a deklarációnak, akkor a viseledés nemdefiniált. Ha a deklaráció az *ellipsis* jelölést használja, vagy a típusok nem kompatibilisak, a viselkedés szintén nemdefiniált.

Ha a deklaráció *nem prototípusos* (azaz üres nyitó-csukó zárójeles, pl. void f()), akkor a függvényhíváskor az ún. *default promóciók* történnek meg, mint az egész típusok **int** vagy **long**-ra és a lebegőpontos típusok **double** vagy **long double** konverziója.

Ha a deklaráció *prototípusos*, azaz felsoroltuk az egyes paramétereket és típusait, akkor az aktuális paraméter értékek pontosan úgy konvertálódnak a formális paraméterekre, mint ha értékadás történne. Az *ellipsis*-től kezdve ez a konverzió megáll, és onnan csak a default promóciók történnek meg.

```
1   double fahr2cels(double);  /* prototípusos deklaráció */
2   double cels2fahr();        /* nem prototípusos deklaráció */
3   /* ... */
4   float f = 3.14;
5   printf("%f\n",fahr2cels(36)); /*ok, double-re konvertálódik*/
6   printf("%f\n",cels2fahr(f));  /*ok, float->double promóció */
7   printf("%f\n",cels2fahr(36)); /*hiba, int param. adódik át */
```

Két további konverzió történhet még meg:

- *signed – unsigned* konverzió, ha az érték reprezentálható mindkét típusban
- *void* * – *char* * konverzió.

Egy függvényhívás **szekvecia-pont**, azaz először az aktuális paraméterek értékelődnek ki nemdefiniált sorrendben, és a függvény törzsének végrehajtása csak azután kezdőthet. A paraméterek kiértékelésének *egymás közötti* sorrendje viszont definiálatlan. (A paramétereket elválasztó vessző *nem* a vessző operátor.)

```
( *t[f1()] ) ( f2(), f3(), f4() );
```

Az *f1*, *f2*, *f3*, *f4* függvények akármilyen sorrendben meghívódhatnak.

A rekurzív függvényhívások akár direkt akár indirekt módon megtörténhetnek:

```
1       int factorial(int n)
2       {
3         if ( 1 == n )
4           return 1;
5         else
6           return n * factorial(n-1);
7       }
```

Ha ezt a függvényt olyan paraméterrel hívjuk meg, ami 1-nél kisebb, akkor *végtelen rekurzió* következik be, ami futási idejű hibát okoz.

A rekurzív függvények gyakran olvashatóbbak, de rosszabb futási időben hatékonyságúak, mint pl. a ciklussal megírt verzió. Ha a rekurzív függvény utolsó utasítása a rekurzív hívás, akkor **végrekurzió**-ról beszélünk (tail recursion). Az ilyen rekurzív függvényeket a fordító programok gyakran automatikusan átalakítják ciklusra.

```
1       int factorial(int n)
2       {
3         int result = 1;
4         int i;
5         for ( i = 2; i <= n; ++i )
6           result *= i;
7         return result;
8       }
```

Paraméterátadás

A programozási nyelvek az idők során számos módszert dolgoztak ki a paraméterek átadására.

Cím szerinti

A *cím szerinti* (call by address, call by reference) paraméterátadás esetében az aktuális és formális paraméterek memória-lokációja megegyezik, az alprogram paramétere lényegében a híváskor átadott tárterület (változó) szinonímája (álneve). Minden módosítás, amit az alprogramban a paraméteren végzünk valójában az átadott aktuális paraméteren történik meg és azonnal látszik. Ez a legegyszerűbben implementálható és az egyik legrégibb paraméterátadási módszer.

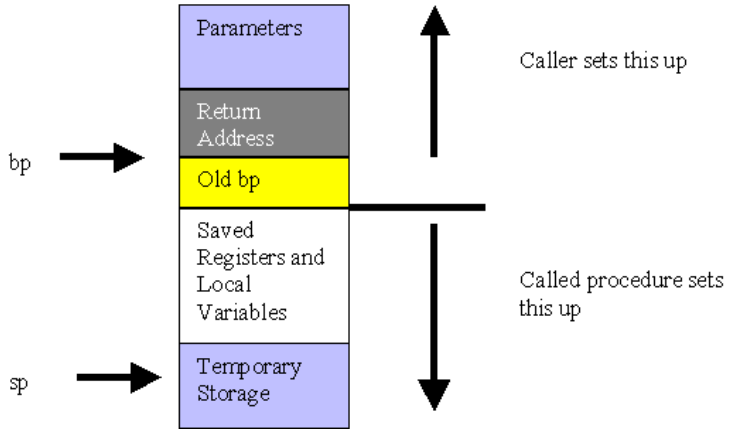
Előnye, hogy az alprogram a paramétereken keresztük kétirányú kommunikációt folytathat a hívóval: onnan információkat kaphat és visszafelé is adhat. A paraméterek módosítása valójában a hívó változóinak módosítása. Ez néha zavaró is lehet, különösen ha mind az aktuális és a formális paraméter is látható/haszálható az alprogramban. Bár a cím szerinti paraméterátadás olcsón implementálható, hiszen nincsenek másolások, mint az érték szerintiben, az *aliasing* lehetősége miatt a fordítóprogram csak óvatosabban optimalizálhat.

Ugyancsak problémát jelenthet, ha kifejezéseket vagy konstansokat (pl. 42) adunk át paraméterként, hiszen ezeknek eredetileg nincsen memóriaterületük.

Érték szerint

A C nyelv paraméterátadása így működik.

Az *érték szerinti* (call by value) paraméterátadás során a függvény formális paramétereit úgy tekintjük, mintha azok a függvény lokális változói lennének. A függvényhívás során ezeket a "lokális változókat" az aktuális paraméter értékekkel inicializáljuk: lényegében azok értékeit másoljuk a formális paraméterekbe. Mindez azt jelenti, hogy az alprogram végrehajtása során az aktuális és formális paraméter jól elkülönül. Ennek nemcsak az az előnye, hogy könyebben érvelhetünk a program működéséről, de gyakran a fordítóprogram is hatékonyabb kódot fordíthat, ha nem lehetséges *alisaing*.



A stack működése C-ben.

A módszer előnye, hogy a paramétereket úgy kezelhetjük, mint a lokális változókat. Hátránya, hogy az alprogramban a paraméterek módosítását nem tudják továbbítani a hívó eljárás felé. Ezt gyakran úgy kerüljük meg, hogy paraméterként eleve a módosítandó területre hivatkozó mutatót adjuk át. Pont így működik pl. a C **scanf** függvénye.

Eredmény szerinti

Az *eredmény szerinti* (call by result) paraméterátadás nagyon hasonló az érték szerintihez, de az alprogramból történő visszatérés pillanatában az alprogramban létező másolat (a formális paraméter) értéke visszamásolódik az aktuális paraméterbe. Így a függvény végrehajtása során külön-külön tárterületet használ a formális és az aktuális paraméter, de az alprogram általi módosítások a visszatérés pillanatában láthatóvá válnak a hívó számára. Más mellett ilyen paraméterátadási módot is használ az ADA *output* és *inout* paraméterek esetében.

Név szerinti

A *név szerinti* (call by name) paraméterátadást elsősorban az Algol 60 és a Simula 67 alkalmazta. Ebben az esetben nem az aktuális paraméter memória területét vagy pillanatnyi értékét használjuk fel a paraméterátadáskor, hanem magát a kifejezést, amit a programozó beírt a függvényhíváskor. Amikor az alprogram végrehajtása során hivatkozunk a formális paraméterre, akkor újra kiértékeljük az átadott kifejezés pillanatnyi értékét, és azt használjuk.

Implemetációja gyakran úgy történt, hogy a kifejezést kiszámoló kis eljárást, ún. *closure*-t adtunk át, és ezt hajtódott végre a paraméter minden meghivatkozásakor. A mai nyelvekben ritkán alkalmazzuk, ha mégis valami hasonlóra van szükségünk, akkor pl. C++-ban *lamdba kifejezést* adunk át paraméterként.

Az egyes programozási nyelvek stratégiái

A FORTRAN és számos azt követő nyelv cím szerinti paraméterátadást alkalmazott. Amikor nem változót, hanem egy kifejezést adtunk át, akkor azt egy temporális tárterületen számoltuk ki és ennek a területnek címét adtuk át. A C programozási nyelv *érték szerinti* paraméretátadást használ. A C++ alapértelmezésben ugyancsak érték szerinti, de a *referencia* típusú paraméterek esetében lényegében a cím szerinti paraméterátadással dolgozik. Ugyanígy a Pascal is használja mindkét módszert: az alapértelmezés az érték szerinti-nek felel meg, viszont a **var** kulcsszó alkalmazásával lényegében cím szerint adhatunk át paramétereket. Az Algol 60 és Simula 67 érték és név szerinti paraméterátadást használt. Javaban a beépített típusok érték szerint adódnak át, de a *class* típusok *referencia* szerint, ilyenkor lényegében olyan pointerek adódnak át, amik a tényleges objektumra mutatnak - hatásában ez leginkább a cím szerinti paraméterátadásnak felel meg. Az ADA nyelv érték és eredmény szerinti paraméterátadást használ.

Paraméterátadás C-ben

A C programozási nyelvben az aktuális paraméterek **érték szerint** adódnak át, azaz a kifejezés értéke **bemásolódik** a formális paraméter területére, pont úgy, mintha a függvényben definiált lokális változó lenne, amit az aktuális paraméterből inicializálnánk.

```
1       #include <stdio.h>
2       void increment(int i)
3       {
4         ++i;
5         printf("i in increment() = %d\n",i);
6       }
7       int main()
8       {
9         int i = 0;
10        increment(i);
11        increment(i);
12        printf("i in main() = %d\n",i);
13        return 0;
14      }
```

```
$ gcc -ansi -pedantic -Wall -W f.c
$ ./a.out
i in increment() = 1
i in increment() = 1
i in main() = 0
```

A *cím szerinti* paraméterátadást szimulálhatjuk avval, ha a változó helyett annak címét adjuk át paraméternek:

```
1       #include <stdio.h>
2       void increment(int *ip)
3       {
4         ++*ip;
5         printf("i in increment() = %d\n",*ip);
6       }
7       int main()
8       {
9         int i = 0;
```

```
10        increment(&i);
11        increment(&i);
12        printf("i in main() = %d\n",i);
13        return 0;
14      }
```

```
$ gcc -ansi -pedantic -Wall -W f.c
$ ./a.out
i in increment() = 1
i in increment() = 2
i in main() = 2
```

Pont így működik a **scanf** függvénycsalád, ami az inputról olvas:

```
1     #include <stdio.h>
2     void f(void)
3     {
4       int i;
5       double d;
6       char c;
7       char buffer[20];
8
9       if ( 4 == scanf("%d %f%19s %c", &i, &d, buffer, &c) )
10        /* 42   3.14e-2   Hello  word
11           i == 42, f = 0.0314, buffer = "Hello", c == ' '
12        */
13    }
```

A tömb paramétereket az első elemre mutató pointer értékként adjuk át:

```
1     #include <stdio.h>
2     int f( int *t, int i)
3     {
4       return t[i];
5     }
6     int main()
7     {
8       int arr[] = {1, 2, 3, 4};
9       printf("%d\n", f(arr,1));/* arr tomb elso elemére mutat */
10      printf("%d\n", f(&arr[0],1)); /* ugyanaz, mint f(arr,1) */
11      return 0;
12    }
```

```
$ gcc -ansi -std=c99 -Wall -W a.c
$ ./a.out
2
2
```

Ezért ezek a deklarációk ekvivalensek:

```
1    int f( int *t, int i)    { return t[i]; }
```

```
int f( int t[], int i)  { return t[i]; }
int f( int t[4], int i) { return t[i]; }

/* a tömbhatárokat nem ellenőrzi a fordító, ezért ez is
 lefordul, de lehet, hogy hibásan működik */
int f( int t[4], int i)  { return t[6]; }
```

Mivel egy függvényparaméter sohasem lehet tömb (melyek helyett mindig egy pointer adódik át), ezért egy függvényparaméterre alkalmazni a **sizeof** operátort hibás eredményre vezet.

```
int f(void)
{
  char buffer[100]; /* sizeof(buffer) == 100 */
  return g(buffer);
}
int g( char t[])
{
  return sizeof(t); /* hiba! sizeof(pointer) */
}
```

A visszatérő típus

A visszatérő értékek a függvény visszatérő típusára konvertálódik:

```
double f(void)
{
  int i;
  /* ... */
  return i;   /* double-ra konvertálódik */
}
```

A main függvény paraméterei

A main() függvényt az alábbi módokon lehet definiálni:

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

/* ha az operációs rendszer támogatja (pl. UNIX) */
int main( int argc, char *argv[], char *envp[]) { /* ... */ }
```

Ha **argc** definiált, akkor **argv[argc]** nullpointer. Ha **argc** nagyobb nullánál, akkor **argv[0]** a program *neve*, ahogy azt meghívták, és **argv[1] ... argv[argc-1]** a program operációs rendszertől kapott paraméterei. Az **argv[i]** paraméterek NUL azaz '\0' karakterrel terminált karaktertömbök.

Normális esetben az **argc** mindig nagyobb, mint 0.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
```

```
4       printf("name of the program = %s\n", argv[0]);
5       for (int i = 1; i < argc; ++i)
6         printf("argv[%d] = %s\n", i, argv[i]);
7       return 0;
8     }
```

```
$ gcc -ansi -std=c99 -Wall -W -o mainpars mainpars.c
$ ./mainpars
name of the program = ./mainpars
$ ./mainpars first second third
name of the program = ./mainpars
argv[1] = first
argv[2] = second
argv[3] = third
```

Függvénymutatók

Függvényere is állíthatunk mutatókat, az ilyen pointerek típusához hozzátartozik a teljes prototípus, így a visszatérő érték típusa is.

A függvénymutató értékeket a (deklarált) függvénynevekből képezzük:

```
1     double sin(double); /* vagy #include <math.h> */
2     ...
3     double f(void)
4     {
5       double result = 0.0;
6       double (*fp)(double); /* fp mutató double(double) fv-re */
7       fp = sin;         /*˙fp most double sin(double)-re mutat */
8
9       if ( NULL != fp )
10      {
11        result = (*fp)(.5)        /* sin(.5) meghívása */
12        result =   fp (.5)        /* ekvivalens a fentivel */
13      }
14      return result;
15    }
```

Az előző increment-es példa pointerrel:

```
1     #include <stdio.h>
2     void increment(int *ip)
3     {
4       ++*ip;
5       printf("i in increment() = %d\n",*ip);
6     }
7     int main()
8     {
9       void (*fp)(int *); /* mutató void (int*) függvényre */
10      void (*gp)();      /* mutató ismeretlen paraméterlistájú
11                            függvényre */
12      int i = 0;
```

```
13      fp = increment;
14      gp = fp;
15      (*fp)(&i);     /* increment() meghívása */
16        fp (&i);     /* ugyanaz egyszerűbben  */
17        gp (&i);     /* increment() mghívása,
18                        de nem ellenőrzi a paramétereket */
19      printf("i in main() = %d\n",i);
20      return 0;
21    }
```

```
$ gcc -ansi -pedantic -Wall -W f.c
$ ./a.out
i in increment() = 1
i in increment() = 2
i in increment() = 3
i in main() = 3
```

Egy függvénymutató:

- mutathat a kompatibilis típusú függvényre
- értékül adható egy kompatibilis függvénymutatónak
- összehasonlítható a nullpointerrel
- meghívható a mutatott függvény, ha nem nullpointer

Néha hasznos a **typedef** használata, hogy egyszerűsítsük a definíciókat és deklarációkat. A typedef nem hoz létre új típust, csak a régi típus egy új szininímáját.

```
1   typedef double length_t; /* lenght_t a double szinonímája */
2   typedef int    index_t;  /* index_t az int szinonímája    */
3
4   /* trigfp_t a double visszatérőértékű, double paraméterű
5      függvénymutató szinonímája                             */
6   typedef double (*trigfp_t)(double);
```

A matematikai függvények használatához include-olni kell a **<math.h>** headert és a szerkesztéshez meg kell adni a matematikai könyvtár **-lm** kapcsolóját.

```
1    #include <stdio.h>
2    #include <math.h>
3
4    typedef double (*trigfp_t)(double);
5
6    trigfp_t inverse(trigfp_t fun)
7    {
8      static trigfp_t from[] = {  sin,  cos,  tan };
9      static trigfp_t to[]   = { asin, acos, atan };
10     int i = 0;
11
12     for ( i = 0; i < sizeof(from)/sizeof(from[0]); ++i)
13     {
14       if ( fun == from[i] ) return to[i];
15     }
16     return fun;
```

```
17  }
18
19  int main()
20  {
21      double   d1  = sin(.5);
22      trigfp_t rev = inverse(sin);
23      double   d2  = rev(d1);
24
25      printf( "sin(.5) = %f, arc sin(%f) = %f\n", d1, d1, d2);
26      return 0;
27  }
```

A fordítás és a futattás eredménye:

```
$ gcc -ansi -pedantic -Wall inverse.c -lm
$ ./a.out
sin(.5) = 0.479426, arc sin(0.479426) = 0.500000
```

Homepage of Dr. Zoltán Porkoláb

Home

Archive

Teaching

Timetable

Bolyai College

C++ (for mathematicians)

Imperative programming (BSc)

Multiparadigm programming (MSc)

Programming (MSc Aut. Sys.)

Programming languages (PhD)

Software technology lab

Theses proposals (BSc and MSc)

Research

CodeChecker

CodeCompass

Templight

Projects

Conferences

Publications

PhD students

Affiliations

Dept. of Programming Languages and Compilers

Ericsson Hungary Ltd

Imperatív programozás 8.

Összetett adatszerkezetek

A programozási nyelvekben gyakan van szükség összetett adatok kezelésére. Ezek közé tartozik a **tömb** (tömbök), ami azonos típusú elemek (egy- vagy többdimenziós) véges sorozatából áll, a **rekord**, ami különböző típusok rendezett N-ese, az **únió**, ami egy időben véges számú típus közül pontosan egyet tud tárolni. Bár nem aggregáció, de itt tárgyaljuk a **felsorolási típust** is. Egyes imperatív nyelvekben további összetett típusok is létezhetnek, mint pl. a **halmaz** (set) típus Pascal-ban, más nyelvekben ez a szabványos könyvtár része (pl. Java, C++).

Az összetett típusok maguk is tartalmazhatnak beépített vagy összetett típusokat, így alkothatunk rekordokból vagy úniókból tömböket, rekordok és úniók tartalmazhatnak tömböket, rekordokat, úniókat, felsorolási értékeket.

Felsorolási típus C-ben

A C felsorolási típus valójában egy egész jellegű értékekből álló halmaz, aminek értékeit szimbolikus nevekkel jelölhetjük. A felsorolási értékek a mögöttes egész típus értékei, és használhatóak bárhol, ahol egy **int** típusú érték használható.

A mögöttes egész típus a **char**, **int**, **unsigned int**, **long**, **unsigned long**, **...** valamelyike,ez implementációtól függ. Bármelyik is, egész számként viselkedik, pl. eritmetikai műveletek végezhetőek rajta.

```
1     enum color { WHITE, BLACK, RED, YELLOW, GREEN };
2
3     enum color read_light(void);
4
5     void f(void)
6     {
7       enum color traffic_light = read_light();
8
9       switch( traffic_light )
10      {
11      case RED:    puts("stop!");         break;
12      case YELLOW: puts("break!");        break;
13      case GREEN:  puts("go!");           break;
14      default:     puts("look around!"); break;
15      }
16    }
```

A felsorolási típus értékei a { } közötti felsorolásból kerülnek ki. Az első felsoroló értéke, ha explicit mást nem rendelünk hozzá a nulla lesz. Ha a felsorolóban egy névnél szerepel a **=** jel, akkor az a felsoroló azt az értéket veszi fel. Ha ilyen nem szerepel, akkor az előző felsorolónál eggyel nagyobb lesz. Az értékeknek **nem kell egyedieknek** lennie.

```
1     include <stdio.h>
2
```

```
3       enum color { A, B=2, C, D, E=1, F=A+B };
4
5       int main()
6       {
7         printf("%d\n", A);  /* 0 */
8         printf("%d\n", B);  /* 2 */
9         printf("%d\n", C);  /* 3 */
10        printf("%d\n", D);  /* 4 */
11        printf("%d\n", E);  /* 1 */
12        printf("%d\n", F);  /* 2 */
13
14        return 0;
15      }
```

```
$ gcc -ansi -pedantic -Wall enum.c
$ ./a.out
0
2
3
4
1
2
```

A felsolási értékek valamely egész típusra képződnek le, mint a **char** *signed* vagy *unsigned* egész. Ennek megfelelően a felsorolási értékek úgy viselkednek, mint az egészek, részt vesznek konverziókban és alkalmazhatóak rájuk az aritmetikai operátorok is.

A felsorolási típus változói teljes értékű C változók, így el lehet kérni a címüket, pointert lehet rájuk állítani, paraméterként átadhatók függvényeknek, stb.

```
1       void next(enum color *cptr)
2       {
3         switch( *cptr )
4         {
5         case RED:    *cptr = GREEN;  break;
6         case YELLOW: *cptr = RED;    break;
7         case GREEN:  *cptr = YELLOW; break;
8         default:     break;
9         }
10      }
11      void f(void)
12      {
13        enum color traffic_light = read_light();
14
15        next( &traffic_light );
16      }
```

Az **enum** kulcszó része a típusnévnek. Ha rövidíteni akarjuk a kódunkat, akkor használjuk a **typedef** kulcsszót.

```
1     enum color { WHITE, BLACK, RED, YELLOW, GREEN };
2
```

```
3    typedef enum color color_t;
4
5    color_t traffic_light;
```

Vagy egy lépésben is megtehetjük:

```
1   typedef
2     enum color { WHITE, BLACK, RED, YELLOW, GREEN } color_t;
3
4   color_t traffic_light;
```

Egyes esetekben név nélküli felsorolási típust is létrehozhatunk. Ilyenkor az adott típusból rögtön egy vagy több változót definiálunk:

```
1   enum { WHITE, BLACK, RED, YELLOW, GREEN } traffic_light;
2
3   traffic_light = RED;
```

A C nyelvben változatos módon használjuk fel a felsorolási típust. Van azonban egy hátrányuk, *nem lehet forward deklarálni* őket, azaz minden esetben amikor használni akarjuk őket, fel kell sorolni az összes felsorolási értéket. Emiatt a felsorolási típusokat leggyakrabban header állományokban definiáljuk és a felsorolási értékek változásakor újra kell fordítani az összes éritett forrásállományt. A C++ az **enum class** segítségével javított ezen a helyzeten.

Rekord típus

Programjainkban gyakran kell többféle adatot együtt kezelnünk: pl. egy dolgozó aznosítóját, nevét, beosztását, születési dátumát; egy előadás kódját, oktatóját, a hozzás tartozó gyakorlatok adatait (amik maguk is lehetnek összetett adatok). Ilyenkor kényelmes lenne, ha ezeket az adatokat egyetlen változóban tárolhatnánk, paraméterként átadhatnánk, egyetlen utasítással adhatnánk értéket. Ezt a fajta adatszerkezetet nevezzük **rekord**-nak, (record) vagy **struktúrának** (struct). Az *R = (R1, R2, …, Rn)* rekord a *T1*, *T2*, … *Tn* típusok direkt szorzata *R = T1 x T2 x … x Tn*.

A rekord típus összetevőit **tag**-nak (member) vagy **mező**-nek (field) nevezzük, minden tag egy névvel és típussal rendelkezik, és ez utóbbinak megfelelő műveleteket lehet elvégezni rajta. A rekord típus leggyakoribb implementációja, hogy az egyes tagok egymás után helyezkednek el a memóriában. Minden tag a rekord elejéhez képest saját távolsággal (offset) rendelkezik. Esetenként azonban a tagok között lehetnek "lyukak" (gap) is, itt nem tárolunk információt. Lyukak amiatt lehetnek, mert egyes fordítók bizonyos típusokat csak adott bájtcímekre helyezhetnek el. Ebből következően a rekord mérete nagyobb vagy egyenlő a mezők méreteinek összegével.

A rekord típussal rendszerint csak a legegyszerűbb műveleteket végezhetjük el, pl. **értékadás**, ide értve az érték szerinti paraméterátadást és függvényvisszatérést is, a rekord **címének** lekérdezése, és az egyes **tagok** (mezők) **elérése**. Miután a rekord egy mezőjét elértük, az adott mező típusának megfelelő műveleteket végezhetünk rajta.

Előfordul, hogy a rekord bizonyos része többféleképpen is lehet definiálva. Az ilyen **variadic record**-okat lentebb, az úniónál tárgyaljuk.

Az **objektum-orientált** nyelvekben az **osztályt** (class) tekinthetjük a rekordtípus olyan általánosításának, ahol az adattagok mellett a rajtuk végzett műveleteket

(tagfüggvényeket) is definiálhatjuk, illetve megadhatjuk az egyes tagok hozzáférési jogait (public, private, …).

Struct C-ben

A C programozási nyelvben a rekord (struktúra) típust a **struct** konstrukció valósítja meg. A **struct** kulcsszó része a struktúra típusnevének, azaz a változó deklarációjából nem hagyhatjuk el. A következő példában egy dátum *int* számhármassal történő lehetséges megvalósítása szerepel:

```
1    struct date
2    {
3      int year;
4      int month;
5      int day;
6    };
7
8    void f(void)
9    {
10     struct date exam = { 2018, 12, 17}; /* mezőnkénti inic. */
11     struct date *ep = &exam;          /* ep a vizsgára mutat */
12     ++exam.day;                    /* egy nappal elhalasztva */
13     (*ep).day += 2;                 /* még két nap halasztás */
14     ep->day += 3;    /* ep->day ugyanaz, mint az (*ep).day  */
15
16     assert(ep == &ep->year);  /* a struct címe azonos
17                                        az első tag címével */
18
19     struct date y2019 = {2019};  /* csak a year mezőt inic. */
20
21      /* Csak C99 óta: designator használata */
22     struct date xmas = { .month = 12, .day = 24 };   /* C99 */
23   }
```

A tagok elérését a **pont** (dot, member access) operátor teszi lehetővé. A struktúra típusú változó címét a szokásos **címoperátorral** (&) kérhetjük le, ennek a C-ben azonosnak kell lennie az első adattag (itt a year) címével. Mivel nagyon gyakori, hogy egy struktúrát a rá mutató pointeren keresztül érünk el, a **(*ptr).field** kifejezés helyett használhatjuk a rövidebb **ptr->field** jelölést. A struktúrát a tömböknél megszokott listával **{ ... }** inicializálhatjuk, illetve C99-től használhatunk inicializálást csak egyes mezőkre is.

Fontos megérteni, hogy a fenti példában az 1-6 sorok egy struktúra típust definiálnak. Ez nem egy változó, nem konkrét adatterület, nem lehet bele írni. Ez csak a dátum típus leírása, hogyan kell értelmezni a *date* összetett adatszerkezetet. Értékeket csak változókba írhatunk, azt létre kell hoznunk, pl. a 10. sorban található definicióval.

A következő példában egy alakzat típust definiálunk.

```
1    struct square
2    {
3      int centerX;
4      int centerY;
5      int side;
```

```
6   };
7
8   struct square move( struct square s, int dX, int dY)
9   {
10    s.centerX += dX;
11    s.centerY += dy;
12    return s;
13  }
14
15  void f(void)
16  {
17    struct square mySquare = { 0, 0, 10 }; /* inicializáció */
18
19    mySquare.centerX += 20;      /* mozgassuk el mySquare-t */
20    mySquare.centerY += 30;      /* a (+20,+30) vektorral   */
21
22    mySquare = move(mySquare, 20, 30);  /* uaz. függvénnyel */
23  }
```

Egy C struct változóit **értékül adhatjuk** hasonló típusú változóknak, ilyenkor az összes adattag **átmásolódik**. Ahogy a fenti példa mutatja, struct-okat átadhatunk paraméterként függvényeknek, és azok vissza is adhatnak struct-okat.Ilyenkor az érték szerinti paraméterátadás szerint a teljes struct másolódik (ellentétben a tömbökkel, ahol az első elemre mutató pointer adódik át). Ha sok adattagból álló nagyméretű struct-ot használunk, néha hatékonysági okokból a címűket adjuk át paraméterként. Struktúrákra a **relációs műveletek**, mint pl. az **==** és **!= nem értelmezett**.

```
1   struct square
2   {
3     int centerX;
4     int centerY;
5     int side;
6   };
7
8   void move2( struct square *sp, int dX, int dY)
9   {
10    sp->centerX += dX;  /* (*sp).centerX += dX */
11    sp->centerY += dY;  /* (*sp).centerY += dY */
12  }
13
14  void f(void)
15  {
16    struct square mySquare = { 0, 0, 10 }; /* inicializáció */
17
18    move2(&mySquare, 20, 30);  /* függvénnyel és pointerrel */
19  }
```

Struktúrák tartalmazhatnak további struktúrákat. Egy *személy* például rendelkezik születési dátummal:

```
1     struct person
```

```
2     {
3       char         name[40];
4       struct date  birthday;
5     };
6
7     int ask(void);
8     void f(void)
9     {
10      struct person dean;
11      strncpy( dean.name, "Horvath Zoltan", 40);
12      dean.name[39] = '\0';
13      dean.birthday.year = ask();
14    }
```

Mivel a pont operátor *balról asszociatív* (operátorok), ezért a dátum adattagjai eléréséhez nem kell zárójelezni.

Előfordul, hogy önhivatkozó adatszerkezeteket szeretnénk létrehozni. Egy *személynek* pl. lehetnek *szülei* és *gyermekei* is, akik szintén emberek. Ilyenkor a *személy* struktúra fizikailag nem tartalmazhatja önnmagát, de a logikai kapcsolatokat kifejezhetjük pointerekkel.

```
1     struct person
2     {
3       char           name[20];
4       struct date    birthday;
5       struct person *father;
6       struct person *mother;
7       struct person *children[10];
8     };
```

Még az a helyzet is előfordulhat, hogy két struktúra kölcsönösen egymásra hivatkozik. Mivel a C fordító feltételezi, hogy egy típus definíciójában a meghivatkozott nevek már definiáltak, a hivatkozási kört egy **előzetes deklarációval** (forward declaration) tudjuk feloldani:

```
1     struct manager;
2
3     struct staffmember
4     {
5       struct person   pers;
6       struct manager *boss;
7     };
8
9     struct manager
10    {
11      struct person       pers;
12      struct manager     *boss;
13      struct staffmember *staff[10];
14    };
```

Ha csak forward deklarációnk van, akkor nem tudunk a típusból változókat létrehozni: ehhez kell a struct tényleges teljes deklarációja. A fordítóprogram onnan tudja csak a

struct méretét, szerkezetét.

A legmagasabb pozícióban dolgozó főnök *boss* pointere NULL lesz.

A struct-nál is használhatjuk a **typedef** kulcsszót a rövidítéshez. Figyeljük meg, hogy a typedef csak a teljes típusdeklaráció után használható, a forward deklaráció után még nem.

```
1        typedef struct date date_t;
2
3        typedef struct person
4        {
5          char           name[20];
6          struct date    birthday;
7          struct person *father;
8          struct person *mother;
9          struct person *children[10];
10       }
11       person_t;
12
13       struct manager;
14
15       typedef struct staffmember
16       {
17         person_t        pers;
18         struct manager *boss;
19       }
20       staffmember_t;
21
22       typedef struct manager
23       {
24         person_t        pers;
25         struct manager *boss;
26         staffmember_t  *staff[10];
27       }
28       manager_t;
```

A struct **mérete** nagyobb vagy egyenlő a mezők méretével. Az egyes mezők ugyanis nem feltétlenül egymás után következnek, köztük **rések** (gap) helyezkedhetnek el. Ezért két ugyanahhoz a struct-hoz tartozó változót **soha se hasonlítsunk össze bájtonként**, erre írjunk egy mezőnkénti összehasonlítást végző függvényt.

másolódik (ellentétben a tömbökkel, ahol az első elemre mutató pointer adódik át). Ha sok adattagból álló nagyméretű struct-ot használunk, néha hatékonysági okokból a címűket adjuk át paraméterként. Struktúrákra a **relációs műveletek**, mint pl. az **==** és **!= nem értelmezett**.

```
1    typedef struct square
2    {
3      int centerX;
4      int centerY;
5      int side;
6    }
7    square_t;
```

```
8
9   int is_eq_square(const square_t *sp1, const square_t *sp2)
10  {
11    return ( sp1->centerX == sp2->centerX   &&
12             sp1->centerY == sp2->centerY   &&
13             sp1->side    == sp2->side         );
14  }
15
16  void f( square_t s1, square_t s2)
17  {
18
19    /* Ez rossz gondolat !!! */
20    if ( memcmp( &s1, &s2, sizeof(square_t) ) ) {    }
21
22    /* Igy helyes */
23    if ( is_eq_square( &s1, &s2 ) )  {   }
24  }
```

Únió típus

Egy másik összetett adatszerkezet, az **únió** (union), ami a halmazok únió jához hasonló konstrukció. Az *U = (U1, U2, …, Un)* únió, a *T1, T2, …, Tn* típuok únió ja *U = T1 u T2 u … u Tn*. Amíg a rekordban a tagokat egy időben egyszerre, egymás után tároljuk, addig az únióban csak egyetlen típust tárolhatunk egy időben, de később ezt felülírhatjuk egy másik típussal. Ebből következően az únió mérete legalább akkora, mint a legnagyobb komponense (néha technikai okokból annál hosszabb).

Az únió tagjait úgy is tekinthetjük, mint típusbiztos interfészt, amelyen keresztül beírhatunk és kiolvashatunk az únióba. Amennyiben nem a megfelelő tagot használjuk kiolvasáshoz, akkor az eredmény hibás lehet.

Statikus típusrendszer esetén azt, hogy mi volt a legutolsó értékadás típusa, vagy a programozó kell számontartsa, vagy tudhatja maga az únió típusú változó. Ez utóbbi esetben ún. **címkézett únió**-ról (tagged union) vagy **variáns** (variant record) típusról beszélünk. Ilyen tagged union létezik számos funkcionális és imperatív nyelvben, pl. a *Pascal*-ban, *Modula-2*-ben, az *Ada*-ban, a *Scala*-ban, *Rust*-ban. A *C++*-ban a C++17-es szabványtól az std::variant típus valósítja meg.

```
1   (* Pascal variant record *)
2   type shapeKind = (square, rectangle, circle);
3    shape = record
4       centerx : integer;
5       centery : integer;
6       case kind : shapeKind of
7         square : (side : integer);
8         rectangle : (lenA, lenB : integer);
9         circle : (radius : integer)
10  end;
```

```
1   -- Ada variant record (discriminated type)
2   type Shape_Kind is (Square, Rectangle, Circle);
3   type Shape (Kind : Shape_Kind) is record
4      Center_X : Integer;
```

```
5        Center_Y : Integer;
6        case Kind is
7           when Square =>
8              Side : Integer;
9           when Rectangle =>
10             LenA, LenB : Integer;
11          when Circle =>
12             Radius : Integer;
13       end case;
14    end record;
```

```
1    // C++ variant típus C++17-től,
2    // korábban használható a boost::variant
3    class Square { ... };
4    class Rectangle { ... };
5    class Circle { ... };
6
7    use Shape = std::variant<Square, Rectangle, Circle>;
8
9    std::vector<Shape> shapes;
```

Az **objektum-orientált** nyelvekben a tagged union-t gyakran *örökléssel* valósítjuk meg: az únió egy interfész vagy ha vannak közös adatok, akkor egy bázisosztály és ez egyes "variánsok" pedig a származtatott típusban valósíthatóak meg.

A **dinamikus típusrendszerű** nyelvekben nincsen szükség variant-ra, hiszen maguk az objektumok "ismerik" saját típusukat. Pythonban például írhatunk ilyet:

```
1       def f(x):
2           return 2 if x else "s"
```

Ugyanakkor a Python 3.5 által bevezetett *type hint*-ek segítségével az únió használatot explicitté lehet tenni, ami sokat segíthet külső eszközök, pl. szintaxis ellenőrzők vagy editorok használatakor.

```
1      from typing import Union,TypeVar
2
3      T = TypeVar('T')
4      def f(x: T) -> Union[int, str]:
5          if x:
6              return 2
7          else:
8              return "s"
```

Unió C-ben

A C-ben az únió típust a **union** kulcsszóval hozzuk létre. Akárcsak a struct-nál, a union kulcsszó is része a típus nevének. A union-t képzelhetjük egy olyan struct-nak, ahol az összes adattag az únió kezdőcímén kezdődik, így a tagok lényegében "átfedik" egymást.

A C nyelv nem rendelkezik tagged union típussal, az úniók tartalmának **aktív** (éppen aktuális) típusát a programozónak kell számon tartania. Ezt gyakran úgy valósítjuk meg, hogy együttesen használjuk a **struct** és **union** konstrukciókat és esetleg a felsorolási típust is:

```
1    typedef struct square
2    {
3      int centerX;
4      int centerY;
5      int side;
6    }
7    square_t;
8
9    typedef struct rectangle
10   {
11     int centerX;
12     int centerY;
13     int lenA;
14     int lenB;
15   }
16   rectangle_t;
17
18   typedef struct circle
19   {
20     int centerX;
21     int centerY;
22     int radius;
23   }
24   circle_t;
25
26   typedef enum shape_tag
27   {
28      square_tag, rectangle_tag, circle_tag
29   }
30   shape_tag_t;
31
32   typedef struct shape
33   {
34     shape_tag_t tag;
35     union shapeKind
36     {
37       square_t    s;
38       rectangle_t r;
39       circle_t    c;
40     }   u;
41   }
42   shape_t;
43
44   void print_shape( shape_t s);
45
46   void f(void)
47   {
```

```
48    circle_t cir = { 0, 0, 100 };
49
50    shape_t  s;
51    s.tag = circle_tag;
52    s.u.c = cir;
53
54    print_shape( s );
55  }
56
57  void print_shape(shape_t s)
58  {
59    switch( s.tag )
60    {
61    default:         printf( "Unknown shape\n");
62                     break;
63    case square_tag: printf( "Square: %d %d %d\n",
64                              s.u.s.centerX, s.u.s.centerY,
65                              s.u.s.side);
66                     break;
67    case rectangle_tag: printf( "Rectangle: %d %d %d %d\n",
68                              s.u.r.centerX, s.u.r.centerY,
69                              s.u.r.lenA, s.u.r.lenB);
70                     break;
71    case circle_tag: printf( "Circle: %d %d %d\n",
72                              s.u.c.centerX, s.u.c.centerY,
73                              s.u.c.radius);
74                     break;
75    }
76  }
```

Látszik, hogy a shapeKind únió nevet sehol sem használjuk a programban. Valójában el is hagyhatjuk, így egy név nélküli (anonym) úniót hozunk létre **u** tagnévvel. Mivel az *s*, *r* és *c* tagnevek csak az *u* union tagban fordulnak elő, ezért használatukkor az *u* el is hagyható:

```
1    void f(void)
2    {
3      circle_t cir = { 0, 0, 100 };
4
5      shape_t  s;
6      s.tag = circle_tag;
7      s.c = cir;
8
9      printf( "%d %d %d\n",
10               s.c.centerX, s.c.centerY, s.c.radius);
11   }
```

Ahogy korábban volt róla szó, a tag-ek lényegében “típusbiztos kapuk”, amelyen keresztül elérjük az únióban eltárolt valamely értéket. Ha nem az a tagot használjuk olvasásra, amin keresztül legutóbb írtunk az únió értékét, hibát kaphatunk. Néha mégis szándékosan csinálunk ilyet, pl. ha egy tpus értékét a bitek megváltoztatása nélkül egy másik típusban akarunk tárolni. Ilyen eset lehet, pl. ha egy bináris adatot

akarunk hálózaton átküldeni, és az únió egyik tagja egy kellően hosszú karaktertömb, amit a hálózati függvény fog továbbítani.

Vigyázat!, ez nem egy klasszikus értelemben vett típuskonverzió, hiszen a bitek értéke nem változik meg. Ha pl. egy double és int adattagú únióba beírunk egy double számot és egészként olvassuk ki, akkor a lebegőpontos ábrázolás első pár bájtját próbálnánk (valószínűleg értelmetlen) egész számként értelmezni.

Homepage of Dr. Zoltán Porkoláb

Home

Archive

Teaching

Timetable

Bolyai College

C++ (for mathematicians)

Imperative programming (BSc)

Multiparadigm programming (MSc)

Programming (MSc Aut. Sys.)

Programming languages (PhD)

Software technology lab

Theses proposals (BSc and MSc)

Research

CodeChecker

CodeCompass

Templight

Projects

Conferences

Publications

PhD students

Affiliations

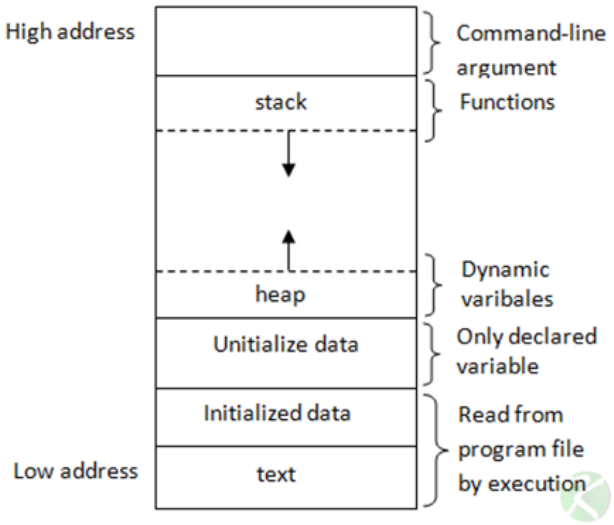
Dept. of Programming Languages and Compilers

Ericsson Hungary Ltd

Imperatív programozás 9.

Dinamikus memóriakezelés

A C programok statikus, automatikus és dinamikus élettartamú tárterületeket hozhatnak létre (Élettartam). A globális és lokális statikus változók élettartama a program elejétől a végéig tart. A nem statikus lokálisak a veremben jönnek létre az őket deklaráló blokkba való belépéskor és felszámolódnak, amikor kilépünk a blokkból. Lehetséges azonban, hogy ad-hoc módon kell tárterületet foglalnunk, vagy előre nem ismerjük a szükséges méretet. Ilyenkor használjuk a dinamikus élettartamot.



A **statikus** élettartamú változók a futásra betöltött program egy direkt erre fentartott területén jönnek létre - külön az inicializált és a nem inicializált, ez utóbbi automatikusan zéró inicializált lesz. A lokális **automatikus** élettartamú változókat a végrehajtási veremben foglaljuk le. Minden egyes függvényhívás egy **aktivizációs rekordot** hoz létre, a függvény visszatérésekor ez felszámolódik.

A **dinamikus** élettartamú objektumokat a **szabad memóriában** (free store, heap) foglaljuk le. Majdnem minden imperatív nyelv rendelkezik szabad memóriával, de annak használata erősen változó. Pl. Java-ban, C#-ban objektumokat csak a szabad memóriában tudunk létrehozni a *new* művelet segítségével. Ami lokális változó(nak) látszik, az is csak egy *referencia* (valójában egy pointer), maga az objektum a heap-ben jön létre. A C nyelvben minden objektum (egyszerű vagy összetett típusú) létrejöhet a statikus, az automatikus és a dinamikus tárterületen is.

A tárterület lefoglalása a **malloc(size_t sz)** függvénnyel történik, melynek a lefoglalandó bájtok számát adjuk meg. Hasznos, ha a paramétert a **sizeof** operátoron keresztül adjuk meg, nem pedig "beégetjük" a programba. A visszatérő érték típusa **void ***, amit a kívánt típusra mutató pointerré kell konvertálnunk. A malloc garantálja, hogy a lefoglalt tárterületen tetszőleges típusú objektum tárolható. Viszont a tárterület nem lesz inicializálva, lényegében "memória-szemetet" fog tartalmazni.

```
1    void alloc1( int nelem)
2    {
3      void *ptr = malloc( nelem * sizeof(int) );
4
5      if ( NULL != ptr )
6      {
```

```
7         /* sikeres memóriafoglalás, de inicializálatlan */
8         int *ip = (int *)ptr;
9         /* memória ip-n keresztüli típushelyes használata */
10        free( ip );   /* memória visszaadása */
11      }
12      else
13      {
14        /* memóriafoglalás sikertelen volt */
15      }
16    }
```

Előfordulhat, hogy nem sikerül lefoglalni a kívánt mennyiségű memóriát. Ekkor a malloc() **NULL** pointer értékkel tér vissza. A programozó fontos kötelessége, hogy **mindig ellenőrizze** a malloc() visszatérő értékét, mert a nullpointer dereferálása futási idejű hibát okoz!

Ha a tárterületet csupa nulla bájtra szeretnénk inicializálni, akkor ehhez a **calloc(size_t num, size_t sz)** függvényt használjuk, ami *num* darab *sz* méretű objektumnak foglal helyet és a területet nullára inicializálja.

```
1     void alloc2( int nelem)
2     {
3       void *ptr = calloc( nelem, sizeof(int) );
4
5       if ( NULL != ptr )
6       {
7         /* sikeres memóriafoglalás, csupa zéró */
8         int *ip = (int *) ptr;
9         /* memória ip-n keresztüli típushelyes használata */
10        free( ip );   /* memória visszaadása */
11      }
12      else
13      {
14        /* memóriafoglalás sikertelen volt */
15      }
16    }
```

Ugyanakkor a tárterület csupa nulla bájtra inicializálása *elvileg* nem garancia arra, hogy pl. egy lebegőpontos érték 0.0 vagy egy pointer NULL értékű lesz. Gyakorlatilag azonban a platformok többségén ez így van.

A tárterület lefoglalva marad, amíg azt a **free(ptr)** függvénnyel vissza nem adjuk a szabad memória számára. A *ptr* mindenképpen egy malloc(), calloc(), vagy *realloc()* függvény által adott kell legyen. Ha a tárterület visszaadásáról megfeledkezünk, akkor **elfogyhat a szabad memória**. Ilyenkor a malloc(), calloc(), realloc() függvények soron következő hívásai NULL pointert adnak vissza. Mindig **ellenőrizzük** ezen függvények **visszatérő értékét**!

Ellentétben más programozási nyelvekkel, a C-ben (és C++-ban) **nincsen automatikus szemétgyűjtés** (garbage collection), azaz a már nem használt, akár már hivatkozás hiányában el sem érhető tárterületek sem szabadulnak fel automatikusan. Ehhez **mindig a free()** függvény **meghívása** kell. Azt a jelenséget, amikor az elérhetetlen tárterület a programozó hibájából lefoglalva marad, nevezzük **memória elszivárgásnak (memory leak)**. Hosszan futó programok esetében (pl.

egy szerver program vagy maga az operációs rendszer) a memóriaelszivárgás súlyos gondokat okozhat.

Mivel a malloc(), calloc(), realloc() visszatérő értékét ellenőriznünk kell, gyakori a következő programozási technika:

```
1   void alloc3( int nelem)
2   {
3     double *dp;
4
5     if ( dp = (double *)malloc(nelem * sizeof(double)) )
6     {
7       int i;
8       for ( i = 0; i < nelem; ++i)  dp[i] = i;
9       /* dp felhasználása */
10      free( dp );
11    }
12  }
```

A lefoglalt területet nem kötelező abban a függvényben felszabadítani, ahol lefoglaltuk. Gyakori, hogy külön függvényekben foglaljuk le a területet és máshol szabadítjuk fel. Ez ugyanakkor megnehezítheti a program megértését.

A dinamikusan lefoglalt tárterületet a **realloc(void *ptr, size_t newsize)** függvénnyel tudjuk **átméretezni**. A *ptr* az egy előzőleg malloc(), calloc(), realloc() által lefoglalt tárterület, a *newsize* pedig a kívánt új méret. Az új méret lehet nagyobb, vagy kisebb is a jelenleg lefoglaltnál. Növeléskor a realloc() megpróbálja helyben megnövelni a tárterületet, az új terület inicializálatlan lesz. Ha ez nem sikerül, akkor más helyen próbál lefoglalni a tárterületet, **odamásolja a régi terület tartalmát**, majd felszabadítja a régi területet. Ha nem sikerül az új helyfoglalás, akkor NULL pointer tér vissza és a régi terület megőrződik, siker esetén az aktuális (akár új) területre mutató pointer tér vissza.

```
1   void alloc5( double *dp, int n)
2   {
3     /* ki akarjuk terjeszteni a területet n double-ra */
4     double *extra = (double *)realloc( dp, n*sizeof(double));
5     if ( extra )
6     {
7       /* sikerült az extra terület lefoglalása, használjuk */
8       free(extra); /* az új terület felszámolása */
9     }
10    else
11    {
12      free(dp); /* az eredeti terület felszámolása */
13    }
14  }
15  void alloc4( int nelem)
16  {
17    double *dp;
18    if ( dp = (double *)malloc(nelem * sizeof(double)) )
19    {
20      int i;
21      for ( i = 0; i < nelem; ++i)  dp[i] = i;
```

```
22        /* dp felhasználása */
23        alloc5(dp, nelem+10);/* 10 elemet hozzá akarunk adni */
24        /* alloc5 felszámolta a memóriát, itt nincs free() */
25      }
26    }
```

A realloc() elvileg alkalmas a lefoglalt terület csökkentésére is, azonban implementáció függő, hogy valóban visszaad-e memóriát.

A realloc() **veszélyes lehet**, ha az átméretezendő tárterületünkre más mutatók hivatkoznak. Ilyenkor elmozgatva a hivatkozott tárterületet az oda mutató pointerek érvénytelenné válnak.

Megjegyzések

1. A szabad memória számon tartja, hogy mekkora területet foglaltunk le, ezt a free()-nek nem kell megadnia. Az információ a heap-ben, a *ptr* környékén tárolódik, többek közt ez az oka, hogy ha nem a helyes pointerrel akarunk felszabadítani, az futási idejű hibát fog okozni.

2. Vannak platformok, ahol a programban több heap is létezik, pl. a főprogramban és egyes dinamikus library-kben (.DLL, .so) is. Ilyenkor fontos, hogy ugyanaz a programrész szabadítsa fel a memóriát, aki lefoglalta, különben esetleg a *free* függvény rossz heap-be próbálná visszaadni a tárterületet. Ez futási idejű hibához vezethet.

3. Úgy is elfogyhat a memória, ha a különböző méretű lefoglalások és felszabadítások során **feldarabolódik** (fragmented) a heap. Ilyenkor lenne még elég szabad bájt, csak nincsen elég egyetlen összefüggő területen. A mai modern szabad memória implementációk számos módon próbálják ennek a lehetőségét csökkenteni. Érdekes megoldás a *.NET* framework-é, ott a *managed* heap feltömöríti a használt területet, így eltüntetve a lyukakat, egy összefüggő memóriaterületet hoz létre a szabad területből. Ilyenkor azonban az átmozgatott területre mutató pointerek értékeit is módosítani kell (*managed pointer*), ami komoly hatékonyságcsökkentő tevékenység. Egy másik megoldás, amit a C allokátorok is alkalmazhatnak, hogy a heapben külön területeket tartanak fel a kis, közepes és nagy memóriaterületek foglalására, illetve gyakran 2 hatvány bájtot foglalnak le, így a szomszédos felszabadított területek összevonása is 2 hatvány lesz.

4. Ha egy területet felszabadítottunk, akkor tilos ismételten felszabadítani. Ilyen hibát akkor szoktunk kapni, ha egy pointert lemásolunk és mindkét helyen fel akarjuk szabadítani a mutatott tárterületet. Ez gyakran az ún. **sekély másolás** (shallow copy) eredménye, amikor a pointert másoljuk a mutatott tárterület helyett.

5. A felszabadított memóriára tilos hivatkozni. Az **indirekció** (*) operátor használata ilyenkor futási idejű hibát okozhat.

6. A *free()* függvénynek átadhatunk **NULL** pointert, ilyenkor a hatása az üres utasítással ekvivalens. A free() viszont nem állítja NULL-ra az átadott pointer értékét.

7. A heap műveletek *szálbiztosak*, azaz többszálú programban is biztonsággal használhatjuk őket. Viszont ez azt jelenti, hogy a heap használata a háttérben lock-olásokkal jár, ami csökkenti a hatékonyságot.

8. Gyakori hiba, hogy egy C string másolása számára akarunk helyet foglalni az *strlen* függvény visszatérő értéke alapján. Azonban ha pont annyi helyet foglalunk

le, akkor nem marad hely a lezáró bináris NUL karakterre. Ezt a hibát hívják **off by one** (OBOE, OB1) errornak. Helyesen használjuk a **malloc(strlen(source)+1)** kifejezést.

9. A dinamikus élettartammal kapcsolatos függvények az **<stdlib.h>** headerben vannak deklarálva.

Példák összetett adatszerkezetek dinamikus kezelésére

Verem implmentálása

A verem egy *utolsónak be - elsőnek ki* (last in - first out, LIFO) adatszerkezet. Elsőnek próbáljuk meg egy fix méretű tömbbel implementálni.

```
1    #include <stdio.h>
2    #define CAPACITY 10
3
4    void print_reverse( double *stack, int size)
5    {
6      while ( --size >= 0 )  /* stack[size-1] ... stack[0] */
7      {
8        printf( "%f\n", stack[size]);
9      }
10   }
11   int main()
12   {
13     double  d;
14     double  stack[CAPACITY];   /* a buffer */
15     int     size = 0;          /* a beszúrt elemek száma */
16
17     while ( size < CAPACITY  &&  1 == scanf("%lf", &d) )
18     {
19       stack[size++] = d;
20     }
21     printf("==== eredmeny ====\n");
22     print_reverse( stack, size );  /* elemek kiírása */
23     return 0;
24   }
```

A programot lefordítva, lefuttatva megadhatjuk az input lebegőpontos számokat, EOF-ig.

```
$ gcc -ansi -pedantic -W fixarray.c -o fixarray
$ ./fixarray
2.3
4.5
7.8
==== eredmeny ====
7.800000
4.500000
2.300000

$ ./fixarray
```

```
1 2 3 4 5 6 7 8 9 0 1 2
==== eredmeny ====
0.000000
9.000000
8.000000
7.000000
6.000000
5.000000
4.000000
3.000000
2.000000
1.000000
```

Működik a program, de legfeljebb 10 elemet tudunk berakni a verembe. Kevesebbet lehet, EOF-ig írhatunk számokat, akkor is jól működik.

Szeretnénk azonban tetszőleges hosszúságú számsorozatokat is berakni a verembe. Ehhez *dinamikus memóriafoglalást* fogunk használni. Most is egy fix kezdőmérettel indulunk, de amikor betelik a verem, akkor megnöveljük a méretét.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   const int INIT_CAP = 10;
5
6   void nomem_error()
7   {
8     fprintf( stderr, "Fatalis hiba: nincs memoria\n");
9     exit(1);
10  }
11  void print_reverse( double *stack, int size)
12  {
13    while ( --size >= 0 )  /* stack[size-1] ... stack[0] */
14    {
15      printf( "%f\n", stack[size]);
16    }
17  }
18  /* megduplázzuk a buffert, és átírjuk a kapacitást */
19  void grow( double **pdp, int *pcap)
20  {
21    double *pnew =
22            (double *)realloc(*pdp, *pcap*2*sizeof(double));
23    if ( NULL == pnew )
24      /* noreturn */ nomem_error();
25    *pdp  = pnew; /* visszaírjuk az új pointert */
26    *pcap *= 2;   /* update-eljük a kapacitást  */
27  }
28  int main()
29  {
30    double  d;
31    int     cap   = INIT_CAP;  /* a buffer mérete */
32    int     size  = 0;         /* beszúrt elemek száma */
33    double *stack = (double *)malloc(cap * sizeof(double));
```

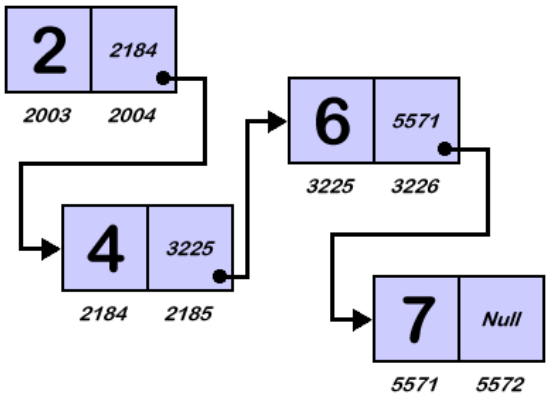
```
34      if ( NULL == stack )
35        /* noreturn */ nomem_error();
36
37      while ( 1 == scanf("%lf", &d) )
38      {
39        if ( size == cap )  /* betelt a buffer */
40        {
41          grow( &stack, &cap);  /* megnöveljük a buffert */
42        }
43        stack[size++] = d;
44      }
45      printf("==== eredmeny ====\n");
46      print_reverse( stack, size );  /* elemek kiírása */
47      return 0;
48    }
```

A programot lefordítva, lefuttatva megadhatjuk az input lebegőpontos számokat, EOF-ig.

```
$ gcc -ansi -pedantic -W stack.c -o stack
$ ./a.out
2.3
4.5
7.8
==== eredmeny ====
7.800000
4.500000
2.300000
```

Verem implementálása listával

A lista az egyik legegszerűbb dinamikus adatszerkezet. A legegyszerűbb esetben a listaelemeket egy irányba fűzzük egymáshoz.



Listával implementálhatunk pl. verem adatszerkezetet is.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   struct elem
5   {
```

```
6       struct elem *next;
7       double       value;  /* payload */
8     };
9
10    void nomem_error()
11    {
12      fprintf( stderr, "Fatalis hiba: nincs memoria\n");
13      exit(1);
14    }
15    /* lista elemeinek kiírása előlről hátra */
16    void print_reverse( struct elem *head )
17    {
18      while ( head  )   /* NULL != head */
19      {
20        printf( "%f\n", head->value);
21        head = head->next;
22      }
23    }
24    int main()
25    {
26      struct elem *head = NULL;  /* fejelem */
27      double d;
28
29      while ( 1 == scanf("%lf", &d) )
30      {
31        struct elem *pnew =
32                 (struct elem *)malloc(sizeof(struct elem));
33        if ( NULL == pnew )
34          /* noreturn */ nomem_error();
35
36        /* első elemként befűzzük az új elemet */
37        pnew->next  = head;
38        pnew->value = d;
39        head = pnew;  /* az új elemre mutat a fejelem */
40      }
41      printf("==== eredmeny ====\n");
42      print_reverse( head );  /* elemek kiírása */
43      return 0;
44    }
```

A programot lefordítva, lefuttatva megadhatjuk az input lebegőpontos számokat, EOF-ig.

```
$ gcc -ansi -pedantic -W list.c -o list
$ ./a.out
2.3
4.5
7.8
==== eredmeny ====
7.800000
4.500000
2.300000
```

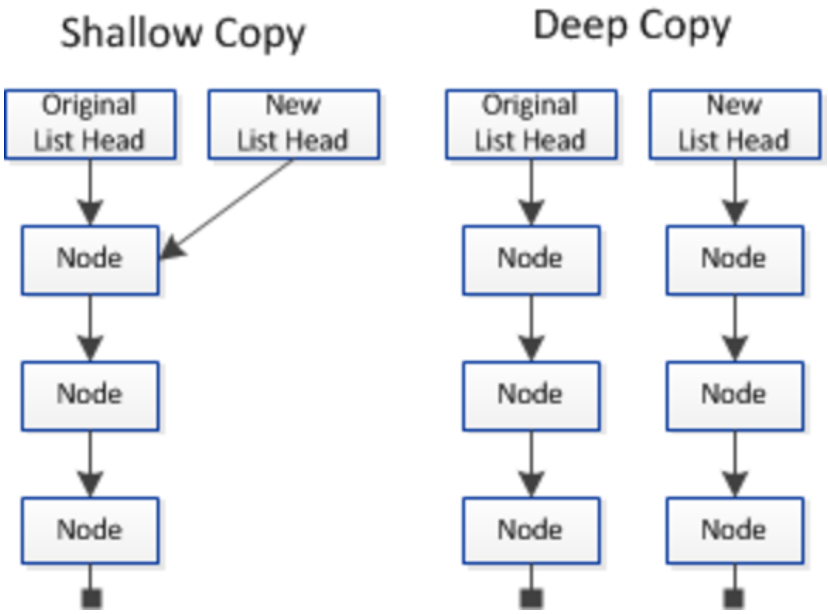
Másolás

Tételezzük fel, hogy szeretnénk másolatot készíteni a felépített listánkról. Mi történik, ha a következőt csináljuk:

```
1   int main()
2   {
3     struct elem *head = NULL;  /* fejelem */
4     struct elem *copy = NULL;  /* ez lesz a másolat */
5     double d;
6
7     while ( 1 == scanf("%lf", &d) )
8     {
9       /* a "head" lista feltöltése */
10      /* ... */
11    }
12    copy = head;  /* mi történik ilyenkor? */
13    /* ... */
14  }
```

A **copy = head** csak a head mutatót másolja át, így a "másolat" igazából az eredeti lista lesz. Ezt a megoldást nevezzük **sekély másolásnak** (shallow copy). A sekély másolás gyors, de a viselkedése megtévesztő is lehet, mert most a listának két *tulajdonosa* is van (head és copy). Ha pl. a copy pointeren keresztül módosítjuk a lista valamely elemét, vagy beszúrunk egy új elemet (a legelsőtől különböző pozícióba), akkor a *másik* lista is változik. A sekély másolást leginkább akkor használjuk, ha a több tulajdonos nem probléma, pl. azért mert a lista nem módosulhat.

Ha a listát ténylegesen, minden elemével le akarjuk másolni, akkor **mély másolást** (deep copy) kell csinálnunk. A mély másolás során a teljes adatszerkezetet, annak minden elemével le kell másolni, és ennek a másolatnak lesz a birtokosa a copy változó.




```
1   struct elem *deep_copy( struct elem *source)
2   {
3     struct elem *copy = NULL;
```

```
4       struct elem *last  = NULL;
5
6       while ( source )
7       {
8         struct elem *pnew =
9                   (struct elem *) malloc(sizeof(struct elem));
10        if ( NULL == pnew )
11          /* noreturn */ nomem_error(); /* fatális hiba */
12
13        if ( NULL == copy )
14          copy = pnew;      /* csak egyszer, az első elemnél */
15        else
16          last->next = pnew; /* eddigi utolsó után befűzzük */
17
18        last        = pnew;  /* kurrens utolsó a másolatban */
19        last->value = source->value;/* átmásoljuk az értéket */
20        last->next  = NULL;       /* ez most a másolat vége */
21        source      = source->next; /* következő a forrásban */
22      }
23      return copy;
24    }
25
26
27    int main()
28    {
29      struct list *head = NULL;  /* fejelem */
30      struct list *copy = NULL;  /* ez lesz a másolat */
31      double d;
32
33      while ( 1 == scanf("%lf", &d) )
34      {
35        /* a "head" lista feltöltése */
36        /* ... */
37      }
38      copy = head;             /* shallow copy */
39      copy = deep_copy(head); /* deep copy */
40      /* ... */
41    }
```

A dinamikus memóriakezelés ellenőrzése

Vajon memória-elszivárgás mentesek-e a programjaink?

A program futását ellenőrizhetjük valamely dinamikus analízis eszközzel, pl. a **google memory sanitizer**-rel vagy a **valgrind**-al.

```
$ valgrind ./fixarray
==5860== Memcheck, a memory error detector
==5860== Command: ./fixarray
==5860==
1 2 3 4 5 6 7 8 9 0 1 2 3 4
==== eredmeny ====
```

```
0.000000
9.000000
8.000000
7.000000
6.000000
5.000000
4.000000
3.000000
2.000000
1.000000
==30117==
==30117== HEAP SUMMARY:
==30117==     in use at exit: 0 bytes in 0 blocks
==30117==   total heap usage: 4 allocs, 4 frees, 80 bytes allocated
  while ( head )
  {
    struct elem *first =  head;   /* az első elem */
    head = head->next;     /* kifűzzük a listából */
    free( first );                    /* töröljük */
  }
==30117==
==30117== All heap blocks were freed -- no leaks are possible
==30117==
==30117== For counts of detected and suppressed errors, rerun with: -
v
==30117== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from
0)
```

A fixméretű tömbbel megírt programunk érthetően rendben van a dinamikus memória használata szempontjából. A malloc/realloc-ot használó megoldásunk azonban a program végéig nem adott vissza minden memóriát. Nyilvánvalóan, a program befejeztekor nem számoltuk fel a(z esetleg közben többször is átméretezett) buffert.

```
$ valgrind ./stack
==6046== Memcheck, a memory error detector
==6046== Command: ./stack
==6046==
1 2 3 4 5 6
==== eredmeny ====
6.000000
5.000000
4.000000
3.000000
2.000000
1.000000
==6046==
==6046== HEAP SUMMARY:
==6046==     in use at exit: 80 bytes in 1 blocks
==6046==   total heap usage: 3 allocs, 2 frees, 2,128 bytes allocated
==6046==
==6046== LEAK SUMMARY:
==6046==    definitely lost: 80 bytes in 1 blocks
==6046==    indirectly lost: 0 bytes in 0 blocks
```

```
==6046==      possibly lost: 0 bytes in 0 blocks
==6046==    still reachable: 0 bytes in 0 blocks
==6046==         suppressed: 0 bytes in 0 blocks
==6046== Rerun with --leak-check=full to see details of leaked memory
==6046==
==6046== For counts of detected and suppressed errors, rerun with: -v
==6046== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from
0)

$ valgrind ./stack
==8900== Memcheck, a memory error detector
==8900== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et
al.
==8900== Using Valgrind-3.14.0 and LibVEX; rerun with -h for
copyright info
==8900== Command: ./stack
==8900==
1 2 3 4 5 6 7 8 9 0 1 2 3 4
==== eredmeny ====
4.000000
3.000000
2.000000
1.000000
0.000000
9.000000
8.000000
7.000000
6.000000
5.000000
4.000000
3.000000
2.000000
1.000000
==8900==
==8900== HEAP SUMMARY:
==8900==     in use at exit: 160 bytes in 1 blocks
==8900==   total heap usage: 4 allocs, 3 frees, 2,288 bytes allocated
==8900==
==8900== LEAK SUMMARY:
==8900==    definitely lost: 160 bytes in 1 blocks
==8900==    indirectly lost: 0 bytes in 0 blocks
==8900==      possibly lost: 0 bytes in 0 blocks
==8900==    still reachable: 0 bytes in 0 blocks
==8900==         suppressed: 0 bytes in 0 blocks
==8900== Rerun with --leak-check=full to see details of leaked memory
==8900==
==8900== For counts of detected and suppressed errors, rerun with: -v
==8900== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from
0)

$ valgrind  ./list
==7722== Memcheck, a memory error detector
```

```
==7722== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et
al.
==7722== Using Valgrind-3.14.0 and LibVEX; rerun with -h for
copyright info
==7722== Command: ./list
==7722==
1 2 3 4 5 6 7 8 9 0 1 2 3 4
==== eredmeny ====
4.000000
3.000000
2.000000
1.000000
0.000000
9.000000
8.000000
7.000000
6.000000
5.000000
4.000000
3.000000
2.000000
1.000000
==7722==
==7722== HEAP SUMMARY:
==7722==     in use at exit: 224 bytes in 14 blocks
==7722==   total heap usage: 16 allocs, 2 frees, 2,272 bytes
allocated
==7722==
==7722== LEAK SUMMARY:
==7722==    definitely lost: 16 bytes in 1 blocks
==7722==    indirectly lost: 208 bytes in 13 blocks
==7722==      possibly lost: 0 bytes in 0 blocks
==7722==    still reachable: 0 bytes in 0 blocks
==7722==         suppressed: 0 bytes in 0 blocks
==7722== Rerun with --leak-check=full to see details of leaked memory
==7722==
==7722== For counts of detected and suppressed errors, rerun with: -v
==7722== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from
0)
```

Amikor egy program befejeződik, a teljes lefoglalt memóriája felszabadul és visszakerül az operációs rendszerhez. Ha azonban a mostani programunkat egy nagyobb és hosszabb ideig futó rendszer részeként szeretnénk felhasználni, lehetnének bajok a memória elszivárgásból.

Javítsuk ki a programjainkat: egészítsük ki a főprogram végén a memória felszabadításával. A realloc()-os programunk végén egyszerűen felszabadítjuk a buffert.

```
1    int main()
2    {
3      /* idáig változatlan a main() */
4      printf("==== eredmeny ====\n");
```

```
5      print_reverse( stack, size );  /* elemek kiírása */
6
7      free( stack );  /* felszabadítjuk a memóriát */
8      return 0;
9    }
```

A listás megoldásnál egy kicsit többet kell dolgoznunk. Írunk egy függvényt, ami egyesével felszabadítja a listaelemeket (mindig a legelsőt).

```
1    void free_list( struct elem *head)
2    {
3      while ( head )
4      {
5        struct elem *first =  head;   /* az első elem */
6        head = head->next;     /* kifűzzük a listából */
7        free( first );                    /* töröljük */
8      }
9    }
10   int main()
11   {
12     /* idáig változatlan a main() */
13     printf("==== eredmeny ====\n");
14     print_reverse( stack, size );  /* elemek kiírása */
15
16     free_list( stack );   /* felszámoljuk a memóriát */
17     return 0;
18   }
```

További példa: Alakzatok

A felhasznált "alakzat" típusokat az (Összetett adatszerkezetek előadáson tárgyaltuk részletesen.

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    struct square { int centerX; int centerY; int side; };
5    struct rectangle { int centerX; int centerY;
6                       int lenA; int lenB; };
7    struct circle { int centerX; int centerY; int radius; };
8
9    typedef struct square    square_t;
10   typedef struct rectangle rectangle_t;
11   typedef struct circle    circle_t;
12
13   enum shape_tag_t { square_tag, rectangle_tag, circle_tag };
14
15   typedef struct shape /* a shape on canvas */
16   {
17     int              id;
18     enum shape_tag_t tag;
```

```
19      union shapeKind
20      {
21        square_t    s;
22        rectangle_t r;
23        circle_t    c;
24      }   u;
25    } shape_t;
26
27    typedef struct elem  /* one node in object list */
28    {
29      shape_t     *shp;
30      struct elem *next;
31    } elem_t;
32
33    typedef struct canvas  /* the canvas for shapes */
34    {
35      int     count;
36      elem_t *first;
37    } canvas_t;
38
39    int genId()
40    {
41      static int i = 0;
42      return i++;
43    }
44
45    shape_t *createSquare( int x, int y, int side)
46    {
47      shape_t *sp = (shape_t *)malloc(sizeof(shape_t));
48      if ( NULL == sp )
49        return NULL;
50      sp->id        = genId();
51      sp->tag       = square_tag;
52      sp->u.s.centerX = x;
53      sp->u.s.centerY = y;
54      sp->u.s.side    = side;
55      return sp;
56    }
57
58    shape_t *createRectangle( int x, int y, int a, int b)
59    {
60      shape_t *sp = (shape_t *)malloc(sizeof(shape_t));
61      if ( NULL == sp )
62        return NULL;
63      sp->id        = genId();
64      sp->tag       = rectangle_tag;
65      sp->u.r.centerX = x;
66      sp->u.r.centerY = y;
67      sp->u.r.lenA    = a;
68      sp->u.r.lenB    = b;
69      return sp;
70    }
```

```
71
72    shape_t *createCircle( int x, int y, int r)
73    {
74      shape_t *sp = (shape_t *)malloc(sizeof(shape_t));
75      if ( NULL == sp )
76        return NULL;
77      sp->id        = genId();
78      sp->tag       = circle_tag;
79      sp->u.c.centerX = x;
80      sp->u.c.centerY = y;
81      sp->u.c.radius  = r;
82      return sp;
83    }
84
85    char *sShapeKind(enum shape_tag_t tg)
86    {
87      switch( tg )
88      {
89      case    square_tag: return "square";
90      case rectangle_tag: return "rectangle";
91      case    circle_tag: return "circle";
92      default           : return "unknown";
93      }
94    }
95
96    void printShape(shape_t *sp)
97    {
98      printf("id = %d, type = %s\n", sp->id, sShapeKind(sp-
99    >tag));
100   }
101
102   void initCanvas( canvas_t *cp)
103   {
104     cp->count = 0;
105     cp->first = NULL;
106   }
107
108   void addCanvas( canvas_t *cp, shape_t *sp)
109   {
110     elem_t *newElem = (elem_t *)malloc(sizeof(elem_t));
111     newElem->shp = sp;
112     newElem->next = NULL;
113     if ( 0 == cp->count )
114       cp->first = newElem;
115     else
116     {
117       elem_t *p = cp->first;
118       while ( NULL != p->next )
119         p = p->next;
120       p->next = newElem;
121     }
122     ++cp->count;
```

```
123   }
124   void deleteCanvas( canvas_t *cp)
125   {
126     elem_t *ep = cp->first;
127     while ( NULL != ep )
128     {
129       elem_t *ptrToDel = ep;
130       ep = ep->next;
131
132       free( ptrToDel->shp);
133       free( ptrToDel);
134     }
135   }
136
137   void printCanvas( canvas_t *cp)
138   {
139     elem_t *ep = cp->first;
140     printf( "num of shapes = %d\n", cp->count);
141
142     while ( NULL != ep )
143     {
144       printShape(ep->shp);
145       ep = ep->next;
146     }
147   }
148
149   int main()
150   {
151     canvas_t cvs;
152
153     initCanvas( &cvs);
154
155     addCanvas( &cvs, createCircle( 10, 10, 20));
156     addCanvas( &cvs, createRectangle( 14, 20, 10, 30));
157
158     printCanvas( &cvs);
159     deleteCanvas( &cvs);
160
161     return 0;
      }
```

```
$ ./a.out
num of shapes = 2
id = 0, type = circle
id = 1, type = rectangle
```

A program futását ellenőrizhetjük valamely dinamikus analízis eszközzel, pl. a **google memory sanitizer**-rel vagy a **valgrind**-al.

```
valgrind ./a.out
==30117== Memcheck, a memory error detector
==30117== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et
```

```
al.
==30117== Using Valgrind-3.10.1 and LibVEX; rerun with -h for
copyright info
==30117== Command: ./a.out
==30117==
num of shapes = 2
id = 0, type = circle
id = 1, type = rectangle
==30117==
==30117== HEAP SUMMARY:
==30117==     in use at exit: 0 bytes in 0 blocks
==30117==   total heap usage: 4 allocs, 4 frees, 80 bytes allocated
==30117==
==30117== All heap blocks were freed -- no leaks are possible
==30117==
==30117== For counts of detected and suppressed errors, rerun with: -
v
==30117== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from
0)
```

Homepage of Dr. Zoltán Porkoláb

Home

Archive

Teaching

Timetable

Bolyai College

C++ (for mathematicians)

Imperative programming (BSc)

Multiparadigm programming (MSc)

Programming (MSc Aut. Sys.)

Programming languages (PhD)

Software technology lab

Theses proposals (BSc and MSc)

Research

CodeChecker

CodeCompass

Templight

Projects

Conferences

Publications

PhD students

Affiliations

Dept. of Programming Languages and Compilers

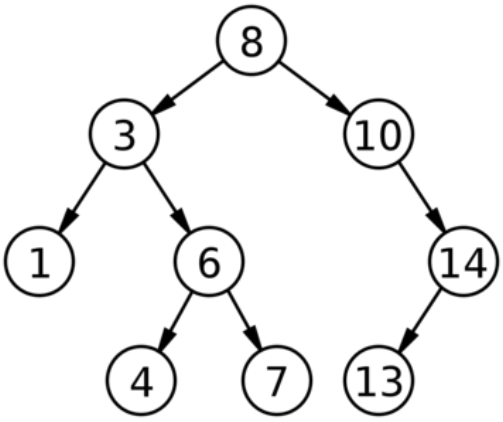
Ericsson Hungary Ltd

Imperatív programozás 9.5.

Dinamikus memóriakezelés és adatszerkezetek gyakorlása

Készítünk egy programot, amely **bináris keresőfa** (binary search tree) alkalmazásával rendezi a standard inputról érkező számokat és sorrendben kiírja őket.

A bináris keresőfa egy gyakran használt adatszerkezet, amely rendezve tárolja elemeit. A fa általánsan fennálló *invariáns* tulajdonsága, hogy minden elem alatti bal rész-fa az elemnél kisebb (vagy kisebb-egyenlő), a jobb rész-fa pedig a nagyobb elemeket tartalmazza.



A fa felépítését a rekurzív insert() függvény fogja végezni, kiíratását az rekurzív *inorder* bejárást akalmazó print() függvény, a fa lebontását pedig a rekurzív *postorder* delete() függvény.

Figyeljük meg, hogy az insert() függvény a gyökérmutató *címét* kapja meg paraméterül, hiszen amennyiben azok NULL értékűek, akkor azokat módosítania kell.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4
5   struct node   /* one node of the search tree */
6   {
7     int          value;   /* payload */
8     struct node *left;    /* pointer to left child  */
9     struct node *right;   /* pointer to right child */
10  };
11  typedef struct node node_t;
12
13  void insert(node_t **pRoot, int i);
14  void print(node_t *root);
15  void delete(node_t *root);
16
17  int main()
18  {
19    node_t *head = NULL;  /* pointer to binary search tree */
```

```
20    int i;
21    int cnt = 0;
22
23    clock_t start = clock();   /* start the timer */
24    while ( scanf("%d", &i) == 1 )
25    {
26      ++cnt;
27      insert(&head, i);     /* recursive insert */
28    }
29    clock_t end = clock();    /* stop the timer */
30    double diff = (end - start); /* elapsed time in 'tick's */
31    diff = diff / CLOCKS_PER_SEC; /* ..converted to seconds */
32
33    print(head);  /* print tree elements: recursive inorder */
34
35    fprintf(stderr,"sorted %d elems in %f sec\n", cnt, diff);
36
37    delete(head); /* delete the tree: recursive postorder */
38    return 0;
39  }
40
41  void insert( node_t **pRoot, int i)
42  {
43    if ( NULL == *pRoot )  /* need to allocate new element */
44    {
45      *pRoot = (node_t*) malloc(sizeof(node_t));
46      if ( NULL == *pRoot )
47      {
48        fprintf( stderr, "No memory\n");
49        exit(1);
50      }
51      else
52      {
53        node_t *root = *pRoot; /* make more readable */
54        root->value = i;       /* set payload */
55        root->left  = NULL;    /* no children (yet) */
56        root->right = NULL;
57      }
58    }
59    else /* not to allocate, just descent left or right */
60    {
61      node_t *root = *pRoot;  /* make more readable */
62      if ( i <= root->value )
63        insert( &root->left, i);  /* descend to left */
64      else
65        insert( &root->right, i); /* descend to right */
66    }
67  }
68  void print(node_t *root)
69  {
70    if ( root )   /* if this is a real node */
71    {
```

```
72      print(root->left);
73      printf("%d ", root->value);  /* inorder */
74      print(root->right);
75    }
76  }
77  void delete(node_t *root)
78  {
79    if ( root )   /* if this is a real node */
80    {
81      delete(root->left);
82      delete(root->right);
83      free(root);          /* postorder */
84    }
85  }
```

Írunk egy kis segédprogramot, ami véletlenszám-generátorral egész számok sorozatát fogja generálni. A generált számok mennyiségét parancssori argumentumként adjuk meg, a default érték 100.000.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4
5   int main(int argc, char **argv)
6   {
7     int i;
8     int max = 100000; /* number of elements by default */
9
10    if ( argc > 1 ) /* if command line argument is given */
11    {
12      int imax = atoi(argv[1]);  /* converts to int */
13      if ( imax > 0 )   /* if argv[1] was meaningful */
14        max = imax;
15    }
16    srand(time(NULL));   /* seeding the random generator */
17    for ( i = 0; i < max; ++i)
18      printf( "%d ", rand() );  /* int between 0..MAXINT */
19
20    return 0;
21  }
```

Megjegyzés: a random generátor inicializálása (*seeding*) a **time(NULL)** hívással általános gyakorlat. Ugyanakkor, ez nem igazán biztonságos: a time(NULL) a kurrens rendszeridővel tér vissza (az ún. UNIX epoch szerint), ami kiszámítható lehet. Biztonság-kritikus programokban ne ezt a módszert használjuk.

Végül, írunk egy kis programot, ami azt ellenőrzi, hogy a programunk tényleg rendezett outputot ír ki.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main()
```

```
5     {
6       int prev = -1;  /* ensure that the first number is ok */
7       int curr;
8
9       while ( 1 == scanf("%d", &curr) )
10      {
11        if ( curr < prev )    /* inversion!!! */
12        {
13          fprintf( stderr, "Inversion: %d %d\n", prev, curr);
14        }
15        printf("%d ", curr);
16        prev = curr;
17      }
18      return 0;
19    }
```

A programok használata például az az alábbi lehet:

```
$ ./gen 5000 | ./binsort | ./check >/dev/null
```

Homepage of Dr. Zoltán Porkoláb

Home

Archive

Teaching

Timetable

Bolyai College

C++ (for mathematicians)

Imperative programming (BSc)

Multiparadigm programming (MSc)

Programming (MSc Aut. Sys.)

Programming languages (PhD)

Software technology lab

Theses proposals (BSc and MSc)

Research

CodeChecker

CodeCompass

Templight

Projects

Conferences

Publications

PhD students

Affiliations

Dept. of Programming Languages and Compilers

Ericsson Hungary Ltd

Imperatív programozás 11.

Unix filterek implementálása

A UNIX **szűrők** (filters) olyan programok, amelyek valami elemi tevékenységet végeznek el. Ilyen filter a *cat*, *grep*, *diff* és még sok más program. Ezek a programok tipikusan a *standard inputról* olvasnak EOF-ig és az eredményt a *standard outputra* írják. Amennyiben a programok egy vagy több fájlnév paramétert kapnak, akkor a standard input heyett ezekről a fájlokról olvasnak. Az így megírt programokat kényelmesen és sokoldalúan lehet *pipe*-okba szervezni.

grep

Az alábbi a **grep** program egy leegyszerűsített megvalósítása. Az eredeti UNIX utility *reguláris kifejezések*-et keres az input állományokban, mi most egy rendkívül leegyszerűsített programot mutatunk be, amelyik egyetlen fix string előfordulását keresi a sorban.

Alapvetően így használható a program:

```
$ gr pattern file1 file2 file3
```

Ebben az esetben a **pattern** mintát keressük a *file1*, *file2*, és *file3* állományokban és kiírjuk azokat a sorokat, ahol találat van.

Megvalósítjuk a **-i** és **-v** kapcsolókat, melyek az alapvető működést befolyásolja.

```
$ gr -iv pattern file1 file2 file3
```

- A **-v** kapcsoló hatására azok a sorok íródnak ki, melyekben *nem* volt találat.
- A **-i** kapcsoló a kis-nagybetűk közötti különbséget nem veszi figyelembe.
- A **-w** kapcsoló csak olyankor jelez találatot, ha az egy *teljes szó*, azaz üreshelyekkel van körbevéve.

A grep. pontos leírása a linken található.

A teljes program

```
1   /*
2    * gr.c -- a simple grep-like program
3    * usage: gr [-ivw] pattern [files...]
4    *
5    */
6
7   #include <stdio.h>
8   #include <stdlib.h>
9   #include <string.h>
10  #include <ctype.h>
11
12  #define BUFSIZE 1024
13
14  struct param_s
15  {
```

```
16      int  iflag;     /* case insensitive on */
17      int  vflag;     /* negation on         */
18      int  wflag;     /* word regex on       */
19      char *pattern;  /* pattern to search   */
20      char *upattern; /* upper case pattern  */
21    };
22
23    void usage( char *prname)
24    {
25      fprintf( stderr,
26            "Usage: %s [-ivw] pattern [files...]\n", prname);
27      exit(1);
28    }
29
30    int do_params( struct param_s *p, int argc, char *argv[])
31    {
32      int i = 1;
33
34      p->iflag = 0;
35      p->vflag = 0;
36      p->wflag = 0;
37
38      /* letter flags first */
39      while ( i < argc  &&  '-' == argv[i][0] )
40      {
41        int j = 1;
42        while ( '\0' != argv[i][j] )
43        {
44          switch(argv[i][j])
45          {
46          case 'i': p->iflag = 1; break;
47          case 'v': p->vflag = 1; break;
48          case 'w': p->wflag = 1; break;
49          default : fprintf( stderr,
50                             "Invalid flag: %c\n", argv[i][j]);
51                    usage(argv[0]); /* invalid flag: exit() */
52          }
53          ++j;
54        }
55        ++i;
56      }
57      /* end of flags, pattern should come here */
58      if ( i >= argc )
59      {
60        fprintf( stderr, "No pattern was given\n");
61        usage(argv[0]);       /* no pattern: exit() */
62      }
63      p->pattern = argv[i];   /* ok, pattern found */
64
65      if ( p->iflag )  /* case insensitive */
66      {
67        int k;
```

```
68        p->upattern = (char *) malloc(strlen(p->pattern)+1);
69        for ( k = 0; k < strlen(p->pattern)+1; ++k)
70        {
71          p->upattern[k] = toupper(p->pattern[k]);
72        }
73      }
74      return ++i;  /* continue from next parameter */
75    }
76
77    int is_delim( char ch)
78    {
79      return !( isalpha(ch) || isdigit(ch) || '_' == ch );
80    }
81
82    int wmatch( char *buffer, char *pattern, char *where)
83    {
84      char *before = where - 1;
85      char *after =  where + strlen(pattern);
86
87      return (  where == buffer  &&  is_delim(*after) ) ||
88             (  is_delim(*before)  &&  '\n' == *after ) ||
89             (  is_delim(*before) && is_delim(*after) );
90    }
91
92    void gr( struct param_s *p, FILE *in, FILE *out)
93    {
94      char buffer[BUFSIZE];
95      while ( NULL != fgets( buffer, BUFSIZE, in) )
96      {
97        int is_match  = 0; /* true if matches */
98        char *where   = 0; /* pointer to beginning of match */
99
100       if ( p->iflag )
101       {
102         char ubuffer[BUFSIZE];
103         int k;
104         for ( k = 0; k < strlen(buffer)+1; ++k)
105         {
106           ubuffer[k] = toupper(buffer[k]);
107         }
108         is_match =
109           (NULL != (where = strstr( ubuffer, p->upattern)));
110         if ( is_match && p->wflag )
111           is_match = wmatch( ubuffer, p->upattern, where);
112       }
113       else
114       {
115         is_match =
116           (NULL != (where = strstr( buffer, p->pattern)));
117         if ( is_match && p->wflag )
118           is_match = wmatch( buffer, p->pattern, where);
119       }
```

```
120
121       if ( p->vflag )
122       {
123         is_match = ! is_match;
124       }
125
126       if ( is_match )
127       {
128         fputs( buffer, out);
129       }
130     }
131   }
132
133
134   int main( int argc, char *argv[])
135   {
136     struct param_s params;
137     int i = do_params( &params, argc, argv);
138
139   #ifdef DEBUG
140     fprintf( stderr, "iflag    = %d\n", params.iflag);
141     fprintf( stderr, "vflag    = %d\n", params.vflag);
142     fprintf( stderr, "wflag    = %d\n", params.wflag);
143     fprintf( stderr, "pattern  = %s\n", params.pattern);
144     fprintf( stderr, "upattern = %s\n", params.upattern);
145     fprintf( stderr, "i == %d\n", i);
146   #endif /* DEBUG */
147
148     if ( i == argc )
149     {
150       gr( &params, stdin, stdout);
151     }
152     else
153     {
154       for ( ; i < argc; ++i)
155       {
156         FILE *fp = fopen( argv[i], "r");
157         if ( NULL != fp )
158         {
159           gr( &params, fp, stdout);
160           fclose(fp);
161         }
162         else
163         {
164           fprintf( stderr,
165                    "Can't open %s for read\n", argv[i]);
166         }
167       }
168     }
169     return 0;
170   }
```

hexdump

Az input állomány(oka)t hexadecimális formátumban írja ki 16 bájtonként. A 16 elemű sor kiírás előtt megelenik a sor címe (szintén hexadecimálisan), utána pedig maguk a karakterek íródnak ki ASCII formátumban (a nem megjeleníthető karakterek helyett pedig egy pont karakter).

A program implementációja C-ben

```
1   #include <stdio.h>
2   #include <ctype.h>  /* az isgraph() miatt */
3
4   #define LINESIZE 16
5
6   void hd(FILE *in, FILE *out);
7   void print(FILE *fp,long addr,unsigned char *buf,int len);
8
9   int main(int argc, char *argv[])
10  {
11    int err = 0; /* ha legalább egy fájl megnyitása sikertelen
12                    akkor jelezni fogjuk ezt az exit kódban */
13    if ( argc < 2 )
14    {
15      hd( stdin, stdout);  /* nincs fájl argumentum */
16    }
17    else
18    {
19      int i;
20      for ( i = 1; i < argc; ++i )   /* a fájl argumentumok */
21      {
22        FILE *fp = fopen( argv[i], "r");
23        if ( NULL != fp )  /* sikeres megnyitás */
24        {
25          hd( fp, stdout);
26          fclose( fp);  /* véges számú fájl lehet megnyitva */
27        }
28        else
29        {
30          fprintf( stderr, "Can't open %s\n", argv[i]);
31          err = 1;    /* legalább egy fájl megnyitási hiba */
32        }
33      }
34    }
35    return err;  /* 0 == ok, 1 == volt hiba */
36  }
37
38  void hd( FILE *in, FILE *out)  /* out paraméter későbbi */
39  {                              /* továbbfejlesztéshez   */
40    char ch;
41    unsigned char buffer[LINESIZE]; /* "%x" miatt unsigned */
42    int cnt = 0;
43    long addr = 0L;  /* a címet mi fájlon belül számoljuk */
44
```

```
45    while ( EOF != (ch = fgetc(in)) )
46    {
47      buffer[cnt++] = ch;
48      if ( LINESIZE == cnt )  /* buffer teli */
49      {
50        print( out, addr, buffer, cnt); /* buffer kiírása */
51        addr += cnt;   /* == BUFSIZE, léptetjük a címet   */
52        cnt = 0;
53      }
54    }
55    /* ha a bufferben még maradt karakter, amikor a fájlunk
56       véget ér, akkor azokat még ki kell írnunk          */
57    if ( cnt > 0 )
58    {
59      print( out, addr, buffer, cnt);
60    }
61  }
62
63  void print(FILE *out,long addr,unsigned char *buf,int len)
64  {
65    int i;
66    fprintf( out, "%08lx  |  ", addr);  /* cím hexa-ban */
67    for ( i = 0; i < len; ++i )
68    {
69      fprintf( out, " %02x", buf[i]);  /* karakter hexa-ban */
70    }
71    for ( ; i < LINESIZE; ++i)  /* utolsó sor végi space-ek */
72    {
73      fprintf( out, "   ");
74    }
75    fprintf( out, "  |  ");
76    for ( i = 0; i < len; ++i)  /* maga a karakter vagy '.' */
77    {
78      fprintf( out, "%c", isgraph(buf[i]) ? buf[i] : '.');
79    }
80    fprintf( out, "\n");   /* a kiírt sor legvége */
81  }
```

Figyeljük meg a *buf* buffer **unsigned char** típusát. Erre azért van szükség, mert a **"%x"** format előjeltelen egész számot vár. A 128..255 közötti kódú **char** értékek bizonyos implementációkban lehetnek "negatívak", és ezek igen nagy pozitív **unsigned int** értékekre konvertálódnának, ami hatására a kiírt hexadecimális kód például valami ilyesmi lenne: **ffffff0a**. Az **unsigned char** értékek viszont garantáltan mindig a 0..255 közötti nemnegatív egész értékekre képződnek le.

Az eredeti UNIX utility úgy működik, hogy több input fájl esetén az outputot összefűzi és a címek folytonosan növekednek. A mi programunk nem fűzi össze az outputot, és a címeket is nulláról kezdi minden input esetében.

Házi feladat a program módosítása a UNIX utility viselkedése szerint.

A program implementációja Python3-ban

```
1   #!/usr/bin/env python3
```

```
2
3     from curses import ascii
4     import io
5     import sys
6
7     def hd(input, output):
8
9        addr = 0
10       line_size = 16
11
12       buffer = input.read(line_size)
13       while buffer != b"":
14          printer(output, addr, buffer, line_size)
15          addr+= line_size
16          buffer = input.read(line_size)
17
18    def printer(output, addr, buf, line_size):
19
20       line_in_hex = " ".join("{:02x}".format(c) for c in buf) \
21           .ljust(line_size*2+line_size)
22       replace_non_ascii = \
23                lambda a: chr(a) if ascii.isgraph(a) else "."
24       line_in_text = "".join(replace_non_ascii(c) for c in buf)
25       formatted_line = "{:08x}  |  {} |  {}". \
26                   format(addr, line_in_hex, line_in_text)
27       print(formatted_line, file=output)
28
29    def main():
30       if len(sys.argv) < 2:
31          hd(sys.stdin.buffer, sys.stdout)
32       else:
33          for arg in sys.argv[1:]:
34             try:
35                with io.open(arg,'rb') as file_content:
36                   hd(file_content, sys.stdout)
37             except IOError as ioerr:
38                   print("Can't open " + arg)
39
40    if __name__ == "__main__":
41       main()
```

Homepage of Dr. Zoltán Porkoláb

Home

Archive

Teaching

Timetable

Bolyai College

C++ (for mathematicians)

Imperative programming (BSc)

Multiparadigm programming (MSc)

Programming (MSc Aut. Sys.)

Programming languages (PhD)

Software technology lab

Theses proposals (BSc and MSc)

Research

CodeChecker

CodeCompass

Templight

Projects

Conferences

Publications

PhD students

Affiliations

Dept. of Programming Languages and Compilers

Ericsson Hungary Ltd

Imperatív programozás 12.

A string- és filekezelő könyvtár használata

Ez alkalommal egy telefonkönyv programot írunk be. Segítségével bemutatjuk a stringkezelés és az alapvető fájlkezelés függvényeit.

Ezt az alkalmazást ad-hoc írtuk meg a hallgatók segítségvel. Lejjebb a 2019-es megvalósítást lehet megtalálni, ami eléggé eltér a 2020-astól.

Implementáció diamikusan növekvő memóriával (2020)

A 2020-as implementáció egy dinamikusan allokált tömböt tartalmaz, ahol elhelyezzük az egyes bejegyzéseket. A bejegyzések **(név,telefonszám)** pointer párok, magukat a neveket és telefonszámokat dinamikusan allokáljuk, a phonebook tömbben csak a pointer-párok vannak. A bejegyzéseket a nevek szerint növekvő sorrendben tároljuk el.

Beszúráskor megkeressük azt a helyet, ahova az ábécé sorrend szerint el kell helyeznünk az új bejegyzést, a hátrább levő bejegyzéseket egy hellyel hátrébb csúsztatva helyet csinálunk az új bejegyzésnek, és a keletkezett helyre rakjuk az új elemet. Szükség esetén megnöveljük a dinamikus tömböt.

Törlésnél a törlendő elem által mutatott név és telefonszám területeket felszabadítjuk és a mögöttes elemeket feltoljuk a keletkezett "lyuk" helyére.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
5   #define BUFSIZE 1024   /* a maximális parancssor hossza*/
6   #define DEFAULT_NAME "phonebook.txt" /* default fájlnév*/
7
8   void add( char *name, char *phone);
9   void del( char *name);
10  void print( char *name);
11  void help(void);
12  void clean(void);
13  void quit(void);
14  void list(void);
15  void save(void);
16  void load(void);
17  void saveAs( char *fname);
18  void loadFrom( char *fname);
19
20  typedef void (*command_t)(); /* fenti függvények típusa */
21
22  struct COMMANDS   /* egy parancsot reprezentáló hármas: */
23  {
24    char      *name;  /* a parancs neve                    */
25    command_t  func;  /* végrehajtandó akció: fv. pointer */
26    int        npar;  /* a parancs paramétereinek száma   */
27  };
```

```
28
29  static struct COMMANDS commands[] = {   /* parancsok */
30        { "add",      add,      2 },
31        { "del",      del,      1 },
32        { "print",    print,    1 },
33        { "list",     list,     0 },
34        { "save",     save,     0 },
35        { "saveAs",   saveAs,   1 },
36        { "load",     load,     0 },
37        { "loadFrom", loadFrom, 1 },
38        { "clean",    clean,    0 },
39        { "help",     help,     0 },
40        { "quit",     quit,     0 }
41  };
42
43  typedef struct ENTRY  /* egy bejegyzés a telefonkönyvben */
44  {
45    char *name;   /* pointer a dinamikusan allokált névre  */
46    char *number; /* pointer a dinamikusan allokált számra */
47  }
48  entry_t;
49
50  int capacity = 0;  /* allokált memória mérete (elemszám) */
51  int size = 0;  /* ennyi elem van ténylegesen (elemszám)  */
52  entry_t *phonebook = NULL;  /* ptr a lefoglalt memóriára */
53
54  /* a szokásos módszer a tömb elemeinek meghatározására */
55  static int nComm = sizeof(commands)/sizeof(commands[0]);
56
57  /* segédfüggvények */
58  void grow();           /* megnöveli a lefoglalt memóriát */
59  int findCommand(char *buffer);
60  int findParams(char *buffer, int ci, char **p1, char **p2);
61  entry_t *find( char *name);
62
63  int main()
64  {
65    char buffer[BUFSIZE];
66    char *param1;
67    char *param2;
68
69    grow();    /* kezdeti memóriafoglalás */
70    printf("PB> ");
71    while ( fgets( buffer, BUFSIZE, stdin) )   /* EOF-ig */
72    {
73      int ci = findCommand( buffer);
74      if ( -1 != ci && -1 != findParams( buffer, ci, &param1,
75  &param2) )
76      {
77        /* megtaláltuk a parancsot és paramétereit */
78        switch( commands[ci].npar )  /* ennyi paraméteres */
79        {
80        /* meghívjuk a parancsot a szükséges paraméterekkel
```

```
81    */
82          case 0: commands[ci].func();                break;
83          case 1: commands[ci].func(param1);          break;
84          case 2: commands[ci].func(param1, param2);  break;
85          default: fprintf( stderr, "more parameters are "
86                           "not implemented yet\n");  break;
87          }
88        }
89        printf("PB> ");
90      }
91      return 0;
92    }
93
94    /* ha capacity == 0, akkor lefoglalja a kezdeti hosszt,
95       egyébként kétszeresére növeli a lefoglalt méretet   */
96    void grow(void)
97    {
98      static const int startcap = 4;  /* kezdeti hossz */
99      if ( 0 == capacity )  /* első alkalom */
100     {
101       phonebook = (entry_t *)
102                           malloc(startcap*sizeof(entry_t));
103       capacity = startcap;
104     }
105     else /* lefoglalt memória kétszeresére növelése */
106     {
107       phonebook = (entry_t *)
108              realloc(phonebook, 2*capacity*sizeof(entry_t));
109       capacity *= 2;
110     }
111     if ( NULL == phonebook ) /* nincsen memória */
112     {
113       fprintf( stderr, "Out of memory\n");
114       exit(1);
115     }
116   }
117
118   /* megkeresi a parancsot a felhasználói inputban és
119      - visszatér a commands tömbbeli indexével
120      - vagy -1-el, ha nincsen ilyen parancs            */
121   int findCommand( char *buffer)
122   {
123     char command[BUFSIZE];
124
125     if  (1 == sscanf(buffer,"%s",command)) /* az első szó */
126     {
127       int i;
128       for ( i = 0; i < nComm; ++i)  /* keressük a tömbben */
129       {
130         if ( 0 == strcmp( commands[i].name, command ) )
131         {
132           return i;  /* megtaláltuk az i-edik indexen */
133         }
```

```
134       }
135       /* nem találtuk meg */
136       printf( "%s: unknown command\n", command);
137     }
138     return -1; /* nem volt parancs vagy ismeretlen */
139   }
140
141   /* megkeresi a ci-ik indexhez tartozó parancs paramétereit,
142      ha megtalálja, akkor ráállítja a p1 ill. p2 pointereket,
143          és visszaadja a paraméterek számát,
144      ha nem találja meg, akkor -1-el tér vissza           */
145   int findParams( char *buffer, int ci, char **p1, char **p2)
146   {
147     int req = commands[ci].npar;  /* ennyi paraméter kell */
148     strtok( buffer, " \t\n");     /* átlépjük a parancsot */
149
150     if ( 0 == req ) /* 0 paraméteres parancs               */
151       return 0;          /* készen vagyunk */
152     else if ((*p1 = strtok( NULL, " \t\n"))) /* szóhatárig */
153     {
154      /* sikerül megtalálni az első paramétert, p1 mutat rá */
155       if ( 1 == req )  /* 1 paraméteres parancs            */
156       {
157         return 1;       /* készen vagyunk */
158       }
159       /* további paraméter kell */
160       if ((*p2 = strtok( NULL, "\t\n"))) /* ha nincsen ' ' */
161       {                  /* az elválasztók között, akkor a  */
162         return 2;        /* telefonszámon belül megengedett */
163       }                  /* a space karakter használata     */
164     }
165     /* nem találtuk meg a szükséges paramétereket, hibaüz. */
166     fprintf( stderr, "usage: %s %s %s\n", commands[ci].name,
167          req > 0 ? "name" : "",   /* van első paraméter    */
168          req > 1 ? "phone" : ""); /* van második paraméter */
169     return -1;
170   }
171
172   /* megkeresi a name első előfordulását a telefonkönyvben,
173      ha nem talál ilyet, akkor NULL-al tér vissza          */
174   entry_t *find(char *name)
175   {
176     int i;
177     for ( i = 0; i < size; ++i)
178     {
179       if ( 0 == strcmp( phonebook[i].name, name) )
180         return phonebook+i; /* pointer a bejegyzésre */
181     }
182     return NULL;
183   }
184
185   /* beszúr egy új elemet a név ábécé sorrend szerinti helyé-
186      re, a mögöttes elemeket hátramozgatja egy bejegyzéssel,
```

```
187     ha szükséges, megnöveli a dinamikus memória méretét   */
188   void add(char *name, char *number)
189   {
190     int remaining;
191     entry_t newEntry;
192
193     int i = 0;
194     /* keressük azt a pozíciót, amely először nagyobb a
195        beszúrandó névnél, ha nincs ilyen, akkor a
196        dinamikus tömb végére pozícionálunk */
197     while (i < size && 0 >= strcmp(phonebook[i].name,name))
198       ++i;
199
200     remaining = size - i;   /* ennyi elem van még hátrébb */
201     /* dinamikus helyet foglalunk a névnek és tel. számnak */
202     newEntry.name   = (char *) malloc( strlen(name)+1 );
203     newEntry.number = (char *) malloc( strlen(number)+1 );
204     if ( NULL == newEntry.name  ||  NULL == newEntry.number )
205     {
206       fprintf( stderr, "Out of memory\n");
207       exit(1);
208     }
209     strcpy( newEntry.name, name);   /* eltároljuk a nevet  */
210     strcpy( newEntry.number,number);/* eltároljuk a számot */
211
212     if ( size == capacity )  /* ha teli a dinamikus tömb */
213       grow();                /* akkor megnöveljük */
214
215     /* hogy helyet csináljunk az új elemnek hátrébb toljuk az
216        inzertálási pozíció után következő elemeket eggyel  */
217     memmove( phonebook+i+1, phonebook+i,
218           /* bájtokban számol */ remaining*sizeof(entry_t));
219     ++size;              /* egy elemmel több van eltárolva */
220     phonebook[i] = newEntry; /* elemet ténylegesen beírjuk */
221   }
222
223   /* ha megtaláljuk az elemet, akkor töröljük a dinamikus
224      memóriákat, ahol a név, és a telefonszám tárolva volt,
225      és feltömörítjük a bejegyzéseket, azaz felcsúsztatjuk
226      a mögöttes elemeket a keletkezett lyuk eltüntetésére */
227   void del( char *name)
228   {
229     entry_t *ptr = find(name); /* pointer a bejegyzésre */
230     if ( NULL == ptr ) /* nincsen ilyen név eltárolva */
231     {
232       printf( "Del: %s not found\n", name);
233     }
234     else /* megtaláltuk */
235     {
236       /* ennyi elem van a megtalált pozíció után */
237       int remaining = (phonebook+size) - ptr;
238       /* felszabadítjuk a dinamikus memóriát */
239       free(ptr->name);
```

```
240       free(ptr->number);
241       /* feltömörítjük a maradék elemeket */
242       memmove( ptr, ptr+1, remaining*sizeof(entry_t));
243       --size;  /* csökkent az elemek száma */
244     }
245   }
246
247   void print( char *name)
248   {
249     entry_t *ptr = find(name);
250     if ( NULL == ptr )
251     {
252       printf( "Print: %s not found\n", name);
253     }
254     else
255     {
256       printf( "%s: %s\n", ptr->name, ptr->number);
257     }
258   }
259
260   void list(void)
261   {
262     int i;
263     printf("=======================\n");
264     for ( i = 0; i < size; ++i)
265     {
266       printf( "%s: %s\n", phonebook[i].name,
267                           phonebook[i].number);
268     }
269     printf("=======================\n");
270   }
271
272   /* töröljük az összes elemet */
273   void clean()
274   {
275     int i;
276     for ( i = 0; i < size; ++i)
277     {
278       free(phonebook[i].name);
279       free(phonebook[i].number);
280     }
281     free(phonebook);
282     capacity = 0;
283     size = 0;
284   }
285
286   void save(void)
287   {
288     return saveAs( DEFAULT_NAME);
289   }
290
291   void load(void)
292   {
```

```
293     return loadFrom( DEFAULT_NAME);
294   }
295
296   void saveAs( char *name)
297   {
298     FILE *fp = fopen( name, "w");
299     if ( NULL == fp )
300     {
301       fprintf( stderr, "Can't open %s for write\n", name);
302       return;
303     }
304     else
305     {
306       int i;
307       for ( i = 0; i < size; ++i )
308       {
309         fprintf( fp, "%s:%s\n", phonebook[i].name,
310                                 phonebook[i].number);
311       }
312       fclose(fp);
313     }
314   }
315
316   /* töröljük a kurrens elemeket, beolvasunk egy fájlt és
317      hozzáadjuk ez elemeit az üres telefonkönyvhöz */
318   void loadFrom( char *name)
319   {
320     FILE *fp = fopen( name, "r");
321     if ( NULL == fp )
322     {
323       fprintf( stderr, "Can't open %s for read\n", name);
324       return;
325     }
326     else
327     {
328       char buffer[BUFSIZE];
329       clean(); /* töröljük a meglevő elemeket */
330       while ( fgets(buffer, BUFSIZE, fp) ) /* a fájl sorai */
331       {
332         /* kéne legyen benne egy ':' karakter */
333         char *ptr = strchr(buffer, ':');
334         if ( ptr )  /* megtaláltuk */
335         {
336           char *name   = buffer;  /* a név a ':' előtt  */
337           *ptr = '\0';            /* terminálja a nevet */
338           char *number = ptr+1;   /* a szám a ':' után  */
339           ptr = strchr( number, '\n');
340           if ( ptr )
341             *ptr = '\0'; /* felülírjuk a sorvégi '\n'-t */
342           printf( "add %s %s\n", name, number);
343           add( name, number);  /* hozzáadjuk a könyvhöz */
344         }
345         else
```

```
346         {
347           fprintf( fp, "Bad file format %s\n", buffer);
348           /* break; nem akarunk továbbmenni hiba esetén */
349         }
350       }
351       fclose(fp);
352     }
353   }
354
355   void help()
356   {
357     int i;
358     printf("commands:\n");
359     for ( i = 0; i < nComm; ++i)
360     {
361       printf( "%s %s %s\n",
362                   commands[i].name,
363                   commands[i].npar > 0 ? "name" : "",
364                   commands[i].npar > 1 ? "phone" : "");
365     }
366   }
367
368   void quit(void)
369   {
370     clean(); /* felszámolja a lefoglalt memóriát
371                 csak, hogy elemző szoftverek, mint
372                 valgrind ne adjanak hibajelzést */
       exit(0);
     }
```

Implementáció láncolt listával (2019)

```
1     #include <stdio.h>
2     #include <stdlib.h>
3     #include <string.h>
4
5     #define INPUTSIZE  80 /* max input line */
6     #define DEFAULT_NAME "phonebook.txt" /* default filenév */
7
8     /* lista elemek a telefonkönyv bejegyzésekre */
9     typedef struct ELEM
10    {
11      char        *name;
12      char        *phone;
13      struct ELEM *next;
14      struct ELEM *prev;
15    }
16    elem_t;
17
18    /* maga a telefonkönyv típusa */
19    typedef struct LIST
20    {
```

```
21      elem_t *first;
22      elem_t *last;
23      int     size;
24    }
25    list_t;
26
27    /* a telefonkönyv */
28    list_t phoneBook = { NULL, NULL, 0};
29
30    /* az egyes parancsok fv-re mutató pointer típusa */
31    typedef int (*comfunc_t)();
32
33    /* az egyes parancsok */
34    typedef struct COMMAND
35    {
36      char     *comStr;  /* neve */
37      int       nPars;   /* paraméterszáma */
38      comfunc_t comFunc; /* a fv pointer */
39    }
40    command_t;
41
42    /* a parancsok deklarációja */
43    int help(void);
44    int quit(void);
45    int list(void);
46    int add( char *name, char *phone);
47    int del( char *name);
48    int print( char *name);
49    int save(void);
50    int saveas( char *fname);
51    int load(void);
52    int loadfrom( char *fname);
53
54    /* segédfüggvények */
55    command_t *findCommand( char *buffer);
56    int findParams( command_t *comPtr, char **pars);
57    int insert( elem_t *before, char *name, char *phone);
58    void erase( elem_t *ptr);
59
60    /* a parancsok */
61    static command_t commands[] = {
62      { "help",  0, help  },
63      { "quit",  0, quit  },
64      { "list",  0, list  },
65      { "add",   2, add   },
66      { "del",   1, del   },
67      { "print", 1, print },
68      { "save",  0, save  },
69      { "saveas",1, saveas},
70      { "load",  0, load  },
71      { "loadfrom", 1, loadfrom }
72    };
73
```

```
74  /* a parancsok száma */
75  const int nCommands = sizeof(commands)/sizeof(commands[0]);
76
77  int main()
78  {
79    char buffer[INPUTSIZE];
80
81    printf("PB> ");
82    while ( NULL != fgets( buffer, INPUTSIZE, stdin) )
83    {
84      command_t *comPtr;
85      if ( (comPtr = findCommand( buffer)) )
86      {
87        char *pars[2] = { NULL, NULL};
88        if( findParams( comPtr, pars) )
89        {
90          switch(comPtr->nPars)
91          {
92          case 0: comPtr->comFunc();                  break;
93          case 1: comPtr->comFunc(pars[0]);           break;
94          case 2: comPtr->comFunc(pars[0], pars[1]);  break;
95          default: /* more par not implemented yet */ break;
96          }
97        }
98      }
99      printf("PB> ");
100   }
101   return 0;
102 }
103
104 command_t *findCommand( char *buffer)
105 {
106   int   i;
107   char *beg = strtok( buffer, " \t\n");
108
109   if ( NULL == beg )
110      return 0;
111
112   for ( i = 0; i < nCommands; ++i)
113   {
114     if ( 0 == strcmp( commands[i].comStr, beg) )
115     {
116       return &commands[i];
117     }
118   }
119   fprintf( stderr, "Unknown command: %s\n", buffer);
120   return 0;
121 }
122
123 int findParams( command_t *comPtr, char **params)
124 {
125   char  *par1, *par2;
126   if ( 0 == comPtr->nPars )
```

```
127     {
128       return 1;
129     }
130     else if ( NULL == (par1 = strtok( NULL, " \t\n")) )
131     {
132       fprintf( stderr, "Too few parameters: %s NAME\n",
133                comPtr->comStr);
134       return 0;
135     }
136     else if ( 1 == comPtr->nPars )
137     {
138       params[0] = par1;
139       return 1;
140     }
141     else if ( NULL == (par2 = strtok( NULL, "\t\n")) )
142     {
143       /* space not here, so phone number can include space */
144       fprintf( stderr, "Too few parameters: %s NAME PHONE\n",
145                comPtr->comStr);
146       return 0;
147     }
148     else if ( 2 == comPtr->nPars )
149     {
150       params[0] = par1;
151       params[1] = par2;
152       return 1;
153     }
154     else
155     {
156       /* more parameters are not implemented yet */
157       return 0;
158     }
159   }
160
161   int help(void)
162   {
163     int i = 0;
164     printf( "Usage: command par1 par2\n  commands: ");
165     for ( i = 0; i < nCommands; ++i)
166     {
167       printf( "%s, ", commands[i].comStr);
168     }
169     printf("\n");
170     return 1;
171   }
172
173   int quit(void)
174   {
175     exit(0);
176   }
177
178   int list(void)
179   {
```

```
180     elem_t *ptr = phoneBook.first;
181     printf("======= entries =======\n");
182     while ( ptr )
183     {
184       printf( "%s:\t%s\n", ptr->name, ptr->phone);
185       ptr = ptr->next;
186     }
187     printf("======================\n");
188     return 1;
189   }
190
191   int saveas(char *fname)
192   {
193     elem_t *ptr;
194     FILE *fp = fopen( fname, "w");
195     if ( NULL == fp )
196     {
197       fprintf( stderr, "Can open %s for write\n", fname);
198       return 0;
199     }
200     ptr = phoneBook.first;
201     while ( ptr )
202     {
203       fprintf( fp, "%s:%s\n", ptr->name, ptr->phone);
204       ptr = ptr->next;
205     }
206     fclose(fp);
207     return 1;
208   }
209
210   int save(void)
211   {
212     return saveas( DEFAULT_NAME);
213   }
214
215   int load(void)
216   {
217     return loadfrom( DEFAULT_NAME);
218   }
219
220   int loadfrom( char *fname)
221   {
222     char *name;
223     char *phone;
224     char  buffer[INPUTSIZE];
225     FILE *fp = fopen( fname, "r");
226
227     if ( NULL == fp )
228     {
229       fprintf( stderr, "Can open %s for read\n", fname);
230       return 0;
231     }
232     while ( NULL != fgets( buffer, INPUTSIZE, fp) )
```

```
233     {
234       if ( NULL == strchr( buffer, ':') )
235       {
236         fprintf( stderr, "Bad file format: %s\n", buffer);
237         return 1;
238       }
239       name = strtok( buffer, ":");
240       phone = strtok( NULL, "\n");
241       add( name, phone);
242     }
243     return 1;
244   }
245
246   int add(char *name, char*phone)
247   {
248     elem_t *ptr = phoneBook.first;
249
250     while ( ptr )
251     {
252       if ( 0 == strcmp( ptr->name, name) )
253       {
254         printf( "add: %s already in phonebook\n", name);
255         return 0;
256       }
257       else if ( 0 > strcmp( ptr->name, name) )
258       {
259         ptr = ptr->next;
260       }
261       else
262       {
263         /* insert before */
264         return insert( ptr->prev, name, phone);
265       }
266     }
267     /* insert last */
268     return insert( phoneBook.last, name, phone);
269   }
270
271   int insert( elem_t *before, char *name, char *phone)
272   {
273     elem_t *newPtr = (elem_t*) malloc(sizeof(elem_t));
274     if ( NULL == newPtr )
275     {
276       fprintf(stderr, "add: no memory\n");
277       return 0;
278     }
279     if ( 0 == phoneBook.size )
280     {
281       newPtr->next = newPtr->prev = NULL;
282       phoneBook.first = phoneBook.last = newPtr;
283     }
284     else
285     {
```

```
286       newPtr->prev = before;
287       newPtr->next = before ? before->next : phoneBook.first;
288       if ( newPtr->prev ) newPtr->prev->next = newPtr;
289       else phoneBook.first = newPtr;
290       if ( newPtr->next ) newPtr->next->prev = newPtr;
291       else phoneBook.last = newPtr;
292     }
293     ++phoneBook.size;
294     newPtr->name  = (char *) malloc( strlen(name)+1);
295     newPtr->phone = (char *) malloc( strlen(phone)+1);
296     strcpy( newPtr->name,  name);
297     strcpy( newPtr->phone, phone);
298     return 1;
299   }
300
301   void erase( elem_t *ptr)
302   {
303     if ( ptr->prev )
304       ptr->prev->next = ptr->next;
305     if ( ptr->next )
306       ptr->next->prev = ptr->prev;
307     if ( phoneBook.first == ptr )
308       phoneBook.first = ptr->next;
309     if ( phoneBook.last  == ptr )
310       phoneBook.last  = ptr->prev;
311     --phoneBook.size;
312     free(ptr);
313   }
314
315   int del(char *name)
316   {
317     elem_t *ptr = phoneBook.first;
318
319     while ( ptr )
320     {
321       if ( 0 == strcmp( ptr->name, name) )
322       {
323         free(ptr->name);
324         free(ptr->phone);
325         erase(ptr);
326         return 1;
327       }
328       ptr = ptr->next;
329     }
330     printf("del: %s not found\n", name);
331     return 1;
332   }
333
334   int print(char *name)
335   {
336     elem_t *ptr = phoneBook.first;
337
338     while ( ptr )
```

```
339     {
340       if ( 0 == strcmp( ptr->name, name) )
341       {
342         printf( "%s:\t%s\n", ptr->name, ptr->phone);
343         return 1;
344       }
345       ptr = ptr->next;
346     }
347     printf("print: %s not found\n", name);
348     return 0;
349   }
```