

# Programozási nyelvek 1

Szathmáry László  
Debreceni Egyetem  
Informatikai Kar

## 11. előadás

- void, tiszta függvény, tipikus hiba, stílus #3, assert
- rekurzió
- header állományok

(utolsó módosítás: 2020. ápr. 26.)

2019-2020, 2. félév



# void

void kulcsszó a formális paraméterlista helyén

A függvény NEM fogad semmilyen inputot sem. Ha mégis úgy hívnánk meg, hogy az aktuális paraméterlistán valamilyen értéket átadunk a függvénynek, akkor fordítási hibát kapunk.

```
1  #include <stdio.h>
2
3  void hello(void) ←
4  {
5      puts("Hello, World!");
6  }
7
8  int main(void) ←
9  {
10     hello();
11
12     return 0;
13 }
14
```

pl. `hello(42);` fordítási hibát adna

# tiszta függvény (*pure function*)

Egy tiszta függvényre a következők jellemzőek:

- **Nincs mellékhatása**, vagyis nem változtatja meg a környezetét (a függvényen kívül SEMMIT sem változtat meg).
- Ugyanazokkal a paraméterekkel meghívva MINDIG ugyanazt az eredményt adja vissza (**determinisztikus**).

Azt a szituációt, amikor egy függvény megváltoztatja a paramétereit vagy a környezetét, a függvény **mellékhatásának** nevezzük.

Mitől lesz egy függvény nem-tiszta?

- I/O művelet
- globális változó használata

# tiszta függvény

Mire jó? Miért jó?

Egy tiszta függvényt sokkal egyszerűbb így használni. Tudom, hogy ha meghívom, akkor az semmit sem fog elrontani a környezetében. Még véletlenül sem módosít semmit a környezetén.

Emiatt nagyon egyszerűen lehet ezeket a függvényeket tesztelni (*unit* tesztek).

Ha egy függvényt meg tudunk írni tisztán is, akkor inkább így írjuk meg.

Egy programban szükség van nem-tiszta függvényekre is, de törekedjünk arra, hogy minél több függvényünk tiszta legyen. A programunk így biztonságosabb lesz.

# tipikus hiba függvényeknél

Írjon függvényt, ami egy egész számról eldönti, hogy páros-e!

```
6
7 int paros_e(int n)
8 {
9     if (n % 2 == 0)
10    {
11        puts("páros");
12    }
13    else
14    {
15        puts("páratlan");
16    }
17 }
```

Ez NEM függvény!

```
6
7 int paros_e(int n)
8 {
9     if (n % 2 == 0)
10    {
11        return 1;
12    }
13    else
14    {
15        return 0;
16    }
17 }
```

Ez már jó. Ez egy logikai igaz / hamis értéket ad vissza.

# stílus #3: beszédes nevek

Nagyon fontos, hogy a változóknak / függvényeknek jó nevet adjunk! Egy jó névből már ki lehet találni, hogy mire való az a változó, mit csinál az a függvény.

```
int paros_e(int n);  
int is_even(int n);
```

```
int w = 5;      // width  
int h = 10;     // height
```

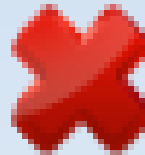
```
int width = 5;  
int height = 10;
```

# stílus #3: beszédes nevek

Nagyon fontos, hogy a változóknak / függvényeknek jó nevet adjunk! Egy jó névből már ki lehet találni, hogy mire való az a változó, mit csinál az a függvény.

```
int paros_e(int n);  
int is_even(int n);
```

```
int w = 5;    // width  
int h = 10;   // height
```



```
int width = 5;  
int height = 10;
```



# assert

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int terület(int a, int b)
5  {
6      assert(a >= 0);
7      assert(b >= 0);
8      //
9      return a * b;
10 }
11
12 int main()
13 {
14     int a = -5;
15     int b = 10;
16
17     int t = terület(a, b);
18     printf("terület: %d\n", t);
19
20     return 0;
21 }
22
```

állítások, melyeknek teljesülniük kell

Amennyiben nem teljesül egy feltétel, akkor a program azon a ponton megáll s hibaüzenetet kapunk.

Nem hibakezelésre való!

Debug módban lehet használni, a kész kódban viszont rendes hibakezelést használjunk!

assert: assert.c:6: terület: Assertion `a >= 0' failed.



# assert

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  int terület(int a, int b)
5  {
6      assert(a >= 0);
7      assert(b >= 0);
8      //
9      return a * b;
10 }
11
12 int main()
13 {
14     int a = -5;
15     int b = 10;
16
17     int t = terület(a, b);
18     printf("terület: %d\n", t);
19
20     return 0;
21 }
22
```

Hogyan lehet egyszerűen figyelmen kívül hagyni az `assert( )` sorokat?

```
#define NDEBUG
#include <assert.h>
```

Fontos, hogy az `NDEBUG` név definiálása ilyenkor **előzze meg** az `assert.h` állomány beépítését!

# rekurzió

Nézzük a klasszikus példát:  $n!$

Iteratív definíció:

$$\begin{aligned} n! &= n * (n-1) * (n-2) * \dots * 1 \\ 5! &= 5 * 4 * 3 * 2 * 1 \end{aligned}$$

Rekurzív definíció:

$$\begin{aligned} n! &= n * (n-1)! \\ 5! &= 5 * 4! \end{aligned}$$

$$0! = 1! = 1$$

**Rekurzió:** az algoritmus meghívja önmagát.

# rekurzió

```
2
3 int fakt(int n)
4 {
5     int x = 1;
6
7     for (int i = 1; i <= n; ++i) {
8         x *= i;
9     }
10
11     return x;
12 }
13
14 int main()
15 {
16     int result = fakt(5);
17
18     printf("%d\n", result);
19
20     return 0;
21 }
22
```

iteratív változat

```
2
3 int fakt(int n)
4 {
5     if (n == 0 || n == 1) {
6         return 1;
7     }
8     // else
9     return n * fakt(n-1);
10 }
11
12 int main()
13 {
14     int result = fakt(5);
15
16     printf("%d\n", result);
17
18     return 0;
19 }
20
```

rekurzív változat

# rekurzió

## Hívási lánc

Egy programegység bármikor meghívhat egy másik programegységet, az egy újabb programegységet, és így tovább. Így kialakul egy **hívási lánc**. A hívási lánc első tagja mindig a főprogram.

A hívási lánc minden tagja *aktív*, de csak a legutoljára meghívott programegység *működik*. Szabályos esetben mindig az utoljára meghívott programegység fejezi be legelőször a működését, és a vezérlés visszatér az őt meghívó programegységbe. A hívási lánc futás közben dinamikusan épül föl és bomlik le.

Azt a szituációt, amikor egy aktív alprogramot hívunk meg, rekurciónak nevezzük.

A rekurzió lehet:

- közvetlen: egy alprogram önmagát hívja meg
- közvetett: a hívási láncban már korábban szereplő alprogramot hívunk meg

A rekurzióval megvalósított algoritmus mindig átírható iteratív algoritmussá.

# rekurzió

## Hívási lánc

```
2
3 void b()
4 {
5     puts("b()");
6 }
7
8 void a()
9 {
10    puts("a()");
11    b();
12 }
13
14 int main()
15 {
16    puts("main()");
17    a();
18
19    return 0;
20 }
21
```

# rekurzió

## Hívási lánc

```
2
3 void b()
4 {
5     puts("b()");
6 }
7
8 void a()
9 {
10    puts("a()");
11    b();
12 }
13
14 int main()
15 {
16    puts("main()");
17    a();
18
19    return 0;
20 }
21
```



main()

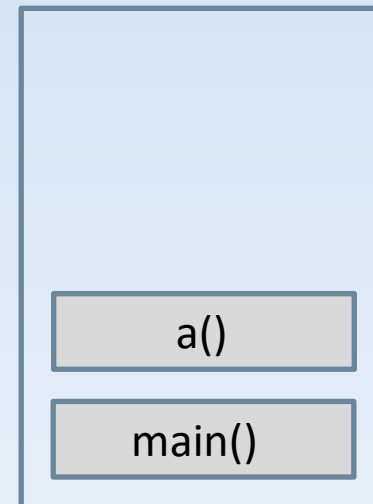
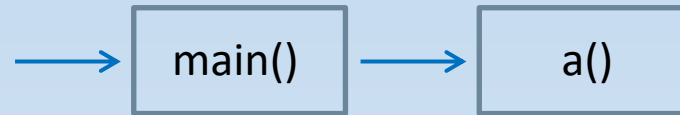
stack

main()

# rekurzió

## Hívási lánc

```
2
3 void b()
4 {
5     puts("b()");
6 }
7
8 void a()
9 {
10    puts("a()");
11    b();
12 }
13
14 int main()
15 {
16    puts("main()");
17    a();
18
19    return 0;
20 }
21
```

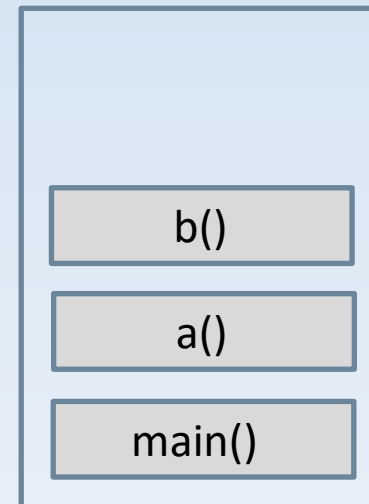
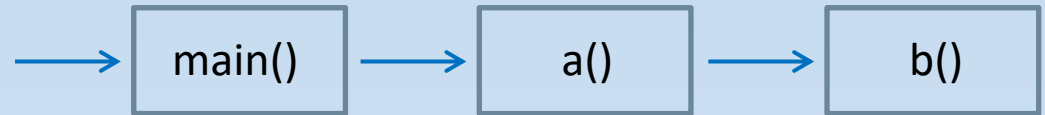


stack

# rekurzió

## Hívási lánc

```
2
3 void b()
4 {
5     puts("b()");
6 }
7
8 void a()
9 {
10    puts("a()");
11    b();
12 }
13
14 int main()
15 {
16    puts("main()");
17    a();
18
19    return 0;
20 }
21
```



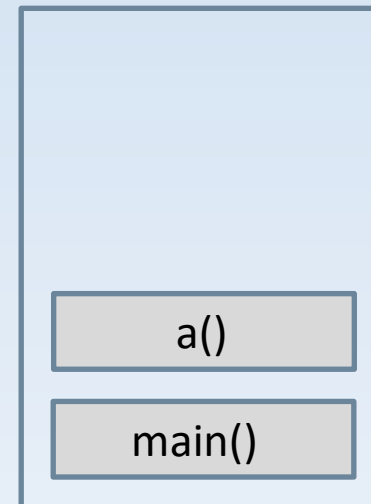
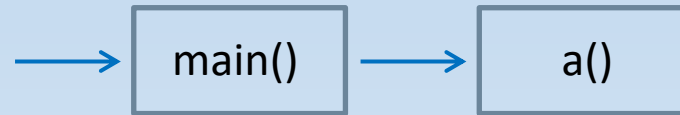
stack



# rekurzió

## Hívási lánc

```
2
3 void b()
4 {
5     puts("b()");
6 }
7
8 void a()
9 {
10    puts("a()");
11    b();
12 }
13
14 int main()
15 {
16    puts("main()");
17    a();
18
19    return 0;
20 }
21
```



stack

# rekurzió

## Hívási lánc

```
2
3 void b()
4 {
5     puts("b()");
6 }
7
8 void a()
9 {
10    puts("a()");
11    b();
12 }
13
14 int main()
15 {
16    puts("main()");
17    a();
18
19    return 0;
20 }
21
```



main()

stack

main()

# rekurzió

## Hívási lánc

```
2
3 void b()
4 {
5     puts("b()");
6 }
7
8 void a()
9 {
10    puts("a()");
11    b();
12 }
13
14 int main()
15 {
16    puts("main()");
17    a();
18
19    return 0;
20 }
21
```

stack

# rekurzió

```
2
3 int fakt(int n)
4 {
5     if (n == 0 || n == 1) {
6         return 1;
7     }
8     // else
9     return n * fakt(n-1);
10 }
```

Meghívása:

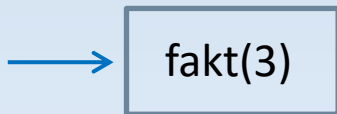
```
int result = fakt(3);
```

# rekurzió

```
2
3  int fakt(int n)
4  {
5      if (n == 0 || n == 1) {
6          return 1;
7      }
8      // else
9      return n * fakt(n-1);
10 }
```

Meghívása:

```
int result = fakt(3);
```

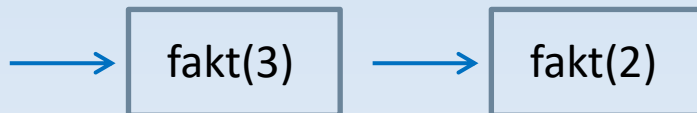


# rekurzió

```
2
3 int fakt(int n)
4 {
5     if (n == 0 || n == 1) {
6         return 1;
7     }
8     // else
9     return n * fakt(n-1);
10 }
```

Meghívása:

```
int result = fakt(3);
```

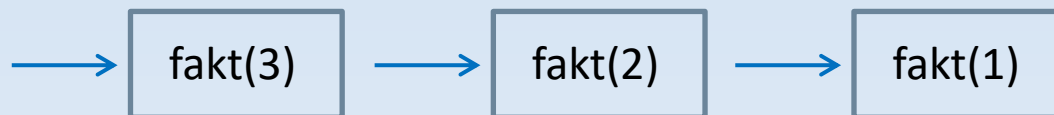


# rekurzió

```
2
3  int fakt(int n)
4  {
5      if (n == 0 || n == 1) {
6          return 1;
7      }
8      // else
9      return n * fakt(n-1);
10 }
```

Meghívása:

```
int result = fakt(3);
```

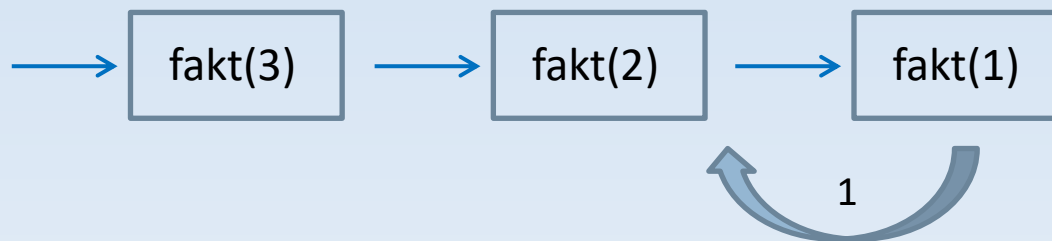


# rekurzió

```
2  
3 int fakt(int n)  
4 {  
5     if (n == 0 || n == 1) {  
6         return 1;  
7     }  
8     // else  
9     return n * fakt(n-1);  
10 }
```

Meghívása:

```
int result = fakt(3);
```



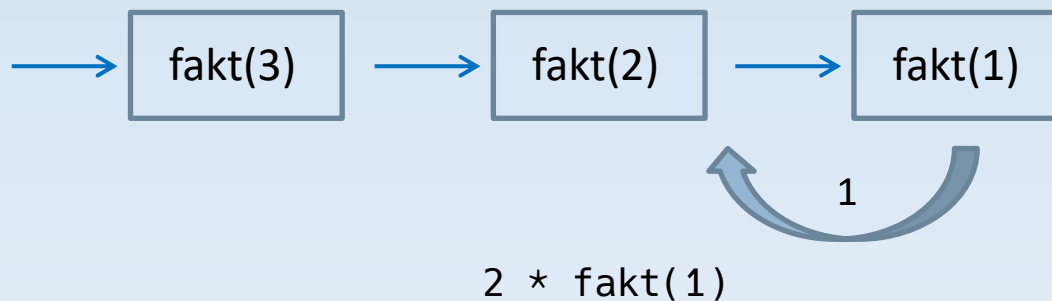


# rekurzió

```
2
3 int fakt(int n)
4 {
5     if (n == 0 || n == 1) {
6         return 1;
7     }
8     // else
9     return n * fakt(n-1);
10 }
```

Meghívása:

```
int result = fakt(3);
```

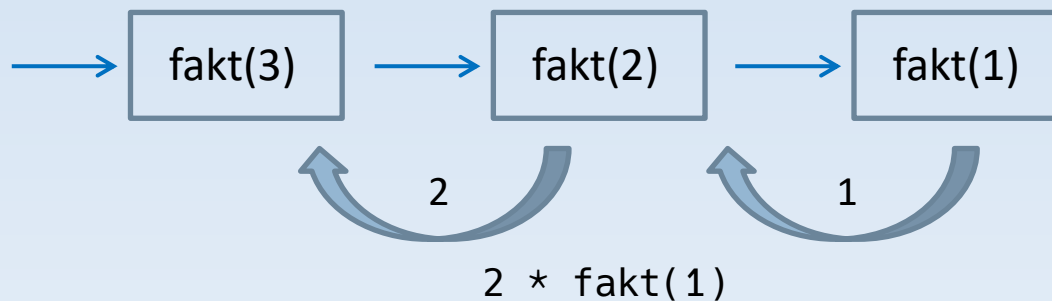


# rekurzió

```
2
3 int fakt(int n)
4 {
5     if (n == 0 || n == 1) {
6         return 1;
7     }
8     // else
9     return n * fakt(n-1);
10 }
```

Meghívása:

```
int result = fakt(3);
```

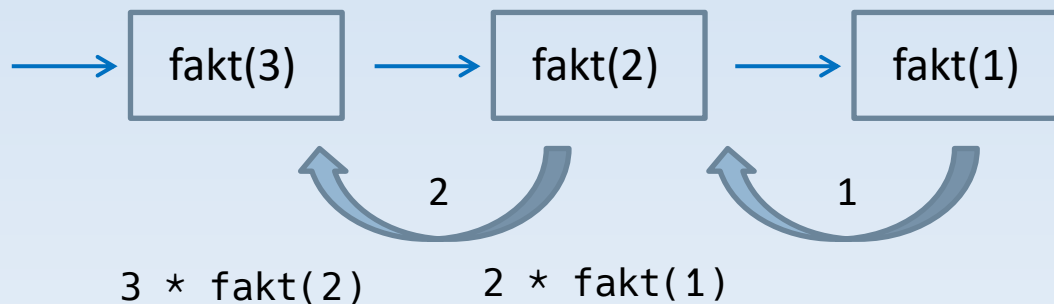


# rekurzió

```
2
3 int fakt(int n)
4 {
5     if (n == 0 || n == 1) {
6         return 1;
7     }
8     // else
9     return n * fakt(n-1);
10 }
```

Meghívása:

```
int result = fakt(3);
```

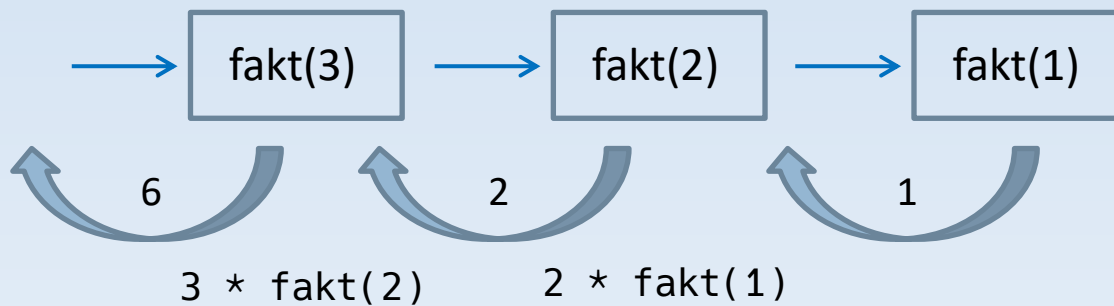


# rekurzió

```
2
3  int fakt(int n)
4  {
5      if (n == 0 || n == 1) {
6          return 1;
7      }
8      // else
9      return n * fakt(n-1);
10 }
```

Meghívása:

```
int result = fakt(3);
```



# rekurzió

Minden rekurzív függvény átírható iteratív módon.

Viszont vannak olyan algoritmusok, amiket rekurzívan sokkal egyszerűbben, rövidebben, olvashatóbban tudunk leírni.

Példák:

- gyorsrendezés (quicksort)
- bináris fa preorder, inorder, postorder bejárása
- stb.

# rekurzió

## THE C PROGRAMMING LANGUAGE

pointer initialization 102, 138  
 pointer, null 102, 198  
 pointer subtraction 103, 138, 198  
 pointer to function 118, 147, 201  
 pointer to structure 136  
 pointer, void \* 93, 103, 120, 199  
 pointer vs. array 97, 99–100, 104, 113  
 pointer-integer conversion 198–199, 205  
 pointers and subscripts 97, 99, 217  
 pointers, array of 107  
 pointers, operations permitted on 103  
 Polish notation 74  
 pop function 77  
 portability 3, 37, 43, 49, 147, 151, 153, 185  
 position of braces 10  
 postfix ++ and -- 46, 105  
 pow library function 24, 251  
 power function 25, 27  
 pragma 233  
 precedence of operators 17, 52, 95, 131–132, 200  
 prefix ++ and -- 46, 106  
 preprocessor, macro 88, 228–233  
 preprocessor name, `__FILE__` 254  
 preprocessor name, `__LINE__` 254  
 preprocessor names, predefined 233  
 preprocessor operator, `#` 90, 230  
 preprocessor operator, `##` 90, 230  
 preprocessor operator, `defined` 91, 232  
 primary expression 200  
 printf function 87  
 printf conversions, table of 154, 244  
 printf examples, table of 13, 154  
 printf library function 7, 11, 18, 153, 244  
 printing character 249  
 program arguments *see* command-line arguments  
 program, calculator 72, 74, 76, 158  
 program, cat 160, 162–163  
 program, character count 18  
 program, `dcl` 125  
 program, `echo` 115–116  
 program, file concatenation 160  
 program, file copy 16–17, 171, 173  
 program format 10, 19, 23, 40, 138, 191  
 program, `fsize` 181  
 program, keyword count 133  
 program, line count 19  
 program, list directory 179  
 program, longest-line 29, 32  
 program, lower case conversion 153  
 program, pattern finding 63, 69, 116, 117

## INDEX 269

ptrdiff\_t type name 103, 147, 206  
 push function 77  
 pushback, input 78  
 putchar library function 161, 247  
 putchar macro 176  
 putchar library function 15, 152, 161, 247  
 puts library function 164, 247  
  
 qsort function 87, 110, 120  
 qsort library function 253  
 qualifier, type 208, 211  
 quicksort 87, 110  
 quote character, ' 19, 37–38, 193  
 quote character, " 8, 20, 38, 194  
  
 \r carriage return character 38, 193  
 raise library function 255  
 rand function 46  
 rand library function 252  
 RAND\_MAX 252  
 read system call 170  
 readdir function 184  
 readlines function 109  
 realloc library function 252  
 recursion 86, 139, 141, 182, 202, 269  
 recursive-descent parser 123  
 redirection *see* input/output redirection  
 register, address of 210  
 register storage class specifier 83, 210  
 relational expression, numeric value of 42, 44  
 relational operators 16, 41, 206  
 removal of definition *see* `#undef`  
 remove library function 242  
 rename library function 242  
 reservation of storage 210  
 reserved words 36, 192  
 return from main 26, 164  
 return statement 25, 30, 70, 73, 225  
 return, type conversion by 73, 225  
 reverse function 62  
 reverse Polish notation 74  
 rewind library function 248  
 Richards, M. 1  
 right shift operator, `>>` 49, 206  
 Ritchie, D. M. xi  
  
 sbrk system call 187  
 scaling in pointer arithmetic 103, 198  
 scanf assignment suppression 157, 245  
 scanf conversions table of 158, 246

# header állományok

Hogyan tudunk létrehozni egy saját programkönyvtárat (library-t)?

```
1  #ifndef MYMATH_H
2  #define MYMATH_H
3
4  typedef struct {
5      int x;
6      int y;
7  } Pont;
8
9  int duplaz(int n);
10
11 #endif // MYMATH_H
12
```

mymath.h

```
1  #include "mymath.h"
2
3  int duplaz(int n)
4  {
5      return 2 * n;
6  }
7
```

mymath.c

```
1  #include "mymath.h"
2  #include <stdio.h>
3
4  int main()
5  {
6      int n = 3;
7      printf("%d kétszerese: %d\n", n, duplaz(n));
8
9      Pont p = { 1, 2 };
10
11      return 0;
12  }
13
```

main.c

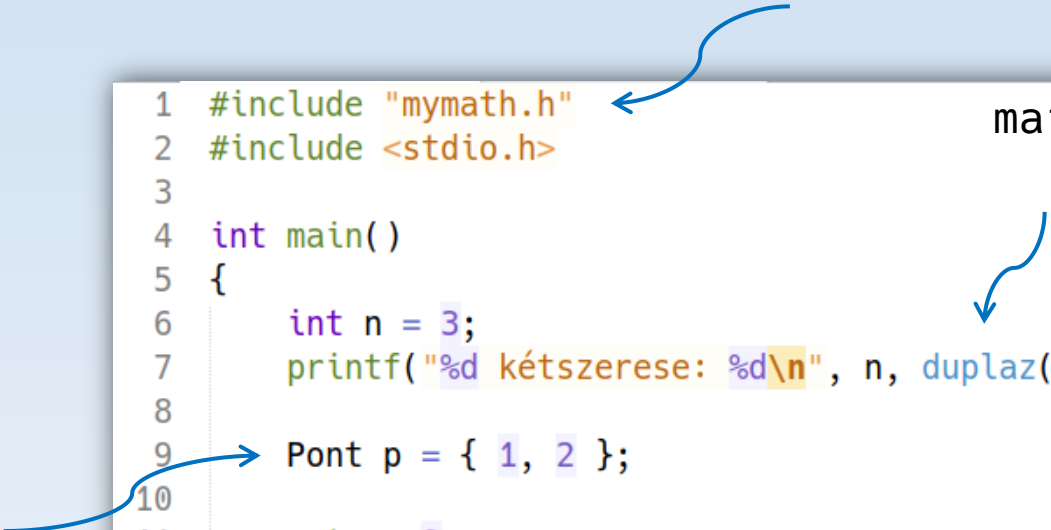
# header állományok

Hogyan tudunk létrehozni egy saját programkönyvtárat (library-t)?

Hivatkozunk a library-re, majd használjuk a benne található típusokat, függvényeket.

```
1  #include "mymath.h"
2  #include <stdio.h>
3
4  int main()
5  {
6      int n = 3;
7      printf("%d kétszerese: %d\n", n, duplaz(n));
8
9      Pont p = { 1, 2 };
10
11     return 0;
12 }
```

main.c





# header állományok

Hogyan tudunk létrehozni egy saját programkönyvtárat (library-t)?

```
1  #ifndef MYMATH_H
2  #define MYMATH_H
3
4  typedef struct {
5      int x;
6      int y;
7  } Pont;
8
9  int duplaz(int n);
10
11 #endif // MYMATH_H
12
```

mymath.h

A library-t két részre szokás osztani.

A header fájlba (.h kiterjesztésű) a függvények **deklarációja** kerül.

Saját típusokat, nevesített konstansokat is meg lehet itt adni többek között. Viszont a függvények implementációja NEM ide jön!

A header fájlt ún. *include guard*-dal kell ellátni (lásd 1., 2. és 11. sorok). Ez fogja azt biztosítani, hogy a header fájl tartalmát az előfeldolgozó csak egyszer fogja beilleszteni.

Az *include guard*-nál egy **egyedi** nevet kell megadni. Gyakori módszer még az is, hogy a nevet kiegészítik a dátummal, pl.: MYMATH\_H\_2020\_04\_26 . A lényeg, hogy a fordítás során ez ne ütközzön egy másik header fájlban használt névvel.

# header állományok

Hogyan tudunk létrehozni egy saját programkönyvtárat (library-t)?

```
1  #include "mymath.h"
2
3  int duplaz(int n)
4  {
5      return 2 * n;
6  }
7
```

mymath.c

A függvények **implementációját** egy .c kiterjesztésű fájlban adjuk meg. Ebben mindig érdemes hivatkozni a .h párjára (lásd 1. sor).

Egy ilyen állomány önállóan is fordítható, így le lehet ellenőrizni, hogy van-e benne hiba:

```
$ gcc -c mymath.c
```

Ez egy tárgykódot fog előállítani, ami közvetlenül még nem futtatható.

# header állományok

Hogyan tudunk létrehozni egy saját programkönyvtárat (library-t)?

```
1  #pragma once
2
3  typedef struct {
4      int x;
5      int y;
6  } Pont;
7
8  int duplaz(int n);
9
```

mymath.h

Láttuk, hogy az *include guard* hagyományosan 3 sort igényel.

Itt látható egy egyszerűbb verzió, ami szintén ugyanazt a szerepet látja el. Ezzel annyi a gond, hogy ez nincs szabványosítva. Ettől függetlenül a legtöbb modern fordító ezt is támogatja.

A hosszabb verzió mindig működik. Ez pedig nagy valószínűséggel működni fog.

# Házi feladat

- A K & R-féle „C Bibliában” nézzék át azokat a részeket, amikről szó volt az előadáson.
- Juhász István jegyzetéből nézzék át azokat a fogalmakat, amikről szó volt az előadáson ([link](#)).