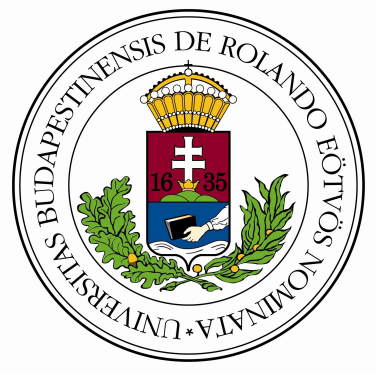
Imperatív programozás

Imperatív programozás



Kozsik Tamás és mások

ELTE Eötvös Loránd Tudományegyetem

Outline

A tárgy célja

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

Használt programozási nyelv: C

(Miért is tanulunk C-t? link!)

A képzés formája

- Előadás
- Gyakorlat
- Konzultáció

Számonkérés

- Rendszeresen: tesztek, feladatok
- Félév végén: vizsgazárthelyi

Az elvárt munka

Összesen 150 munkaóra

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

További információk

A tárgy honlapján:

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

A canvasben

`http://canvas.elte.hu/`

Outline

Programozási nyelvek

- Ember-gép kommunikáció
- Ember-ember kommunikáció

Programozási paradigmák

Gondolkodási sémák, szükséges nyelvi eszközök

Például:

- Imperatív programozás
- Funkcionális programozás
- Logikai programozás

- Procedurális programozás
- Moduláris programozás
- Objektumelvű programozás

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

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

Gépi kód


Unsorted array is: %d
Sorted array is:
GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3
.symtab .strtab .shstrtab .rel.text .data .bss .rodata .comment .note.GNU-stack .rel.eh_frame
quickSort.c main printf quickSort partition

Assembly

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

„Magas szintű” programozási nyelvek

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

stb.

Modern, kényelmes nyelvek

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

stb.

Ada Lovelace (Analytical Engine, Charles Babbage)

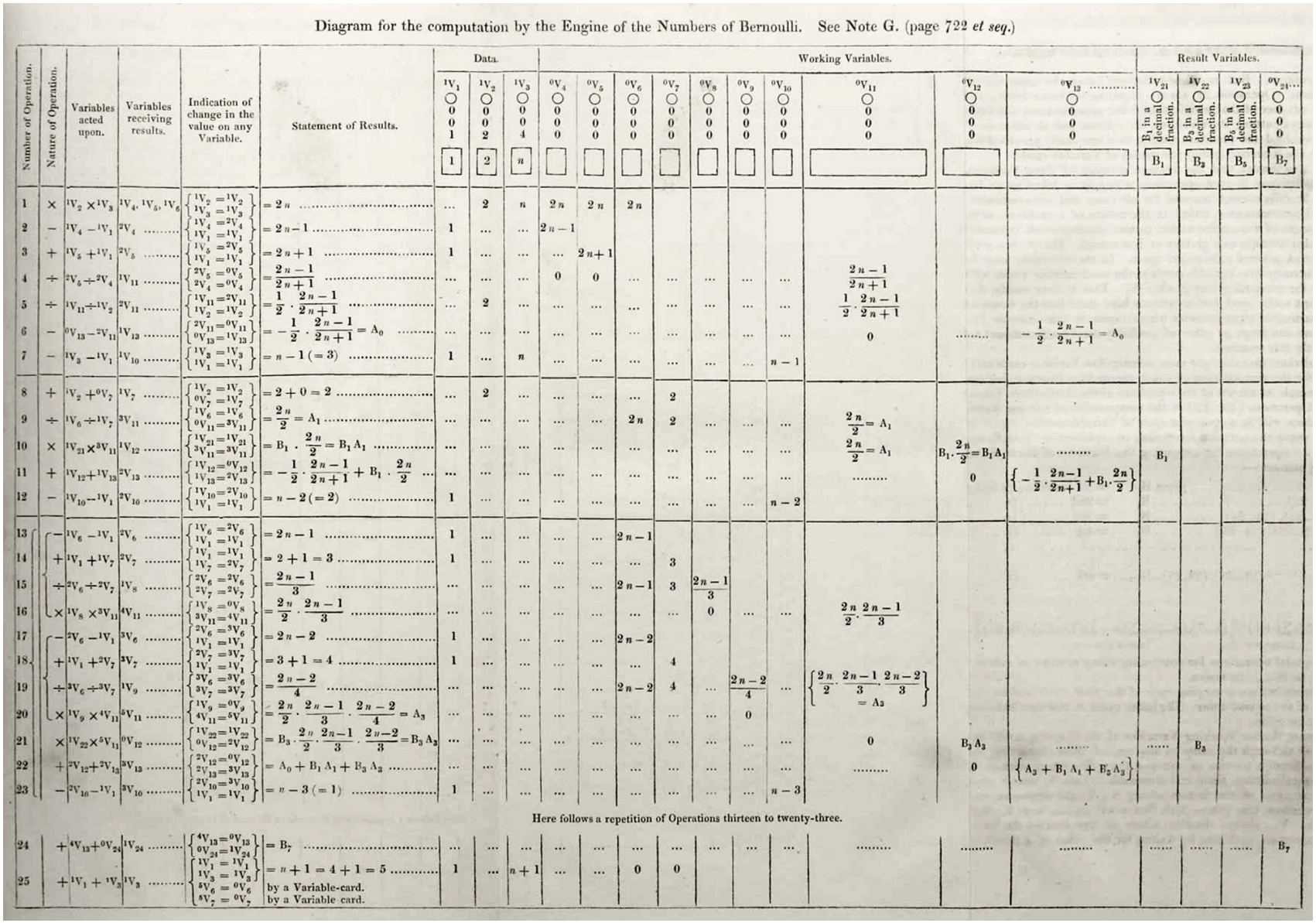


Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 *et seq.*)

Number of Operation.	Nature of Operation.	Variables acted upon.	Variables receiving results.	Indication of change in the value on any Variable.	Statement of Results.	Data. ${}^1V_1$	Data. ${}^1V_2$	Data. ${}^1V_3$	Working Variables. ${}^0V_4$	${}^0V_5$	${}^0V_6$	${}^0V_7$	${}^0V_8$	${}^0V_9$	${}^0V_{10}$	${}^0V_{11}$	${}^0V_{12}$	${}^0V_{13}$	Result Variables. ${}^1V_{21}$ B_1	${}^1V_{22}$ B_3	${}^1V_{23}$ B_5	${}^0V_{24}$ ... B_7
1	×	${}^1V_2 \times {}^1V_3$	${}^1V_4, {}^1V_5, {}^1V_6$	${}^1V_2 = {}^1V_2$, ${}^1V_3 = {}^1V_3$	$= 2n$	...	2	n	$2n$	$2n$	$2n$											
2	−	${}^1V_4 - {}^1V_1$	${}^2V_4$	${}^1V_4 = {}^2V_4$, ${}^1V_1 = {}^1V_1$	$= 2n-1$	1	...	...	$2n-1$													
3	+	${}^1V_5 + {}^1V_1$	${}^2V_5$	${}^1V_5 = {}^2V_5$, ${}^1V_1 = {}^1V_1$	$= 2n+1$	1	...	...	...	$2n+1$												
4	÷	${}^2V_5 \div {}^2V_4$	${}^1V_{11}$	${}^2V_5 = {}^0V_5$, ${}^2V_4 = {}^0V_4$	$= \frac{2n-1}{2n+1}$	...	...	...	0	0	...	...	...	...	...	$\frac{2n-1}{2n+1}$						
5	÷	${}^1V_{11} \div {}^1V_2$	${}^2V_{11}$	${}^1V_{11} = {}^2V_{11}$, ${}^1V_2 = {}^1V_2$	$= \frac{1}{2} \cdot \frac{2n-1}{2n+1}$	...	2	...	...	...	...	...	...	...	...	$\frac{1}{2} \cdot \frac{2n-1}{2n+1}$						
6	−	${}^0V_{13} - {}^2V_{11}$	${}^1V_{13}$	${}^2V_{11} = {}^0V_{11}$, ${}^0V_{13} = {}^1V_{13}$	$= -\frac{1}{2} \cdot \frac{2n-1}{2n+1} = A_0$	...	...	...	...	...	...	...	...	...	...	0	.....	$-\frac{1}{2} \cdot \frac{2n-1}{2n+1} = A_0$				
7	−	${}^1V_3 - {}^1V_1$	${}^1V_{10}$	${}^1V_3 = {}^1V_3$, ${}^1V_1 = {}^1V_1$	$= n-1\ (=3)$	1	...	n	...	...	...	...	...	...	$n-1$							
8	+	${}^1V_2 + {}^0V_7$	${}^1V_7$	${}^1V_2 = {}^1V_2$, ${}^0V_7 = {}^1V_7$	$= 2 + 0 = 2$	...	2	...	...	...	...	2										
9	÷	${}^1V_6 \div {}^1V_7$	${}^3V_{11}$	${}^1V_6 = {}^1V_6$, ${}^0V_{11} = {}^3V_{11}$	$= \frac{2n}{2} = A_1$	...	...	...	...	...	$2n$	2	...	...	...	$\frac{2n}{2} = A_1$						
10	×	${}^1V_{21} \times {}^3V_{11}$	${}^1V_{12}$	${}^1V_{21} = {}^1V_{21}$, ${}^3V_{11} = {}^3V_{11}$	$= B_1 \cdot \frac{2n}{2} = B_1 A_1$	...	...	...	...	...	...	...	...	...	...	$\frac{2n}{2} = A_1$	$B_1 \cdot \frac{2n}{2} = B_1 A_1$	.....	B_1			
11	+	${}^1V_{12} + {}^1V_{13}$	${}^2V_{13}$	${}^1V_{12} = {}^0V_{12}$, ${}^1V_{13} = {}^2V_{13}$	$= -\frac{1}{2} \cdot \frac{2n-1}{2n+1} + B_1 \cdot \frac{2n}{2}$ ...	...	...	...	...	...	...	...	...	...	...	.....	0	$\left\{-\frac{1}{2} \cdot \frac{2n-1}{2n+1} + B_1 \cdot \frac{2n}{2}\right\}$				
12	−	${}^1V_{10} - {}^1V_1$	${}^2V_{10}$	${}^1V_{10} = {}^2V_{10}$, ${}^1V_1 = {}^1V_1$	$= n-2\ (=2)$	1	...	...	...	...	...	...	...	...	$n-2$							
13	−	${}^1V_6 - {}^1V_1$	${}^2V_6$	${}^1V_6 = {}^2V_6$, ${}^1V_1 = {}^1V_1$	$= 2n-1$	1	...	...	...	...	$2n-1$											
14	+	${}^1V_1 + {}^1V_7$	${}^2V_7$	${}^1V_1 = {}^1V_1$, ${}^1V_7 = {}^2V_7$	$= 2 + 1 = 3$	1	...	...	...	...	...	3										
15	÷	${}^2V_6 \div {}^2V_7$	${}^1V_8$	${}^2V_6 = {}^2V_6$, ${}^2V_7 = {}^2V_7$	$= \frac{2n-1}{3}$	...	...	...	...	...	$2n-1$	3	$\frac{2n-1}{3}$									
16	×	${}^1V_8 \times {}^3V_{11}$	${}^4V_{11}$	${}^1V_8 = {}^0V_8$, ${}^3V_{11} = {}^4V_{11}$	$= \frac{2n}{2} \cdot \frac{2n-1}{3}$	...	...	...	...	...	...	...	0	...	...	$\frac{2n}{2} \cdot \frac{2n-1}{3}$						
17	−	${}^2V_6 - {}^1V_1$	${}^3V_6$	${}^2V_6 = {}^3V_6$, ${}^1V_1 = {}^1V_1$	$= 2n-2$	1	...	...	...	...	$2n-2$											
18	+	${}^1V_1 + {}^2V_7$	${}^3V_7$	${}^2V_7 = {}^3V_7$, ${}^1V_1 = {}^1V_1$	$= 3 + 1 = 4$	1	...	...	...	...	...	4										
19	÷	${}^3V_6 \div {}^3V_7$	${}^1V_9$	${}^3V_6 = {}^3V_6$, ${}^3V_7 = {}^3V_7$	$= \frac{2n-2}{4}$	...	...	...	...	...	$2n-2$	4	...	$\frac{2n-2}{4}$	...	$\left\{\frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{3} = A_2\right\}$							
20	×	${}^1V_9 \times {}^4V_{11}$	${}^5V_{11}$	${}^1V_9 = {}^0V_9$, ${}^4V_{11} = {}^5V_{11}$	$= \frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{4} = A_3$ ...	...	...	...	...	...	...	...	...	0	...							
21	×	${}^1V_{22} \times {}^5V_{11}$	${}^0V_{12}$	${}^1V_{22} = {}^1V_{22}$, ${}^0V_{12} = {}^2V_{12}$	$= B_3 \cdot \frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{3} = B_3 A_3$	...	...	...	...	...	...	...	...	...	...	0	$B_3 A_3$	.....	.....	B_3		
22	+	${}^2V_{12} + {}^2V_{13}$	${}^3V_{13}$	${}^2V_{12} = {}^0V_{12}$, ${}^2V_{13} = {}^3V_{13}$	$= A_0 + B_1 A_1 + B_3 A_3$	...	...	...	...	...	...	...	...	...	...	.....	0	$\left\{A_3 + B_1 A_1 + B_3 A_3\right\}$				
23	−	${}^2V_{10} - {}^1V_1$	${}^3V_{10}$	${}^2V_{10} = {}^3V_{10}$, ${}^1V_1 = {}^1V_1$	$= n-3\ (=1)$	1	...	...	...	...	...	...	...	...	$n-3$							

Here follows a repetition of Operations thirteen to twenty-three.

24	+	${}^4V_{13} + {}^0V_{24}$	${}^1V_{24}$	${}^4V_{13} = {}^0V_{13}$, ${}^0V_{24} = {}^1V_{24}$	$= B_7$	...	...	...	...	...	...	...	...	...	...	.....	.....	.....	.....	.....	.....	B_7
25	+	${}^1V_1 + {}^1V_3$	${}^1V_3$	${}^1V_1 = {}^1V_1$, ${}^1V_3 = {}^1V_3$, ${}^5V_6 = {}^0V_6$, ${}^5V_7 = {}^0V_7$	$= n+1 = 4+1 = 5$ ..... by a Variable-card. by a Variable-card.	1	...	$n+1$	...	...	0	0										

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

A programozás őskora

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

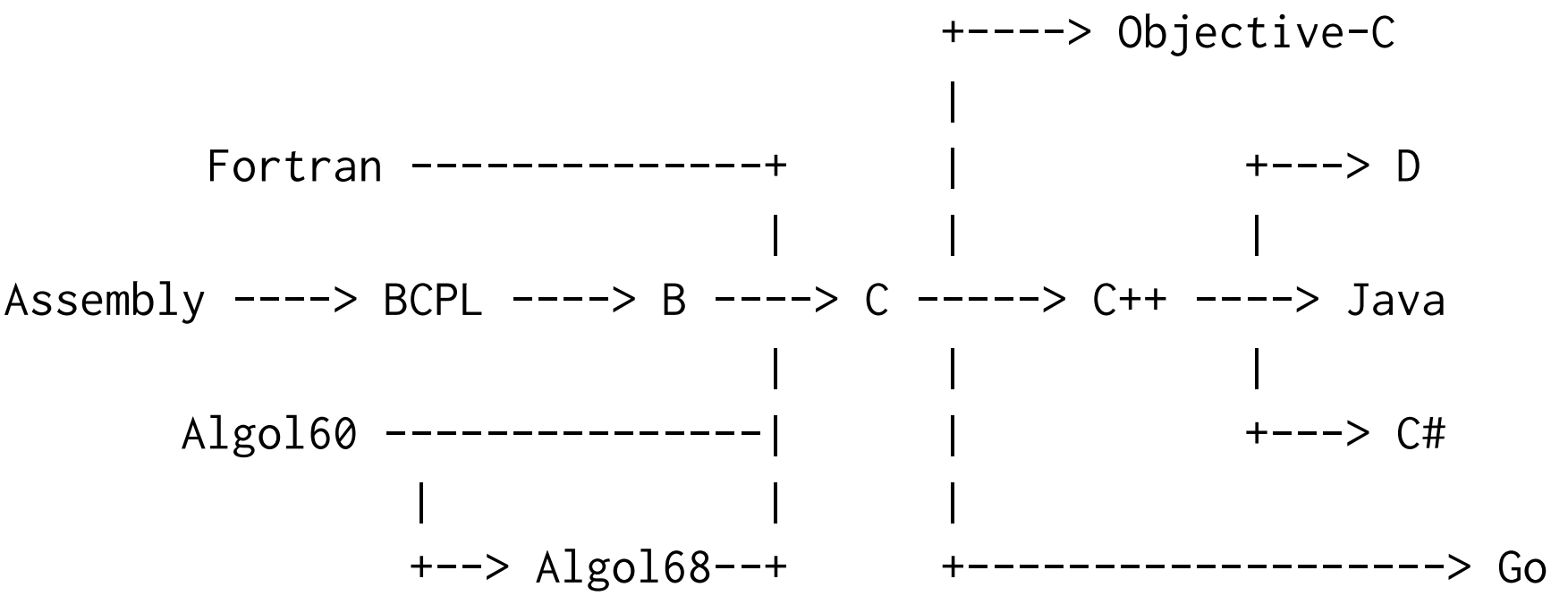
Néhány fontos nyelv

- Simula-67 (Dahl–Nygaard, 1967)
- Pascal (Niklaus Wirth, 1970)
- C (Dennis Ritchie, 1972)
- Ada (1980)
- SQL (Chamberlin–Boyce, 1974)
- C++ (Bjarne Stroustrup, 1985)
- Eiffel (Bertrand Meyer, 1986)
- Erlang (Armstrong–Virding–Williams, 1986)
- Haskell (1990)
- Python (Guido van Rossum, 1990)
- Java (James Gosling, 1995)
- JavaScript (Brendan Eich, 1995)
- PHP (Rasmus Lerdorf, 1995)
- C# (2000)
- Scala (Martin Odersky, 2004)

Legnépszerűbb nyelvek (2018. szeptember, TIOBE-index)

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

A C nyelv kialakulása



A C nyelv fejlődése

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

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

Outline

Programok felépítése

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

Példa

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

Kifejezések

```
n
```

```
"Hello world!"
```

```
100
```

```
n+1
```

```
++i
```

```
range(2,n+1)
```

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

Utasítások

```
result = 1;
```

```
        result *= i;
```

```
                                    return result;
```

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

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

Egyszerű utasítások

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

Vezérlési szerkezetek

- elágazások
- ciklusok stb.

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

Kapcsos zárójelek vezérlési szerkezetekben

Elhagyott kapcsos zárójelek

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

Bolondbiztos megoldás

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

Csellengő else (dangling else)

Ezt írtam

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

Ezt jelenti

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

Ezt akartam

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

Lásd még...

goto-fail (Apple) link!

Kiírás a szabványos kimenetre

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

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

Bonyolultabb kiírás

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

Típusok

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

C-ben

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

stb.

Típus szerepe

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

Típusellenőrzés

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

Statikus és dinamikus típusrendszer

A C fordító ellenőrzi *fordítási időben* a típushelyességet

Erősen és gyengén típusos nyelv

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

Alprogramok (subprograms)

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

Főprogram (main program)

Ahol a program végrehajtása elkezdődik

C

Egy megfelelő nevű alprogram: `main`

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

Megjegyzés

```c
int main()
{
    int half = 21;
    printf("%d\n",2*half);
    return 0;                    /* itt így írok megjegyzést */
}
```

Modul

Modularitás: egységbe zárás, függetlenség, szűk interfészek

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

Modulokra bontás

Újrafelhasználható `factorial`

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

tenfactorial.c

```c
#include <stdio.h>

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

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

Outline

Forráskód

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

Parancsértelmező (interpreter)

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

Hátrányok

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

Előnyök

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

Forrásfájl C-ben

factorial.c

```c
#include <stdio.h>

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

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

Fordítás és futtatás szétválasztása

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

“Fordítási idő” és “futási idő”

Fordítás

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

Fordítási egység

(compilation unit)

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

Egy program több fordítási egységből szokott állni!

C-ben

Egy forrásfájl tartalma

Szerkesztés, végrehajtható kód

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

Sok tárgykódból lesz egy végrehajtható kód!

Végrehajtás

```
./factorial
```

Több fordítási egység

factorial.c

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

tenfactorial.c

```
#include <stdio.h>

int factorial( int n );

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

Fordítás, szerkesztés, futtatás

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

A két lépés összevonható egy parancsba

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

Fordítási hibák

- Nyelv szabályainak megsértése
- Fordítóprogram detektálja

factorial.c

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

gcc -c factorial.c

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

Szerkesztési hibák

factorial.c

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

tenfactorial.c

```c
#include <stdio.h>

int faktorial( int n );

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

Fordítás, szerkesztés, hiba

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

Fordítási és futási idejű szerkesztés

Statikus szerkesztés

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

Dinamikus szerkesztés

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

Előfeldolgozás

C preprocessor: (forráskódból) forráskódot állít elő

Makrók

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

Deklarációk megosztása

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

Feltételes fordítás

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

Programok C-ben

Fordítási idő

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

Futási idő

- Futtatható állomány, tárgykódok
- Dinamikus szerkesztés
- Futó program

Outline

Programozási nyelv szabályai

- Lexikális
- Szintaktikus
- Szemantikus

Lexikális szabályok

Milyen építőkockái vannak a nyelvnek?

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

Case-(in)sensitive?

Literál: egész szám

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

Literál: lebegőpontos szám

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

Literál: karakter és sztring

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

```
"alma\
fa"
```

Azonosító

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

Jó

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

Rossz

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

Szintaktikus szabályok

Hogyan építkezhetünk?

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

Backus-Naur form (Backus normal form) – BNF

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

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

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

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

Szemantikus szabályok

Értelmes, amit építettünk?

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

stb.

A típus szerepe

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

Típusellenőrzés

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

Statikus és dinamikus típusrendszer

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

Erősen és gyengén típusos nyelv

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

Statikus és dinamikus szemantikai szabályok

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

Eldönthetőségi probléma...

Összefoglalva

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

A fordítóprogramok részei

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

(vagy különböző szintű fordítási hibák)

Formális nyelvek

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

Program szemantikája

A (nyelv szabályainak megfelelő) program jelentése

A nyelv definíciója

- Lexika
- Szintaktika
- Szemantika
- Pragmatika

Pragmatika

Hogyan tudjuk hatékonyan kifejezni magunkat?

- Konvenciók
- Idiómák
- Jó és rossz gyakorlatok

stb.

Konvenció

általános vagy fejlesztői csoportra (cégre) specifikus

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

Outline

Példák

```
n + 1
```

```
3 * v[0]
```

```
x < 3.14
```

```
3.14 * r * r
```

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

Lexika

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

Számok ábrázolása

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

(különböző méretekben)

Előjel nélküli egész számok

Négy biten

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

n biten

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

C-ben

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

Előjellel: „kettes komplemens” ábrázolás

(two's complement)

- első bit: előjel
- többi bit: helyiértékek

Négy biten

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

```
  0011
+1101
-----
10000
```

Előjeles egész C-ben

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

Aritmetika előjeles egészekre

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

Egész típusok mérete

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

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

Haskell

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

Lebegőpontos számok

$$1423.3 = 1.4233 \cdot 10^{3}$$

$$14.233 = 1.4233 \cdot 10^{1}$$

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

Bináris ábrázolás

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

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

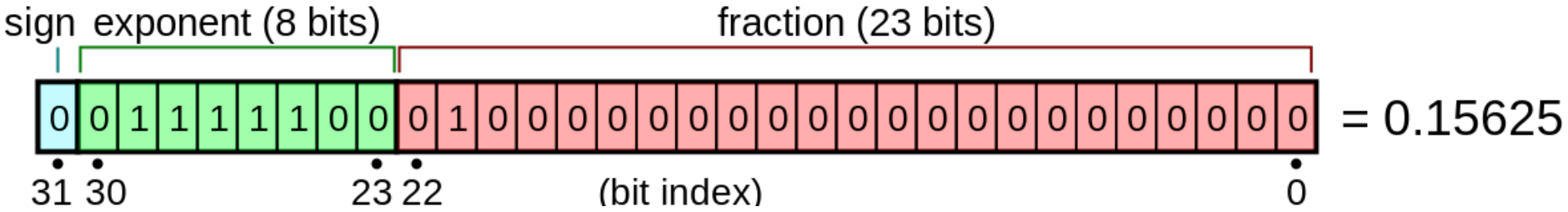
Rögzített számú biten reprezentálandó

- Előjel
- Kitevő
- Értékes számjegyek

IEEE 754

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

32 bites példa



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

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

Lebegőpontos számok tulajdonságai

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

Lebegőpontos aritmetika

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

Pénzt például sosem ábrázolunk lebegőpontos számokkal!

Komplex számok

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

C99

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

Komplex számok C99-től

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

Konvertálás típusok között

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

Operátorok

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

Aritmetikai operátorok

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

„Valós” és egész osztás

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

Osztási maradék

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

Eredmény előjele: `left` előjele

Hatványozás

```
#include <math.h>

    pow( 5.1, 2.1 )
```

Értékadás

Értékadó utasítás

```
n = 1;
```

Mellékhatásos kifejezés

```
n = 1
```

Mellékhatásos kifejezés értéke

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

Érték továbbgyűrűzése

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

Értékadó operátorok

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

Eggyel növelő/csökkentő operátorok

Mellékhatás

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

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

Érték

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

Relációs operátorok

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

Logikai (boolean) értékűek

Mit jelent ez?

$3 < x < 7$

Logikai típus?

ANSI C: nincsen

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

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

C99-től

```
_Bool v = 3 < 5;
int one = (_Bool) 0.5;
int zero = (int) 0.5;
```

```
#include <stdbool.h>
...
bool v = true;
```

Végtelen ciklus idiómája

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

Mit csinál ez a kód?

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

Logikai operátorok

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

Feltételes operátor

```
condition ? left : right
```

Bitműveletek

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

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

Függvényhívás szintaktikája

aktuális paraméterlista

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

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

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

Operátorok használata

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

Kifejezések kiértékelése

- teljesen bezárójelezett kifejezés

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

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

  `12 - 3 * 4`

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

Kifejezések kiértékelése (folyt.)

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

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

- operandusok, függvényparaméterek kiértékelési sorrendje

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

Szekvenciapont

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

Vessző-operátor

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

- Az eredménye a jobboldali kifejezés eredménye
- Alacsony precedenciaszintű

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

Vessző: operátor vagy elválasztójel

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

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

Értékek

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

Karakterek

Valójában egy egész szám!

- Egy bájtos karakterkód, pl. ASCII

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

Escape-szekvenciák

- Speciális karakter: `\n`, `\r`, `\f`, `\t`, `\v`, `\b`, `\a`, `\\`, `\'`, `\"`, `\?`
- Oktális kód: `\0` – `\377`
- Hexadecimális kód, pl. `\x41`

Előjeles és előjel nélküli char

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

Szélesebb ábrázolás

```
wchar_t w = L'é';
```

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

Szövegek

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

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

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

wchar_t wide[] = L"körte";
```

Ékezetes betűk a szövegben

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

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

Karaktertömb lefoglalása

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

Veszély: túl kis tömb foglalása

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

Szövegen belül nulla

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

Szövegek manipulálása

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

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

Kitekintés

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

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

C tömb

```
double point[3];      /* a méret legyen konstans */
point[0] = 3.14;      /* nullától indexelünk */
point[1] = 2.72;
point[2] = 1.0;
```

Tömb inicializációja

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


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

Feldolgozás

```
#define DIMENSION 3

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

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

Általánosítás

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

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

Veszélyforrások

Fordítási hiba

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

Inicializálatlan elemek

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

Túlindexelés, illegális memóriaolvasás

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

Túlindexelés, illegális memóriaírás (buffer overflow)

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

Szövegek megadása tömbként


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

Outline

Eddig megismert utasítások

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

Változódeklaráció

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

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

Kifejezés-utasítás

(Mellékhatásos) kifejezés kiértékelése

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

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

Tipikus példa: értékadások

Függvények

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

Tiszta függvény

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

Csak mellékhatás

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

Kevert viselkedés

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

Visszatérés

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

```
return 42;
return v + 3.14;

return;
```

Több return utasítás

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

Üres utasítás

```
;
```

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

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

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

Vezérlési szerkezetek

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

Strukturált programozás

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

Szekvencia

- Utasítások egymás után írásával
- Pontosvessző
- Blokk utasítás

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

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

Vezérlési szerkezetek belseje

- egy utasítás
- lehet a blokk-utasítás is

Elágazás

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

Többágú elágazás

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

Többágú elágazás konvencionális tördelése

Idióma

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

Konvencionális tördelés

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

A kapcsos zárójelek nem ártanak

Idióma

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

Konvencionális tördelés

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

switch-case-break utasítás

egész típusú, fordítási idejű konstansok alapján

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

Adatban kódolt vezérlés

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

Nem mindig kényelmes adatként

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

Átcsorgás

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

Nem triviális átcsorgás

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

A `switch` és a strukturált programozás

Strukturáltnak tekinthető

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

Nem felel meg a strukturált programozásnak

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

Elöltesztelő ciklus

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

Olvashatóság

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

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

while – szintaxis

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

Hátultesztelő ciklus

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

Jellemző példa

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

Átírás – 1

Milyen feltétel mellett igaz ez?

`do` σ `while ( ` ε `);` $\equiv$ σ `while ( ` ε `)` σ

Átírás – 2

Milyen feltétel mellett igaz ez?

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

Az előző példa átírva

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

Refaktorálva

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

Vége jelig való beolvasás idiómája

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

Léptető ciklus

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

(inicializáció; feltétel; léptetés)

Végtelen ciklus

```
while(1) ...

for(;;) ...
```

Karaktertábla készítése

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

Célszerű így fordítani

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

Átírások

Mindig megtehető

`while ( ` ε `) ` σ $\quad\Rightarrow\quad$ `for ( ; ` ε `; ) ` σ

Milyen feltétel mellett igaz ez?

`for ( ` ι `; ` ε `; ` λ `) ` σ $\quad\Rightarrow\quad$ ι`; while ( ` ε `){ ` σ λ`; }`

Egyszerű utasítások

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

Strukturált programozás vezérlési szerkezetei

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

Nem strukturált vezérlésátadás

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

break utasítás

- Kilép a legbelső ciklusból (vagy switch-ből)

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

continue utasítás

- Befejezi a legbelső ciklusmag végrehajtását

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

- for-ciklusnál végrehajtja a léptetést

goto utasítás

- Egy függvényen belül a megadott címkéjű utasításra ugrik

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

Keressünk nulla elemet egy mátrixban

goto-val

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

szabályosan

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

Rekurzív alprogramok

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

Másképpen fogalmazva

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

Számítási lépések ismétlése

Imperatív programozás

- Iteráció (ciklus)
- Hatékony

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

Funkcionális programozás

- Rekurzió
- Érthető

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

Rekurzió imperatív nyelvben

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

Van, amikor kényelmes

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

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

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

Végrekurzív függvény (tail-recursion)

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

Kézenfekvő

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

Végrekurzív

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

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

A fordítóprogram optimalizálhatja

Végrekurzív

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

Optimalizált

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

Strukturáltan

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

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

Outline

Programszerkezet

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

Mellérendelt szerkezetek

- Fordítási egységek
- Programkönyvtárak
- Újrafelhasználhatóság

Alá-/fölérendelt szerkezetek

- Egymásba ágyazódás
- Hierarchikus elrendezés
- Lokalitás: komplexitás csökkentése

Hierarchikus programfelépítés

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

Hierarchia nélkül

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

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

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

Függvények egymásba ágyazása, lokális fv-definíció?

Nem valid C-kód!

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

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

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

Függvények egymásba ágyazása tetszőleges mélységben?

Nem valid C-kód!

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

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

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

Deklaráció és definíció

Gyakran együtt, de lehet az egyik a másik nélkül!

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

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

Deklaráció hatóköre (scope)

Amíg a névvel elérhető az, amire hivatkozik

- Globális: legkívül van
- Lokális: valamin belül van

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

Globális és lokális függvény

Ha a C blokkszerkezetes lenne...

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

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

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

Blokk (block)

A (statikus) hatóköri szabályok alapja

- Alprogram
- Blokk utasítás

Törzs (body)

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

Statikus hatóköri szabályok (static/lexical scoping)

a deklarációtól a deklarációt közvetlenül tartalmazó blokk végéig

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

Globális – lokális deklaráció

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

Lokális, non-lokális, globális deklaráció

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

Lokális, non-lokális, globális változó

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

Globális deklarációk és definíciók

```c
int x;

extern int y;

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

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

Fordítás

Minden fordítási egységben minden használt név deklarált kell legyen

Szerkesztés

Az egész programban minden globális név pontosan egyszer legyen definiálva

Elfedés (shadowing/hiding)

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

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

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

Lokális változók deklarációja

ANSI C

- Blokk elején, egyéb utasítások előtt

C99-től

- Keverve a többi utasítással

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

- For-ciklus lokális változójaként

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

Nonlokális definíció elérése

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

Deklarációk sorrendje

Hibás!

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

Helyes

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

Összegzés

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

Ciklusváltozó

C99: ciklusra lokális változó

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

C: 012345678910

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

C: végtelen ciklus

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

Definíció deklaráció nélkül

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

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

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

Magasabbrendű függvények

Funkcionális programozási paradigma

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

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

Dinamikus hatóköri szabályok

Bash

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

f
echo $x
```

C

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

Outline

Dinamikus programszerkezet

Hogyan működik a program?

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

Adunk egy absztrakt modellt!

Végrehajtási verem

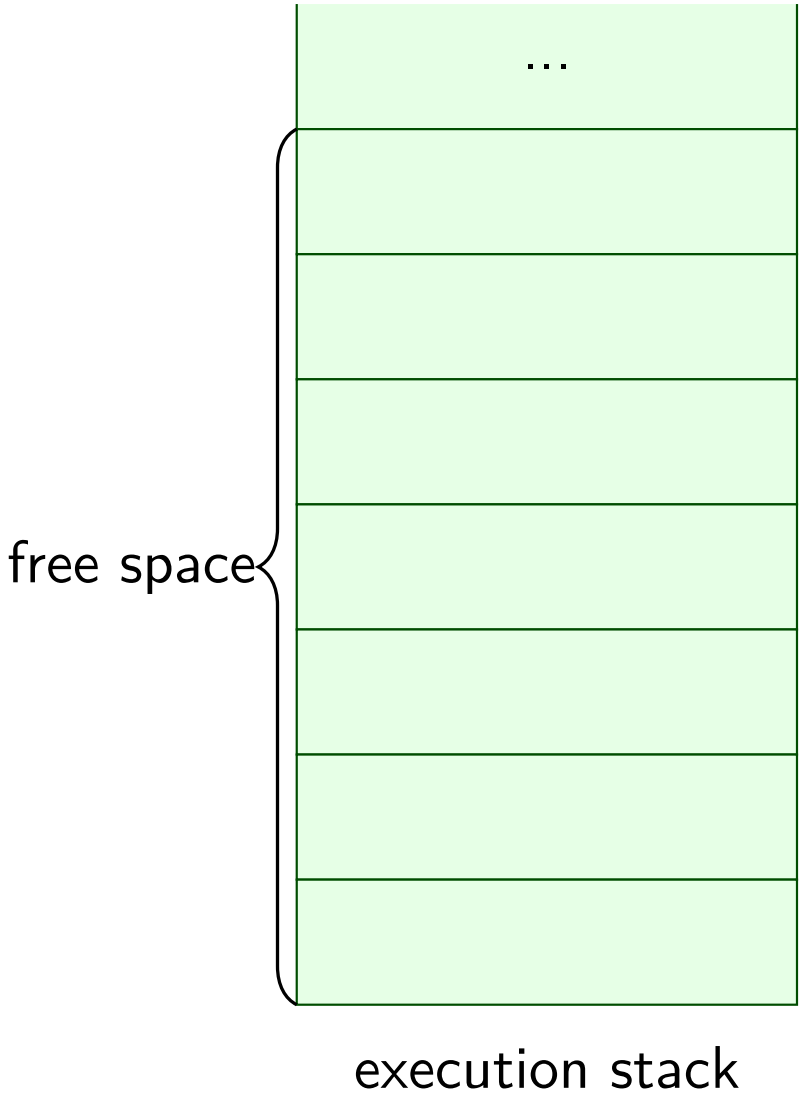
```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
}
int main()
{
    f();
    h();
    return 0;
}
```

Execution stack

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

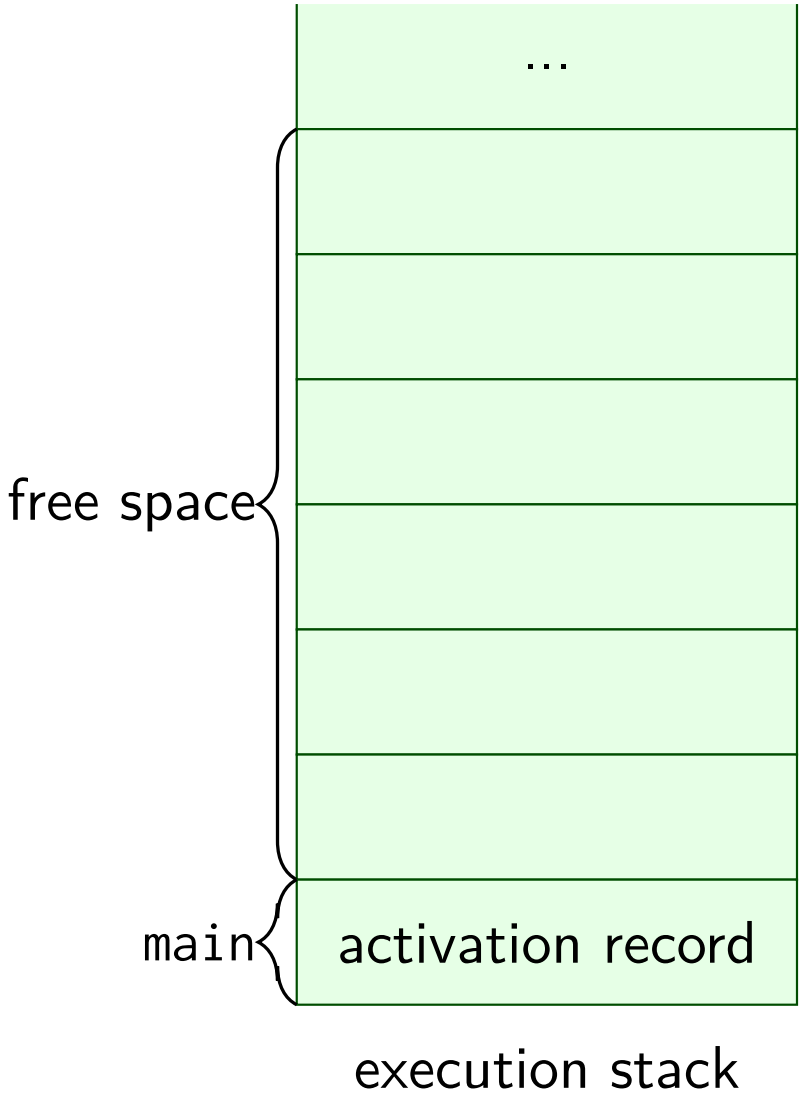
Alprogramhívások nyilvántartása

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



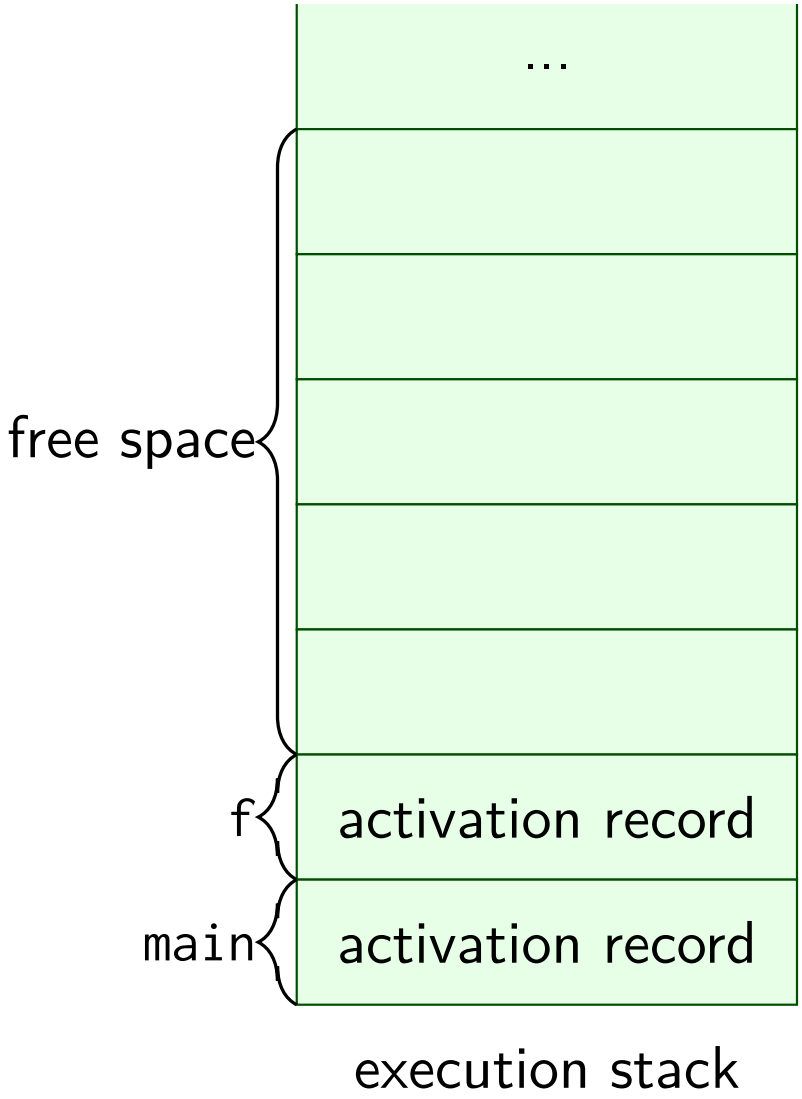
Alprogramhívások nyilvántartása

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



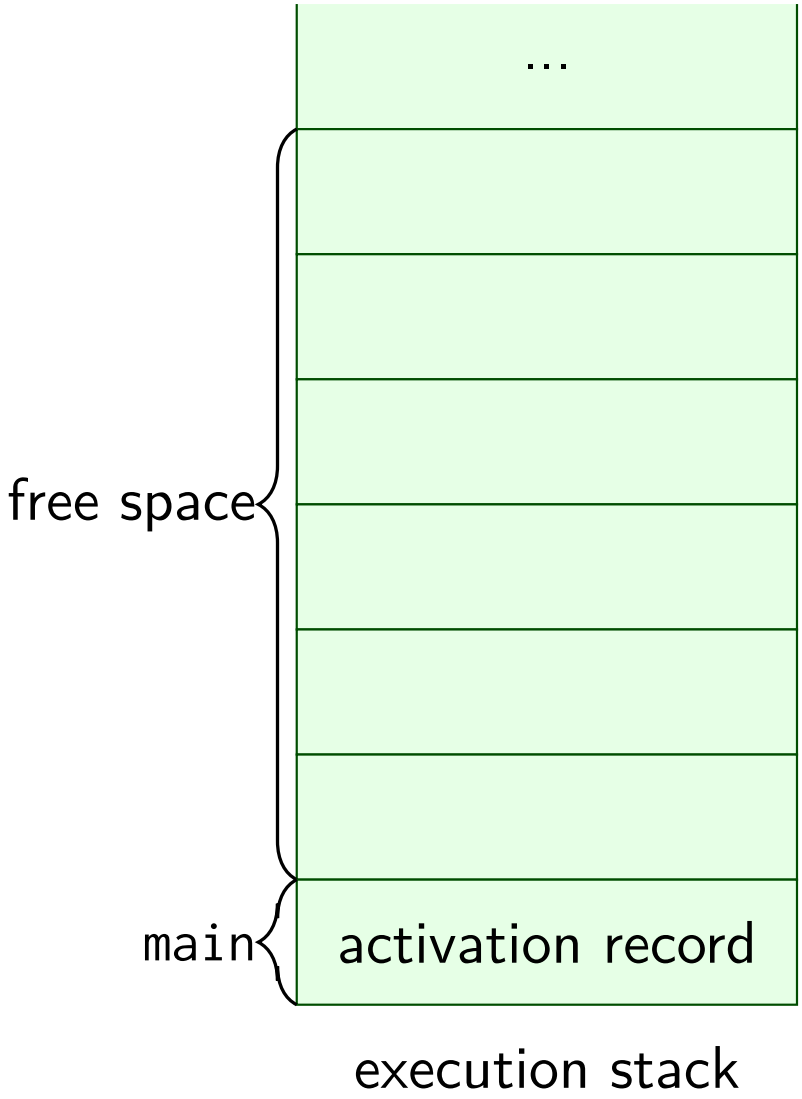
Alprogramhívások nyilvántartása

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



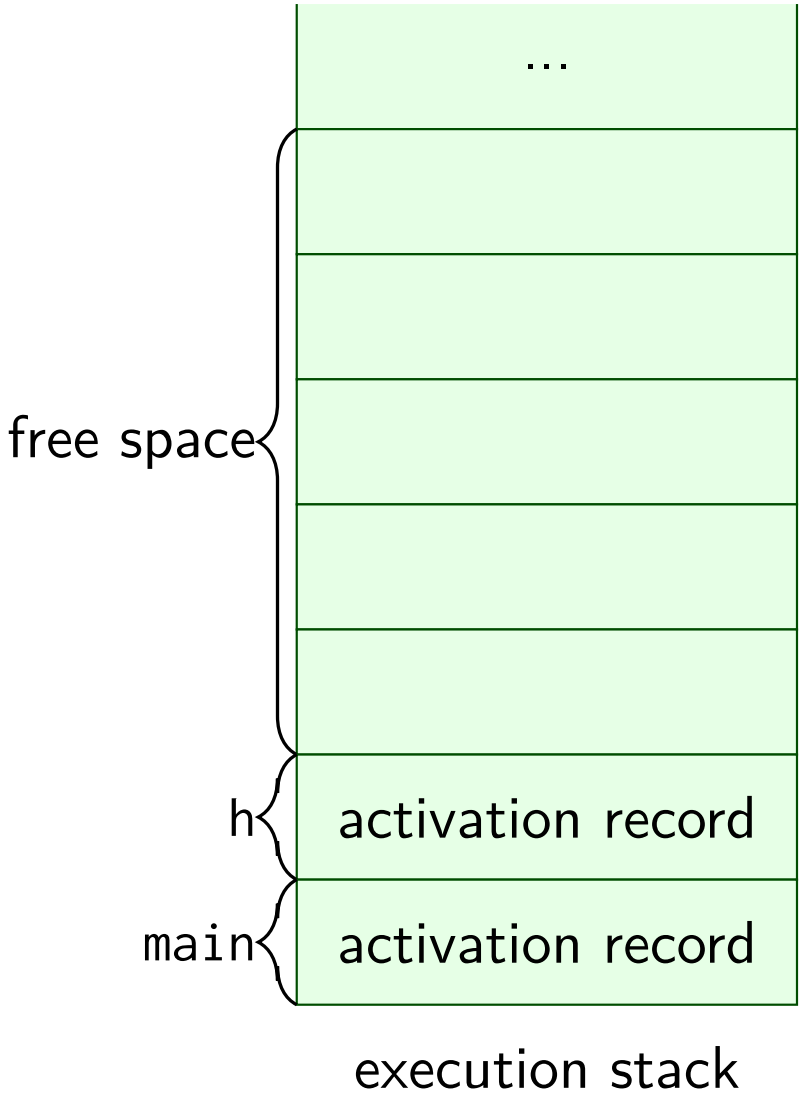
Alprogramhívások nyilvántartása

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



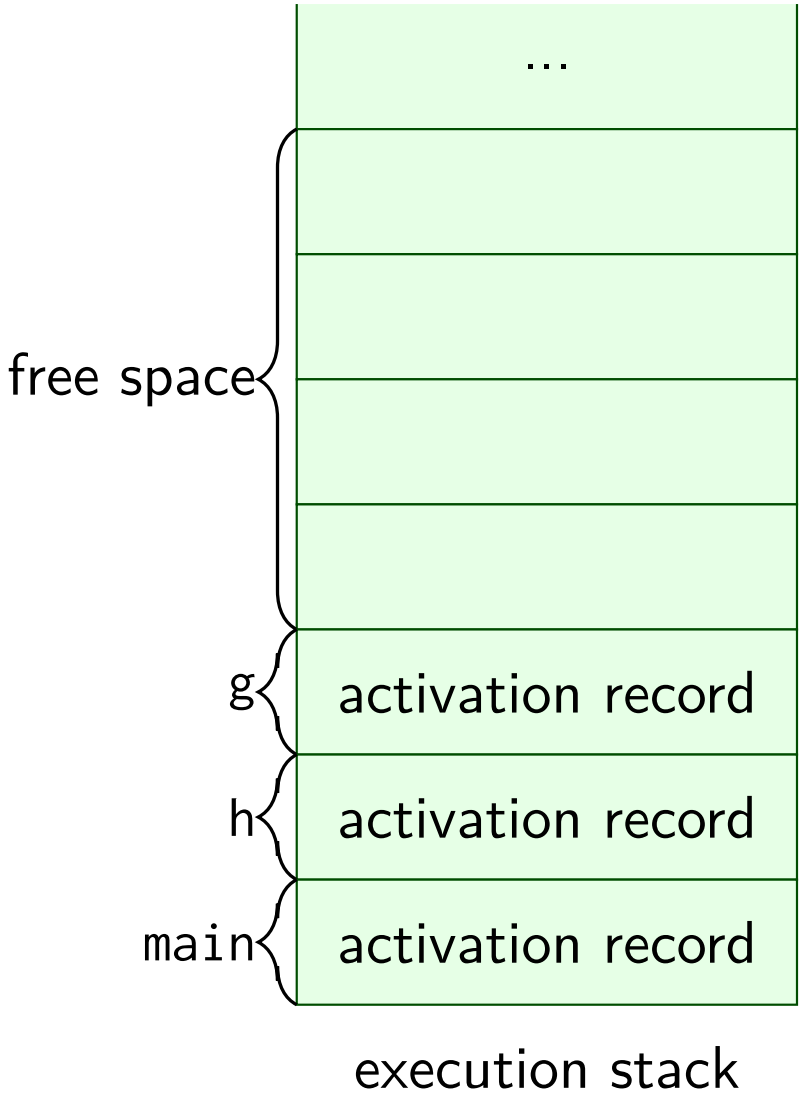
Alprogramhívások nyilvántartása

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



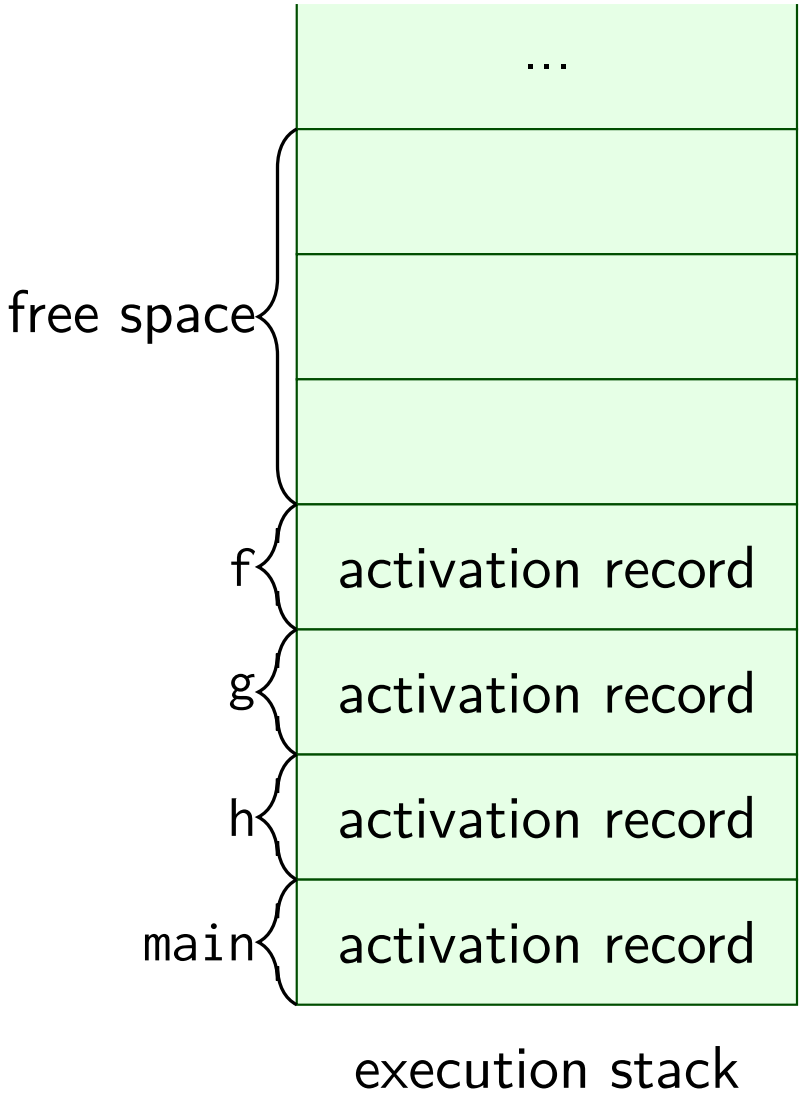
Alprogramhívások nyilvántartása

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



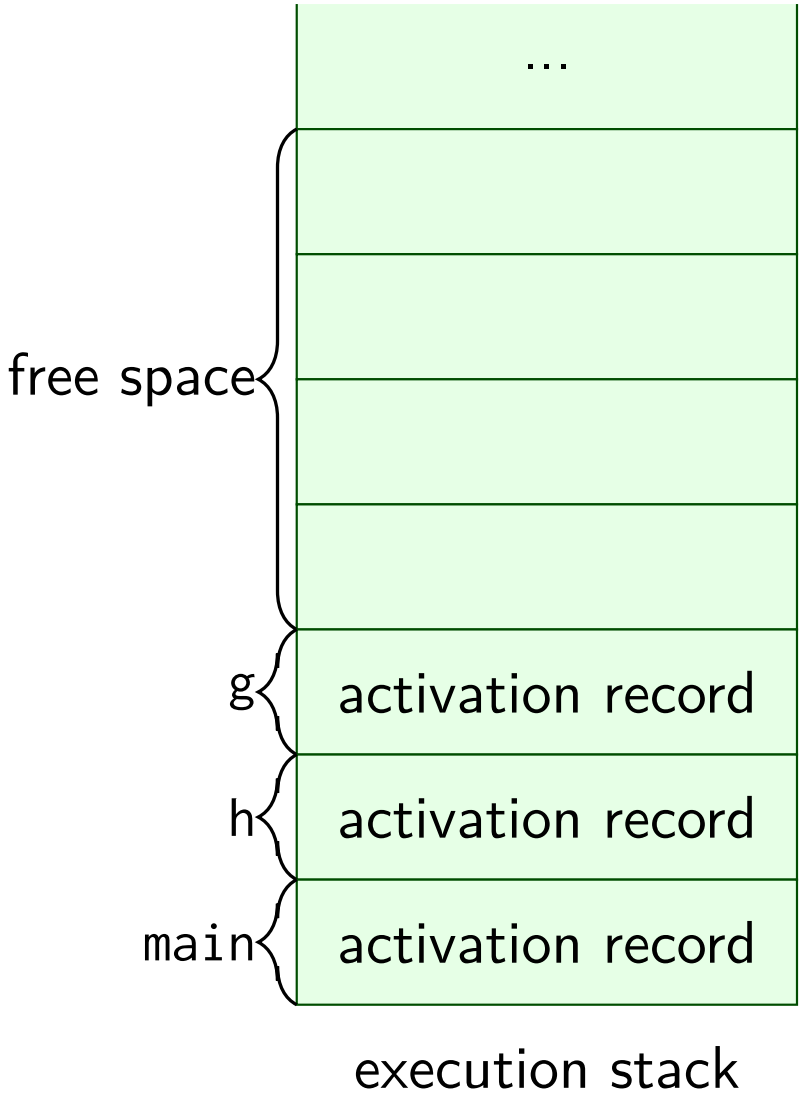
Alprogramhívások nyilvántartása

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



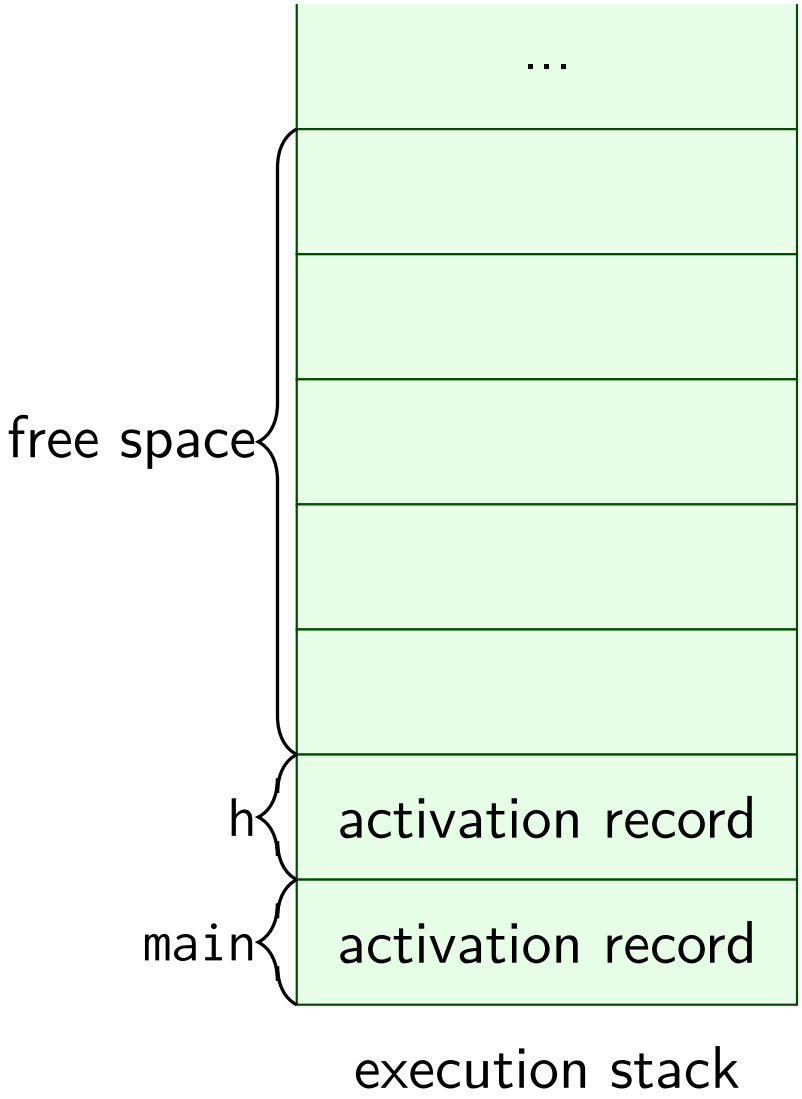
Alprogramhívások nyilvántartása

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



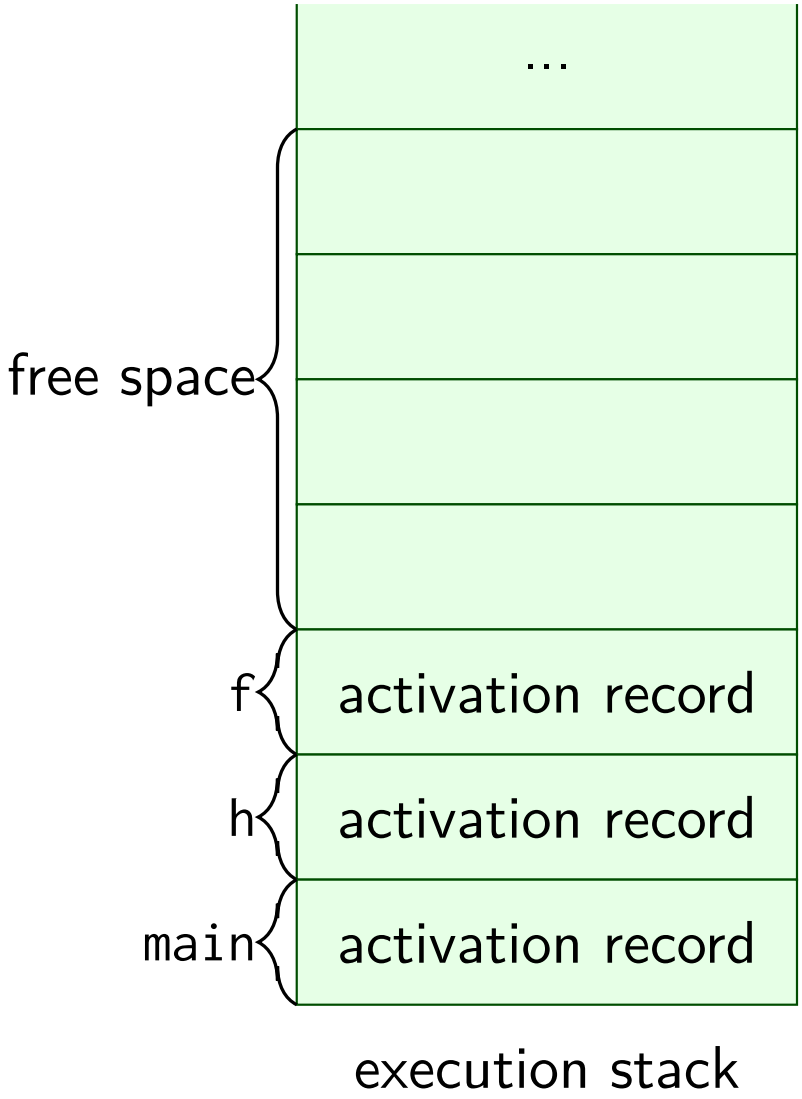
Alprogramhívások nyilvántartása

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



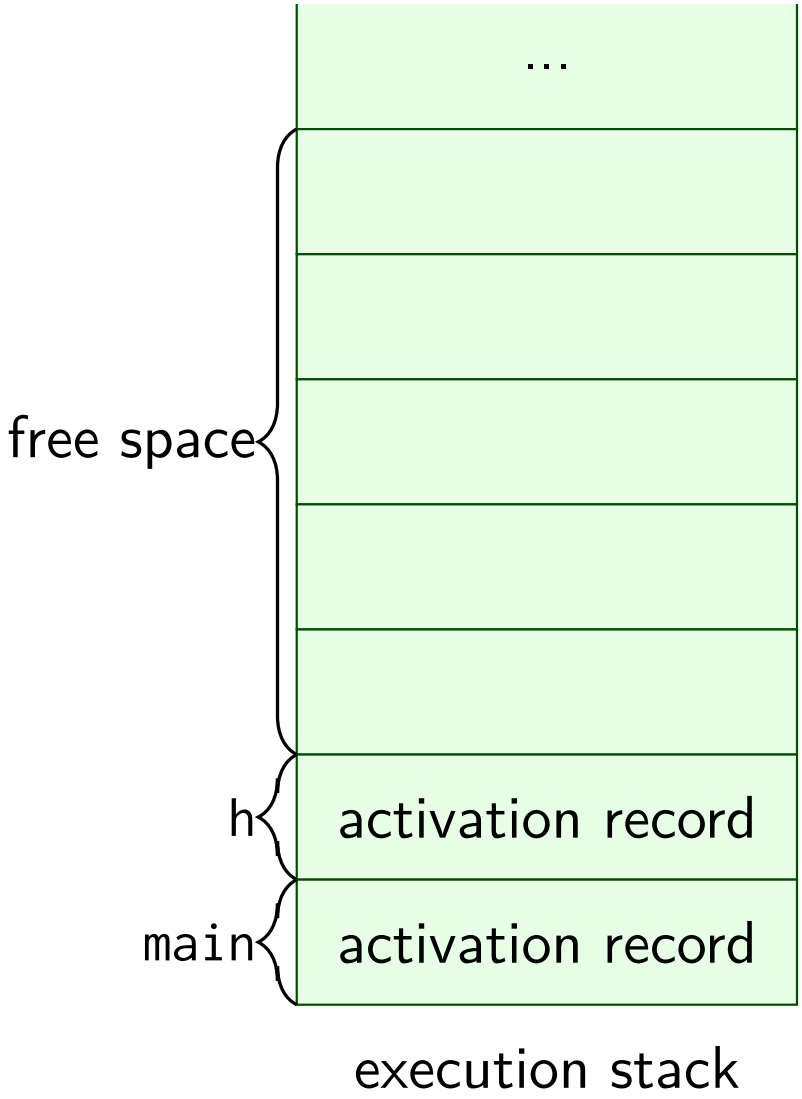
Alprogramhívások nyilvántartása

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



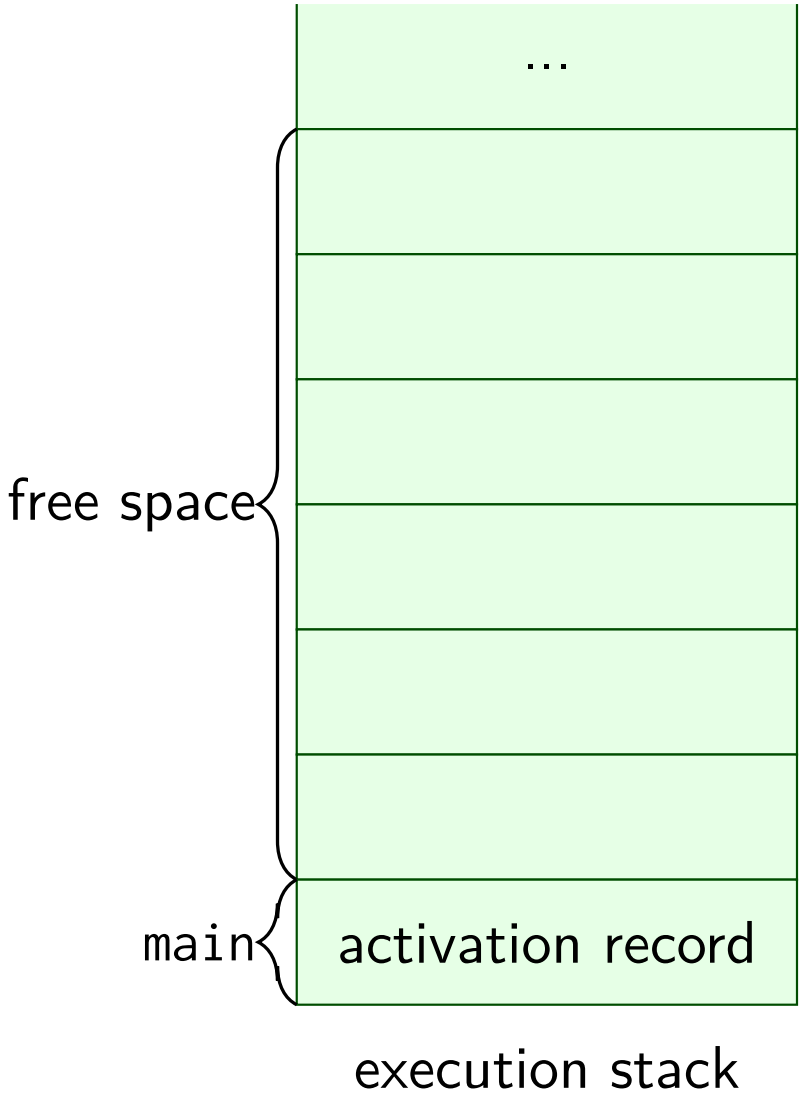
Alprogramhívások nyilvántartása

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



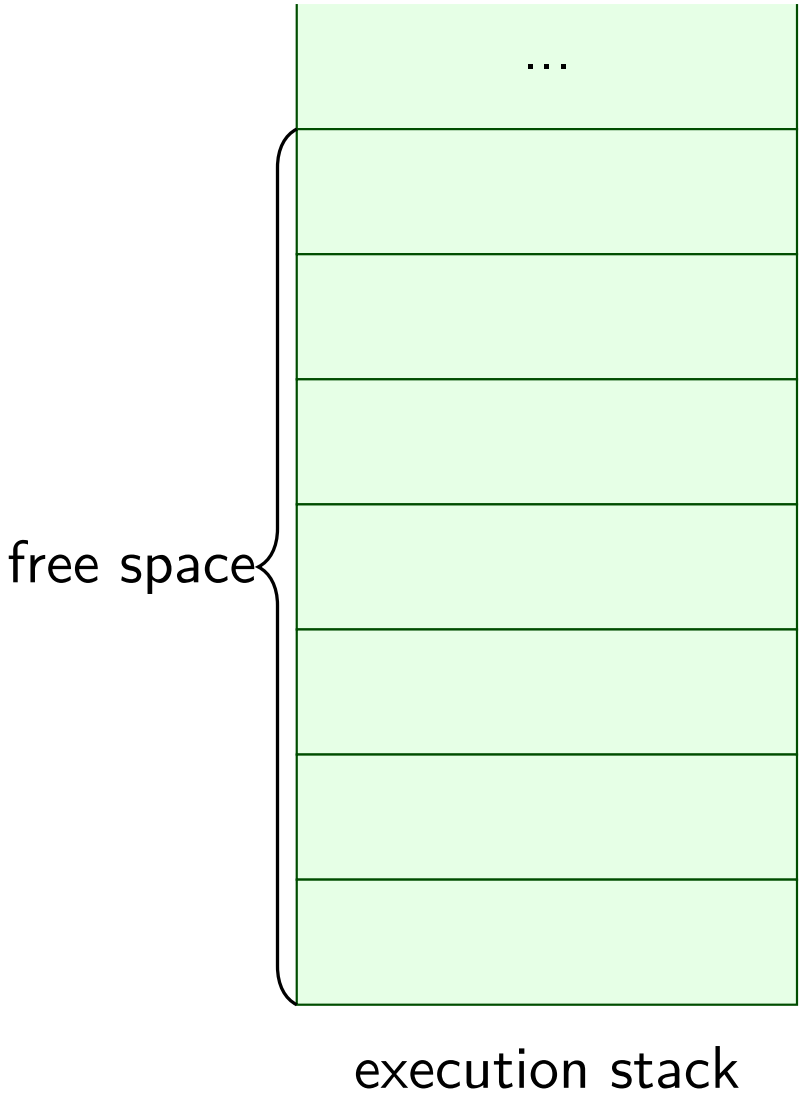
Alprogramhívások nyilvántartása

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



Alprogramhívások nyilvántartása

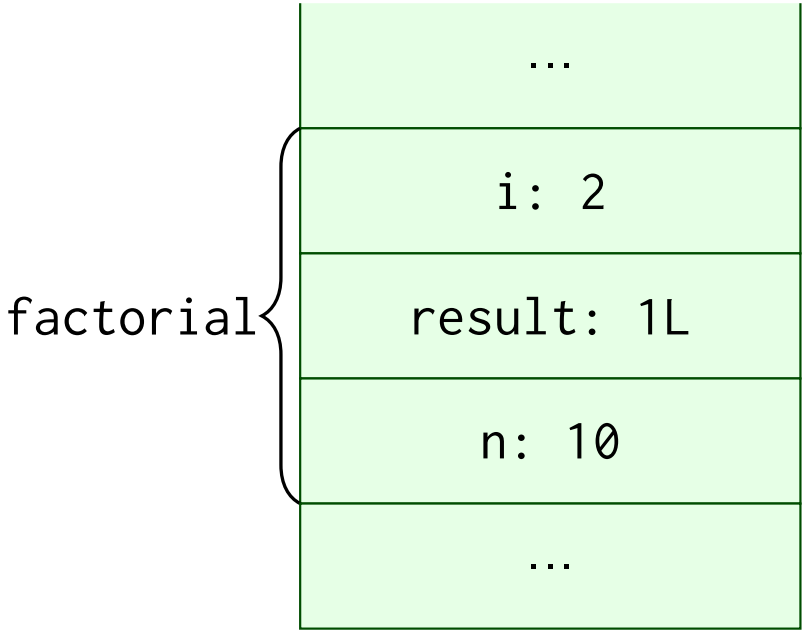
```
void f(void)
{
}
void g(void){
    f();
}
void h(void){
    g();
    f();
}
int main()
{
    f();
    h();
    return 0;
}
```



Aktivációs rekord

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

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



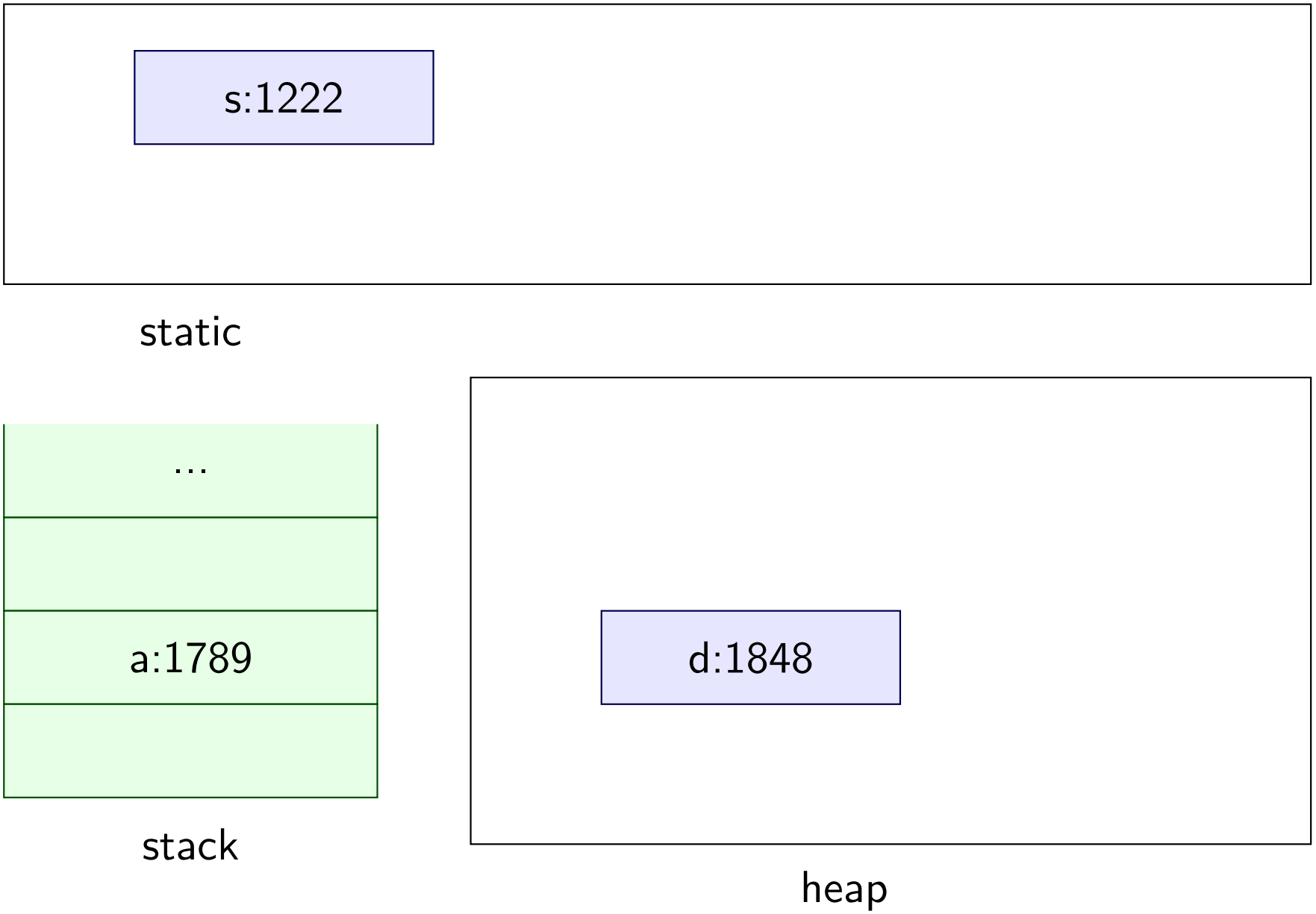
Rekurzió

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

„Változók” tárolása a memóriában

- statikus tárhely $\rightarrow$ statikus
- végrehajtási verem $\rightarrow$ automatikus
- dinamikus tárhely (heap) $\rightarrow$ dinamikus

static - stack - heap



Statikus tárolású változó

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

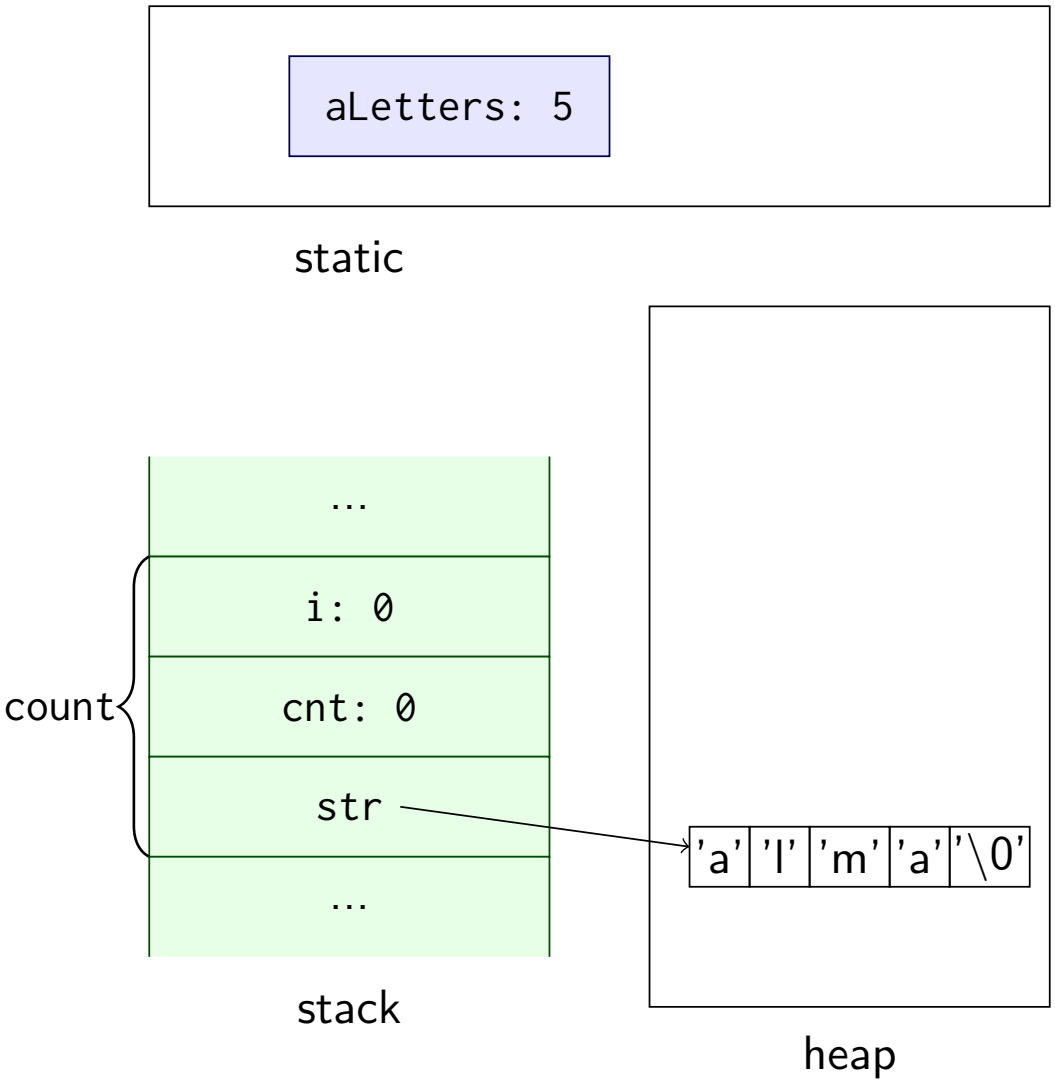
```
int counter = 0;
void signal(void)
{
    ++ counter;
}
```

Automatikus tárolású változó

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

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

static - stack - heap




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

C - statikus lokális változók

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

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

}
```

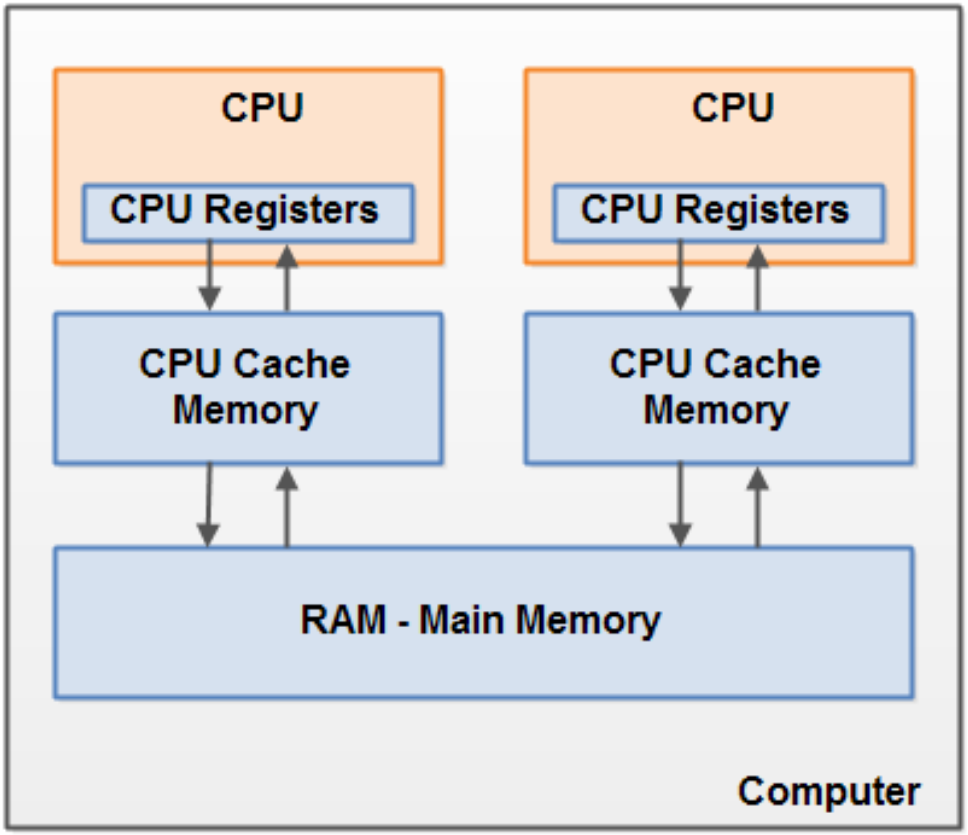
```
int signal(void)
{
    static int counter = 0;
    ++ counter;
    return counter;
}
```

Tárolási mód kifejezése

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

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

Számítógép memóriája



Optimalizáció: memóriaműveletek emberi skálán

Forrás: David Jeppesen

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

Globális változók használata

Kerülendő!

Változók definiálása

Deklarációval

- Statikus és automatikus tárolású
  - Statikus tárhely
  - Végrehajtási verem
- Élettartam: programszerkezetből
  - A hatókör
  - Kivéve lokális statikus

Allokáló utasítással

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

Blokk utasítás

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

Alprogram paraméterei

- Definícióban: formális paraméterlista
- Hívásnál: aktuális paraméterlista

Paraméterátadási technikák

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

Érték szerinti paraméterátadás

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

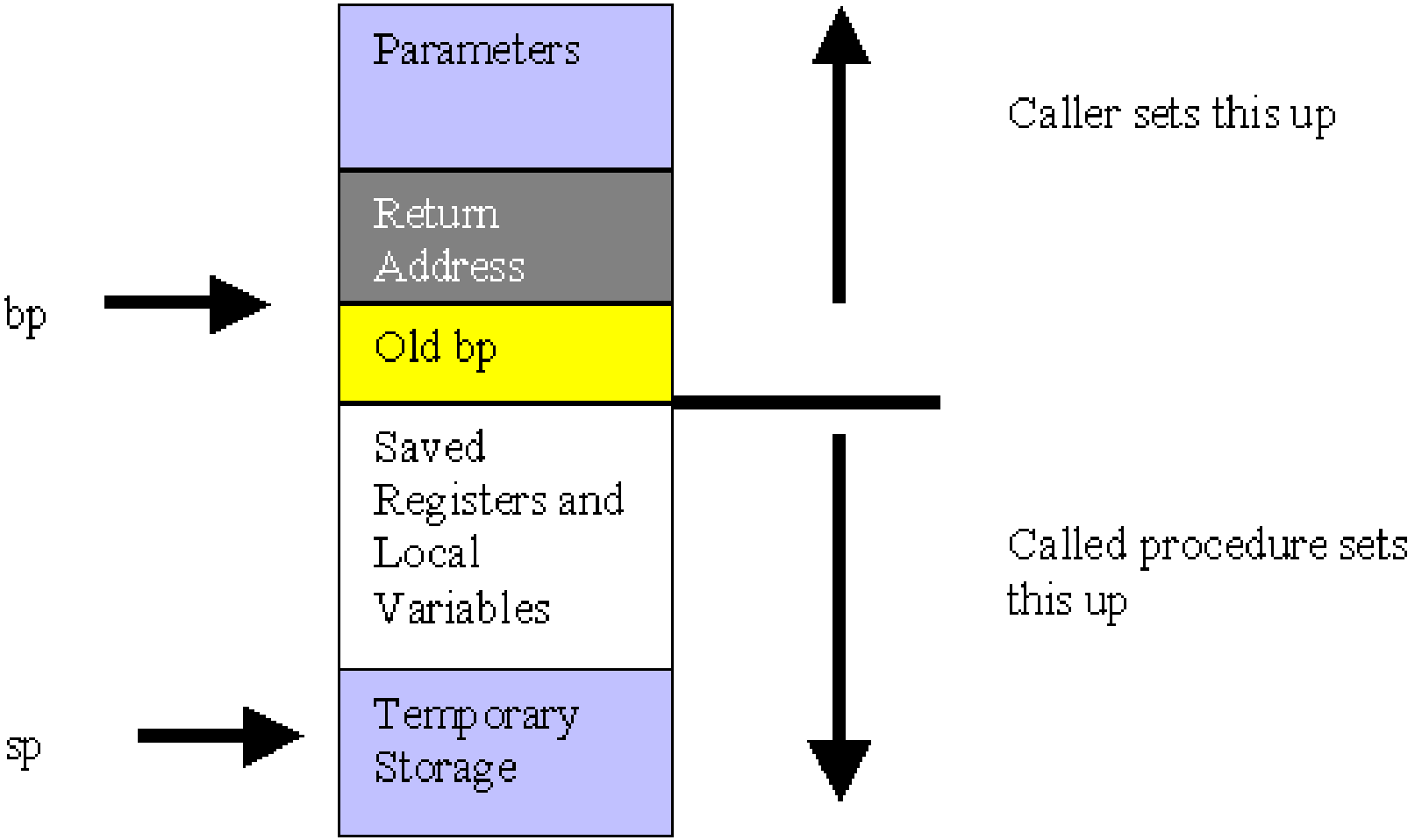
Érték szerinti paraméterátadás – példa

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

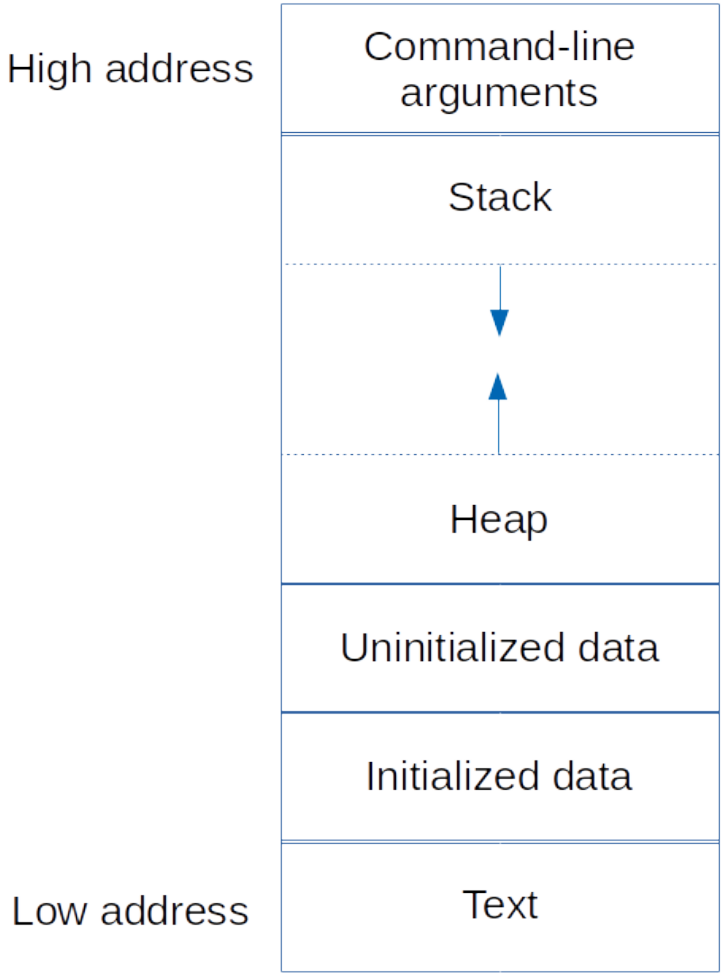
Aktivációs rekord

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

Precízebben



C programok címtere



Outline

Dinamikus memóriakezelés

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

Mutatók C-ben

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

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

Összetevők

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

Mire jó?

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

Dinamikus méretű adatszerkezet

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

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

Kerülendő megoldás

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

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

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

Láncolt adatszerkezet

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

Bejárás közben konstans idejű törlés/beszúrás

Aliasing

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

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

Felszabadítás

Minden dinamikusan létrehozott változót pontosan egyszer!

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

Felszabadított változóra hivatkozni hiba!

Hivatkozás felszabadított változóra

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

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

Többszörösen felszabadított változó

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

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

Fel nem szabadított változó

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

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

Tulajdonos?

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

Könnyű elrontani!

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

Alias

ugyanarra a tárterületre többféle névvel hivatkozhatunk

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

Mutató gyűjtőtípusa

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

Mutató gyűjtőtípusa: típuskényszerítés

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

Dinamikus tárhely elérése

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

Mutató nem dinamikus változóra

```
int global = 1;

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

Érvénytelen mutató

Értelmetlen

```
int *make_ptr(void)
{

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

Értelmes

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

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

Outline

Statikus programszerkezet

- kifejezés
- utasítás
- alprogram
- modul

Modul

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

Modulok C-ben

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

C - statikus globális deklarációk

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

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

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

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

Több modulból álló C program

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

positive.c

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

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

main.c

```c
#include <stdio.h>

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

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

Fejállományok

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

Include guard

vector.h (részlet)

```
#ifndef VECTOR_H
#define VECTOR_H

#define VEC_EOK      0
#define VEC_ENOMEM   1

struct VECTOR_S;
typedef struct VECTOR_S *vector_t;

extern int vectorErrno;

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

#endif
```

Outline

Függvénydeklarációk és -definíciók

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

int h();
int i(void);

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

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

extern int k(int,int);

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

Paraméterátadási technikák

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

Érték szerinti paraméterátadás

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

Bemenő szemantika

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

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

Mutató átadása érték szerint

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

```
int main()
{
    int *n, *m;
    n = (int*) malloc(sizeof(int));
    m = (int*) malloc(sizeof(int));
    if( n != NULL && m != NULL )
    {
        *n = 1984;
        *m = 356;
        swap(n,m);
        printf("%d %d\n",*n,*m);
        free(n); free(m);
        return 0;
    } else return 1;
}
```

Cím szerinti paraméterátadás emulációja

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

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

Cím szerinti paraméterátadás – Pascal

```
program swapping;

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

   var n, m: integer;

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

Cím szerinti paraméterátadás – C++

```
#include <cstdio>

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

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

Érték-eredmény szerinti paraméterátadás: Ada

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

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

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

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

Megosztás szerinti paraméterátadás

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

Ez nem cím szerinti paraméterátadás

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

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

Automatikus változó visszaadása?

Hibás!

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

Igény szerinti paraméterátadás

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

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

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

Szövegszerű helyettesítés

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

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

Szövegszerű helyettesítés – becsapós

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

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

Szövegszerű helyettesítés – zárójelezés

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

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

Szövegszerű helyettesítés – még így is veszélyes

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

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

Outline

Típuskonstrukciók

- Felsorolási típusok
- Mutató típusok
- Összetett típusok

Felsorolási típus

Haskell

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

C-ben: valójában egy egész szám típusra képződik le

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

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

Felsorolási típus C-ben

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

typedef enum color Color;

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

int main( int argc, char *argv[] )
{
    for( --argc; argc>0; --argc )
    {
       printf("%s\n", property( atoi(argv[argc]) ));
    }
    return 0;
}
```

Összetett típusú értékek

- Sorozat
- Direktszorzat
- Unió típus
- Osztály

Sorozat típusok

- C tömbök
- Haskell listák

Sorozat: azonos típusú elemekből álló összetett típus

Tömb fogalma

Azonos típusú (méretű) objektumok egymás után a memóriában.

- Bármelyik hatékonyan elérhető!
- Rögzített számú objektum!

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

Indexelés 0-tól

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

C tömbök deklarációja

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

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

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

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

Tömbök indexelése

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

Mutatók

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

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

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

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

*p = 5;          /* dereferálás */
```

Deklarációk mutatókkal

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

char *str = "Hello!";

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

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

Tömbök és mutatók kapcsolata

- Tömb: *second-class citizen*
- Tömb $\rightarrow$ mutató
- Nem ekvivalensek!

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

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

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

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

Tömb átadása paraméterként?

Valójában mutató típusú a paraméter!

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

Mutató-aritmetika – léptetések

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

Mutató-aritmetika – összehasonlítások

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

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

Mutató-aritmetika – indexelés

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

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

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

Mutató-aritmetika: példa

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

Mutatók és tömbök közötti különbségek

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

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

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

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

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

Tömbök átadása paraméterként: általánosítás?

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

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

Tömbök paraméterként: fordítási időben rögzített méret

```
#define DIMENSION 3

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

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

Tömbök paraméterként: futási időben rögzített méret?

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

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

Tömbök paraméterként: hibás megközelítés

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

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

Tömbök paraméterként: helyesen

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

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

Bonyolult struktúra átadása paraméterként

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

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

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

Több dimenziós tömbök paraméterként

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

transpose(m);

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

Túl merev megoldás

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

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

Sorfolytonos ábrázolás: egybefüggő memóriaterület

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

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

Alternatív reprezentáció: mutatók tömbje

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

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

Magasabbrendű függvények

függvénymutatók segítségével

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

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

C: függvénymutatók

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

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


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

Függvénymutatók - néhány észrevétel

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

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

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

Konstansok

Definíciók

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

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


const int *p = &i;

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

Hibás használat

```
i = 4;
j = 4;

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

*p = 4;


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

Nem teljes a biztonság

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

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

const-ra polimorf megoldás nincs

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

```
char *p = strchr("Hello",'e'),
*p = 'o';    /* hibás! */
```

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

Élettartammal kapcsolatos hibák

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

(legvégén)

Direktszorzat típusok

(Potenciálisan) különböző típusú elemekből konstruált összetett típus

- tuple
- rekord
- struct

C struct

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

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

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

C struct

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

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

typedef struct person Person;

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

Paraméterátadás

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

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

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

Unió típus

Típusértékei több típus valamelyikéből

C: union

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

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

Címkézett unió

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

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

Egységes használat

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

void move( struct shape *aShape, int dx, int dy ){
    aShape->x += dx;
    aShape->y += dy;
}
```

Használat esetszétválasztással

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

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

Biztonságos létrehozás

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

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

Osztály

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

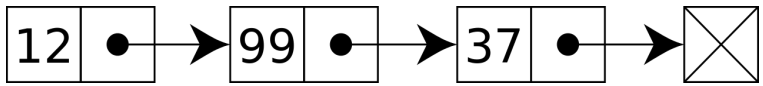
Outline

Adatszerkezetek

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

Láncolt adatszerkezetek

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



Sorozatok ábrázolása

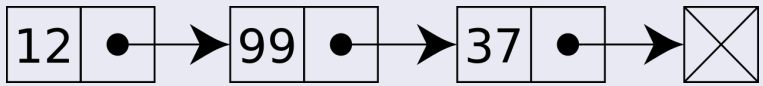
Tömb

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

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

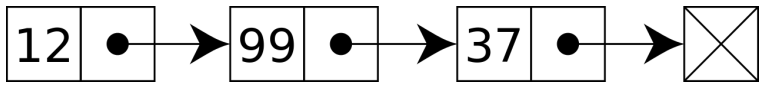
Láncolt lista

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



Láncolt lista

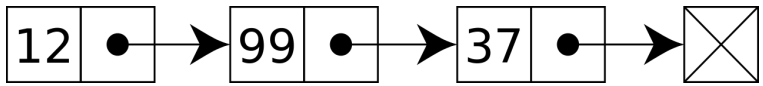
```
struct node
{
    int data;
    struct node *next;
};
```



Láncolt lista felépítése

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

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



Outline

Egyenlőségvizsgálat és másolás elemi típusokon

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

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

Mutatókkal?

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

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

Tömbökkel?

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

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

Tömbökkel!

```
#define SIZE 3

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

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

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

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

Struktúrákkal?

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

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

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

Struktúrákkal!

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

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

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

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

Láncolt lista?

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

struct node *a, *b;
...

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

Sekély megoldás – nem jó ide

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

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

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

Mély egyenlőségvizsgálat

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

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

Mély másolás

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

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

Outline

Hibakezelés

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

Hibakezelés C-ben

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

C hibakezelés hátulütői

- Túl sok elágazás, feltételvizsgálat
  - A hibakezelés akár a kód 30-40%-a is lehet
  - Elvész a kódban a lényeg
- Kispórolt hibakezelés veszélye
- Elfelejtett hibakezelés veszélye

Hibakezelés modern nyelvekben

Nyelvi támogatás!

Kivételek

- Kivétel kiváltódása és kiváltása
- Kivétel terjedése
- Kivétel lekezelése

Kivétel terjedése

Vezérlésátadás!

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