CPROG

2. előadás (09.25.) az 1.-n nem volt semmi

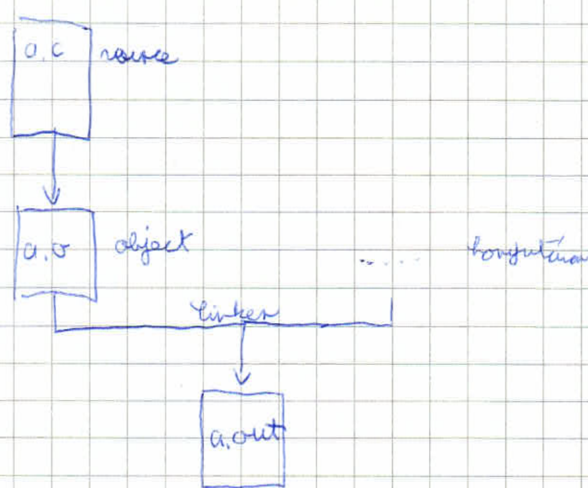
Forrásfájlt írunk (.c kiterjesztés) source file

Ezt fordítjuk a C fordítóval (pl.: g++) => Tárgykódot kapunk (.o mint object)

Miért?

a különböző hardvereknek különböző utasítás készletük van, ezeket tartalmazza a .o

De ez még nem futtatható, linkelés kell.



hw.c:

```c
#include <stdio.h>
int main()
{
    printf("Hello world\n");
}
```

#: preprocesszor direktíva

az include hatására a preprocesszor bemásolja az adott fájlt a kért helyre.

a header file tartalmazza kellő függvényeket

main: a program futása a main függvényt hajtja végre.

integer visszatérési értékű, ez a program exit-kódja:

0 ha rendben lefutott

1 ha hiba történt

(Általában a return 0-t elhagyhatjuk)

printf: kiír egy stringet a standard outputra

de honnan ismertük a printf függvényt? A object-fájlban csak be van linkelve, hogy felhasználunk egy valahol rögzített fgv-t, de majd a linker összeköti a helyével

Most már van egy forrás fájlunk, fordítsuk:

```
$ ls
  hw.c
$ gcc hw.c
```

← ez lefordítja a fájlt, ha talál stdio.h fájlt, ez lefut

a gcc -E kiírja a preprocesszálás eredményét

a gcc -S kiírja az elkészült assembly kódot hw.s néven (vagyis áll a fordító)

az assembly a gépi kód utasításait kódolja, csak nevekkel
⇒ az assembly nyelv hardver és oprendszer függő

a gcc -c hatására létrejön a hw.o fájl, ami gépi kódot tartalmaz, ez általunk megint dolgozható, de a hivatkozásai nem

Ha eredetileg több forráskóddal dolgoztunk, akkor itt kellett őket összeszerkeszteni a gcc hw.o paranccsal, mert a fordító feltételezi, hogy nem .c-re végződik

a végén létrejön az a.out (assembly output) ez már futtatható állomány

$ ./a.out   így tudjuk futtatni

a gcc -o hw.exe hw.c paranccsal tudunk egyéni nevet adni
ezt a $./hw.exe paranccsal futtatjuk

De mi van hiba esetén?

Ha hibás fgv-t adok meg, akkor a hw.o elkészül, mert akár létezhet akármilyen fgv, de a linker nem tudja összeszerkeszteni ⇒ linker hiba

Írjuk meg több fájlban:

```c
/* hw.c */
void print (char *)
int main ()
{
  print ("Hello");
  print ("world");
  return 0;
}
```

```c
/* p.c */
#include <stdio.h>
void print (char * par)
{
   printf ("%s\n", par);
}
```

A % egy string formátumot jelent: %s - string; %c - char; %d - szám; %f - float
ezek konkrét értékét a string után paraméterezzük

A hw.c elejére nem kell a deklaráció, mert lefordul, de figyelmeztet, ezért illik deklarálni.

Hogyan fordítsunk:

```
$ gcc hw.c p.c -o hw
```
← lefordítja a hw-t és a p-t is .o-ra, majd összeszerkeszti, és törli a .o-t

vagy:

```
$ gcc -c hw.c
$ gcc -c p.c
$ gcc hw.o p.o -o hw
```
← most külön-külön fordítottuk őket és utána szerkesztettük.
Ha utólag az egyiket módosítanánk, elég csak azt újrafordítani és összeszerkeszteni

Ez statikus szerkesztés volt. A másik a dinamikus szerkesztés

Ha nagy folyamatokban dolgozunk együtt és sok külső könyvtár van, akkor a kész fájl akár több MB is lehet.

⇒ Dinamikusan betölthető könyvtár (dll; so) A külső fgv-eket a futtatható állomány futása közben tölti be, nincs benne ragadva az exe-ben

C PROG

3. előadás (10.02.)

```c
#include <stdio.h>
int main()    /* hello prog */
{ printf("Hello");
  return 0;
}
```

preprocesszor utasítások

```
#include <file>      a file valahol a default mappában van
         "file"      a file a relatív útvonal szerint van valahol
```

`# gcc -I/home/gsd/..` kapcsoló a fájlokat máshol is keresi

`#define BUFSIZE 1024` ilyenkor a BUFSIZE minden előfordulását kicseréli a [illegible] 1024-re

konvenció: a makrók nevei nagybetűsek

`MAX(x,y)` `x>y ? x : y` függvény jellegű makró

a makrók használata kerülendő, mert [illegible]. léteznek makrók, de csak spec esetben. Mondjuk mert nincs túlterhelés, így nem tudunk típusfüggetlen max fgv-t írni

include guard (az ellen, hogy ne include-oljunk többször mit):

header file mindig kezdődjön így:

```c
#ifndef FILE_H
#define FILE_H
  ⋮
#endif F
```

az #if kifejezés ha igaz, akkor az őt követő kódrészlet lefut

ifndef: ha a FILE_H makró nincs definiálva, akkor fut le.

Kommentek

/* komment, komment, komment */

Tokenek

- azonosítók
- kulcsszavak
- literálok
- operátorok
- szeparátorok

Hőmérséklet átszámító program:

```c
#include <stdio.h>


int main ()
{
   int fahr;

   for ( fahr =-100 ; fahr <= 400 ; fahr +=10)
   {
       printf (" fahr =%d, \t cels = %d \n", fahr , 5/9 * (fahr -32) );
   }
   return 0;
}
```

A %d helyére beírja a string után írt paramétereket.
DE mivel a kifejezésben egész számok állnak, az 5/9 miatt minden eredmény 0 lesz

javítási elképzelés: → 5./9. * (fahr -32)
ez se lesz jó, mert a %d miatt a valódi eredmény csak az első 4 bájtját veszi ki és azt egészként értelmezi

→ `printf ("fahr = %d, \t cels = %f \n", fahr, 5./9. * (fahr -32));`

A konkrét számok makróban, mert később módosíthatnánk őket. Célszerűbb lenne az alábbi

```c
#include <stdio.h>
# define  LOWER  -100
# define  UPPER  400
# define  STEP  10
# define  F2C(x)  ((5./9.) * ((x)-32))
int main ()
{
   int fahr;
   for (fahr = LOWER; fahr <=UPPER; fahr += STEP)
   {
      printf ("...", fahr, F2C(fahr))
   }
   return 0;
}
```

Mivel a makrók nem praktikusan célszerűek, ezt konstansokkal:

```c
#include <stdio.h>
double fahr2cels (double f);

int main ()
{
   const int lower = -100;
   const int upper = 400;
   const int step = 10;
   for (int fahr = lower; fahr <= upper; fahr += step)
   {
      printf ("...", fahr, fahr2cels(fahr));
   }
   return 0;
}
double fahr2cels (double f)
{
  return 5./9.*(f-32);
}
```

C-ben a függvény deklarálásánál nem szükséges kiírni a paraméterlistát, ez azt jelenti nem érdekel mi az, de ha adok, akkor a fordító az átadott paramétert automatikusan konvertálja, mint várt.

Ha a deklarációban nem írtuk volna paraméterezését, akkor futási hiba jött volna.

C PROG

4. előadás (10.09)

operátorok

$y = a + b * c$ — itt a szorzást kell előbb elvégezni: precedencia

de akkor sem világos, hogy az azonos precedenciájúak milyen sorrend
pl.: $a*b/c*d$ — ez lehet: $((a*b)*\frac{1}{c})*d$
vagy $((a*b)*d)*\frac{1}{c}$

A mai nyelvekben az azonos precedenciájú műveleteket balról jobbra végzi
DE a C-ben nem csak az aritmetikai kifejezéseket lehet írni operátorokkal
pl.: `!*++*p` ennek zárójelezése: `(*(++(*p)))`

tehát ugyanazon jeleknek külön kontextusban más a asszociativitási szabály

Elsődleges operátorok

`t[i]` "tömb eleme" operátor

`r.m` "rekord mezője" operátor — mivel a . nagyobb, mint a *, ezért a `(*ptr).elethor` kifejezést így kell írni, erre egy külön jelölés: `ptr->elethor`

`sin(.5)` függvényhívás operátor

`i++; i--` postfix utasítások
(itt a kif értéke a változó régi értéke)

Postfix műveletek
`++i`
`--i` — itt a kif értéke a változó új értéke

`!a` negálás
`~a` bitenkénti negálás
`&a` memória cím lekérdező operátor
`*p` dereferencia operátor

„szorzási operátorok"
`a*b`
`a/b`
`a%b`
+ még ezek
`a+b`
`a-b`

Shift operátorok
`a<<b` `a>>b` bináris eltolás

Relációs műveletek
`a<b` `a<=b` `a>b` `a>=b`

`a==b` `a!=b`

`&`	bitenkénti és
`^`	bitenkénti ...
`|`	bitenkénti vagy
`&&`	logikai és
`||`	logikai vagy

(logikai és / logikai vagy: rövidzár)

a ? b : c

Értékadás

a = b — ez nem csak utasítás, hanem kifejezés is
van típusa és értéke

a bal oldalon olyan kifejezés állhat, ami egy írható tárterületet azonosít
pl.: változó, *ptr, de más is van

a += b   a /= b   a %= b

Vessző operátor

a, b — kiértékeli a-t, majd b-t, a kif értéke b lesz

```c
int f()
{
  printf("f");
  return 2;
}

int g()
{
  printf("g");
  return 1;
}

int h()
{
  printf("h");
  return 0;
}

int main()
{
    printf("%d", f() == g() == h());
}
```

Mit ír ki? Először f()-t és g()-t hasonlítja, mivel $2 \neq 1 \Rightarrow 0$
aztán $0 == h() \Rightarrow 1$

Ki fogja írni az 1-t, de előtte ki fog írni 3 betűt, és azt, hogy milyen sorrendben azt nem tudjuk, mert a kiértékelés sorrendje nincs benne a szabványban

Tipikus hiba:

```c
while (i < 10)
{
   t[i] = i++;
}
```

Ez nem értelmes, hogy mi lesz, fordítótól függ

CPROG

9. előadás (10.16.)

utasítás : pl.: - egy kifejezés után pontosvessző
a = b;
printf("Hello");

- if (kifejezés) utasítás1
[else utasítás 2]

pl.:
```
if (5 != printf("Hello"))
{
    printf("Meghalsz");
}
else
{
    printf("OK");
}
```

A kifejezésjelző zárójelek közötti részt utasításnak hívjuk, még akkor is, ha több sorból áll.

```
char conv(char ch)
{
    if ('a' == ch)
        ch = 'e';
    else if ('e' == ch)
        ch = 'i';
    else if ('i' == ch)
        ch = 'o';
    else if ('o' == ch)
        ch = 'u';
    else if ('u' == ch)
        ch = 'a';
    return ch;
}
```

Ez előző szöveges program másik verziója

```
char conv(char ch)
{
    switch (ch)
    {
        case 'a': ch = 'e'; break;
        case 'e': ch = 'i'; break;
        case 'i': ch = 'o'; break;
        case 'o': ch = 'u'; break;
        case 'u': ch = 'a'; break;
        default: /* nothing to do */ break;
```

ciklusok:

while (feltétel) utasítás

for (kif1, kif2, kif3) ut. $\Longleftrightarrow$ kif1;
while (kif2)
kif3;

```
char conv (char ch)
{
    char from[5] = {'a','e','i','o','u'};
    char to[5] = {'e','i','o','u','a'};
    int i;
    for (i=0; i<5; i++)
        if (from[i] == ch)
        {
            ch = to[i];
            break;
        }
    return ch;
}
```

A tömb elemét nem tudjuk kiírni, mert az első indexet igényelné
Trükk: sizeof(tomb)/sizeof(tomb[0])

assert(feltétel): ha a feltétel hamis, akkor leáll, és kiírja, hogy hol a baj

```
struct conv_t
{
    char from
    char to;
};
```

Ennek a fajta:

```
char conv (char ch)
{
    struct conv_t t[]{{'a','e'},{'e','i'},{'i','o'}...};
    int i;
    for(i=0; i<sizeof(t)/sizeof(t[i]); i++)
        if (ch == t[i].from)
            ch = t[i].to; break;
    return ch;
}
```

C PROG
6. előadás (11.06.)

<u>Élettartam, láthatóság</u>

Láthatóság : az azonosítókat mi adjuk, de különböző környezetben egyszer az
(scope)   azonosító jelentést vált

Ez egy statikus fogalom, amit a kódban vagy tudjuk vizsgálni, hogy
mi ismeri láthatóan

Élettartam : Egy memóriaterület lefoglalása meddig érvényes futás közben
(life)

<u>Scope:</u>

ismerünk globális és lokális változót
- globális, ami függvényen kívül van pl.:
  a deklaráció után az lehet érni
  statikus élettartam ⇒ implicit 0-ra inicializálja

```
int glob;

int main()
{
    glob = 1;
}
```

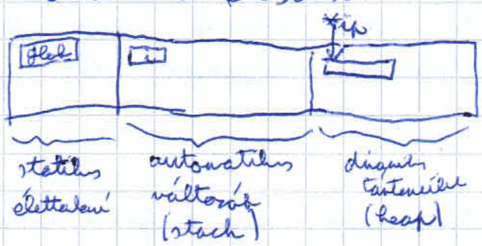
Mi van, ha több forrásfájlban akarjuk használni
ugyanazt a globális változót? nem szabad definiálni,
mert akkor több helyen lefoglalná. ⇒ extern deklaráció

```
pl. extern glob
int f()
{ int i;
  glob++;
}
```

Mi van, ha akarunk egy lokálisan használt, de statikus élettartamú változót
⇒ static kulcsszó

⇒ több forrásfájlban is létrehozhatók ugyanazzal az azonosítóval a statikus változók

A memóriaterület felosztása:



- Ha egy függvény belül használunk valamit, akkor a függvény meghívása után már nincs rá szükség és felszabadul
  memória fajtája lesz: stack
  A stackben a deklarált sorban a memóriaterület
  Adott függvényhíváshoz lefoglalt terület a stack-frame
  lokális statikus változó láthatósága lokális, de az élettartama statikus

```
int f()
{
  static int k;
}
```

Továbbá a lokális változó eltakarhatja a külső változót
az adott blokkban ilyenkor nem lehet elérni a külső változót

– ha a programozó önmaga akarja lefoglalni és felszabadítani a területet

```c
int* ip = (int*) malloc(10*sizeof(int));
```

A malloc visszatérési típusa void*, azt át kell konvertálni az adott típus pointerére

A lefoglalt területet fel kell szabadítani!

```c
free(ip);
```

példa:

kiír egy kérdést, a felhasználó válaszol, és a választ kiírjuk

```c
#include <stdio.h>
char* answer(char *q);

int main()
{
  printf("%s\n", answer("Hogy vagy? "));
  return 0;
}
```

naiv változat:

```c
char* answer(char *q)
{
   char buffer[80];
   printf("%s\n", q);
   fgets(buffer, 80, stdin);
   return buffer;
}
```

itt a függvény végén a buffer felszabadul, és egy ide mutató pointert adunk vissza

Ez nem lesz jó

javítás 1:

```c
char buffer[80];
char* answer(char *q)
{
   :
}
```

működik, de nem túl elegáns, mert a buffert csak az answer függvénynek kell, mégis globális

javítás 2:

```c
char* answer(char* q)
{
   static char buffer[80];
   :
}
```

ez már jobb, mert ugyan a név el van rejtve, de a tárterület megmarad.

Egy másik viszont nem férhet hozzá és a tárterület is foglalt.

+, ha egyszerre meghívom az answer függvényt, akkor az első válasz felülíródik

javítás 3:

```c
int main()
{
  char buffer[80];
  printf("%s", answer("Hogy vagy", buffer, 80));
}
char* answer(char *q, char *out, int len)
{
  printf("%s", q);
  fgets(out, len, stdin);
  return out;
}
```

Mivel a main fogja használni az answer fgv eredményét, ezért az a legtisztább, ha ő foglalja le a tárterületet

C PROG

7. előadás (11.13.)

Deklaráció

storage class	type	declarator
static	int	e[10]
		*ptr
		f()

tömb deklarálásnál ismerni kell a méretét globális és lokális statikus esetben
ha a stackbe kerül, akkor nem feltétlen

A deklarátorok kombinálhatók: `int tt[10][20];`
`tt[i][j]` — úgy viselkedik, mint egy több D-s tömb

másfajta: `int **pptr` — pointerre mutató pointer

`int *t[10]` — mivel a [] nagyobb prioritású, mint a *, ezért ez egy pointerekből álló tömb

függvényeknél meg kell különböztetni a deklarációt és a definíciót
dekl: megmondjuk, hogy a fordító hogy kezeljen egy nevet
def: megmondjuk, hogy egy dolog hogy működjön

```
int ipow(int a, int b)
{ int result = 1;
  for(int i=0; i<b; ++i) {
     result *= a;
  }
  return result;
}
```

ez egy definíció

adott helyen ez egy könyvtárban van, ha használni akarjuk, akkor deklarálni kell az adott helyen.

másikkal:

```
int ipow(int, int);

int k = ipow(2,4);
```

paraméter átadás MINDIG érték szerint
olyan, mintha lokális változói lennének, felülírhatóak, módosíthatóak stb..
a külvilág felé ez nem látszódik
ha meg akarjuk változtatni az értékét pointert kell átadnunk:

```
void incr(int x)
{
  ++x;
}
int main()
{
   int i = 0;
   incr(i);
   incr(i);
   printf("%i", i);   //0
}
```

```
void incr(int* ptr)
{
  ++*ptr;
}
int main()
{
   int i = 0;
   incr(&i);
   incr(&i);
   printf("%i", i);   //2
}
```

függvényre mutató pointerek: számológép program alapján, az exec-nek megadjuk hogy hogy melyik függvényt hajtsa végre!

```
void exec(double par, char ch)
{
  if ('s' == ch)    return sin(par);
  :
}
```

↙ ez a buta megoldás, mert mind függvényt le kell implementálni a szétválogatásban

Okos megoldás:

```
double (*func)(double)
```

// ez egy double visszatérési értékű, double bemenetű függvényre mutató ptr deklarálása.

ezek tömbje:

```
double (*ft[4])(double) = { sin, cos, tg, ctg };
```

meghívás:

```
(*ft[i])(double);
```

// az i. fgv-nek a-t adjuk értékül!

így az exec fgv:

```
void exec(double par, double (*f)(double))
{
  return (*f)(par);
}
```

Mi van, ha az értékátadás nem praktikus?

pl.:

```
char t[10000];
f(t);
```

↙ itt valójában egy 1. elemre mutató pointert adunk át
⇓
void f(char * tp) típusú

Mi van, ha több D-s a tömb?

```
int tt[10][20][30];
f2(tt);
```

hogy kell deklarálni?

hibás: void f2(int *)

biztonság: void f2(int tt[10][20][30])

majd a fordító megoldja, de mit jelent

az első elemre mutató pointer alakul.

tt első eleme egy 20 dbn 30 elemű tömböt tartalmazó tömb

void f2(int (*t)[20][30]) ⟺ (int tt[][20][30])

C PROG

8. előadás (11.20.)

typedef : létező típusokat nevezünk át. egyszerűsíti a életet

```
typedef long distance_t;
distance_t a = 1;
a *= 2;
```

A distance_t innentől longként viselkedik

praktikusabb megoldás:

pl.: `double (* fptr)(double) = sin;`

ha egy olyan fgv-t akarok, ami ilyet) vár, az nem bonyi. Célszerű lenne a fgv-re mutató pointert átnevezni:

```
typedef double (* trigfun_t)(double);
trigfun_t fptr2;
trigfun_t f(trigfun_t);
```

ez sokkal kényelmesebb

Összetett típusok:

- enum : névvel ellátott szimbolikus konstansokat hoz létre

pl.: `enum days {mon, tue, wed, thr, fri, sat, sun};`

```
    :
if (curr_day == mon)
    printf("Ma hétfő van.\n");
```

- struct : különböző típusú adatokat tárolunk együtt.

```
struct student
{
    int id;
    char name[50];
    char neptun[6];
};
```

Ezzel egy új típust definiálunk

```
struct student h1, h2;
h1 = h2;
```

mindent elvégezhető, amit típussal lehet, csak egyenlőségvizsgálatot nem.

hozzáférés adattaghoz:

```
struct student h3;
h3.id = 12;
h3.name[0] = 'A';
```

Hogyan hozunk létre neptunkódot hozzá? Kezdőértéknélkül nem lehet, az nem tömb!

```
void init_student(student_t *p)
{
    p->id = get_new_id();
    p->name = get_new_name();
    p->neptun = get_new_neptun();
}
```

// p->id == (*p).id   ez egy rövidítés

- union: hasonló a struct, de egyszerre a tagváltozóinak csak az egyikét tárolja

```
union member
{ int i;
  long l;
  double d;
};
```

Ugyanolyan helyfoglalású, de nem feltétlenül ugyanolyan jelentésű

CPROG

9. előadás (11.27.)

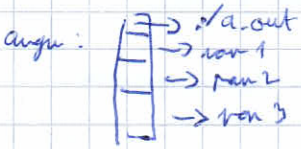
main függvény változatai

Lehet 1, 2 vagy 3 változatos

általában `int main (int argc, char *argv[])`
vagyis: `char **argv`

A miben meghívjuk egy programot, átadhatjuk változóként, egy tömbnek az argv-ban

pl.: `$ ./a.out var1 var2 var3`



vala lehet egy `char **envp` is, ami az environment adatait tartalmazza

Példa 1: Írassuk ki fordítva a kapott változatokat!

```
int main (int argc, char *argv[])
{
  int i = argc - 1;
  while (i > 0)
  {
    printf("%s\n", argv[i]);
    --i;
  }
  return 0;
}
```

Példa 2: A kapott fájlokról számoljuk meg a whitespace-ek számát

Lehetséges problémák: - nem tudjuk megnyitni a fájlt ⇒ írjuk ki!
- nem kapunk fájlt ⇒ ne hibát adjunk, hanem a stdin-ról olvassunk

```
#include <stdio.h>
#include <ctype.h>
int count(FILE *fp)
int main (int argc, char **argv)
{
  int summa = 0;
  int i = 1;
  while (i < argc) {
    FILE *fp = fopen (argv[i], "r");
    if (NULL == fp)
      fprintf (stderr, "Can't open file: %s\n", argv[i]);
    else {
      summa += count (fp);
      fclose(fp);
    }
    ++i;
  }
  return 0;
}
```

A elején kell, hogy
`if (argc == 1)`
`summa = count(stdin);`
és utána az else-ág

```
int count (FILE * fp)
{
    int cnt = 0;
    int ch;
    while (EOF != (ch = fgetc (fp)))
    {
        if (isspace (ch)) ++cnt;
    }
    return cnt;
}
```

FILE * fopen (char * fname, "r" | "r+" | "w" | "w+" | "a" | "a+") — "r": olvas; "w": ír új fölülír; "a": ír, de nem csonkol

int fclose (fp) — lezárja a fájlt

int fseek (FILE * fp, long off, int x) — a pozíciónált állítja
long ftell (FILE * fp) — megmondja hol vagyunk

int fgetc (FILE * fp) — karaktert olvas
int fputc (int ch, FILE * fp) — karaktert ír
int feof (fp) — eldönti, hogy EOF van-e

int fscanf (FILE * fp, char *, ...) — beolvas egy string formát-t
nem biztonságos, ezért sokan fscanf-s-t egyedül

int fgets (char * buffer, long c, FILE * fp) — a bufferbe olvas be max c karaktert
int puts (buffer, fp) — kiad

Eddig mindig formátum-konverzió volt. Mi van, ha nem szeretnénk?
int fread (char * buffer, int i, int s, FILE * fp) — fp-ből beolvas s db byte-ból i db-t.

pl.: studenteket akarok beolvasni tudom, egy student s byte, és i darabot akarok

int fwrite (ugyanaz) — az ellentettje

C PROG

10. előadás (12.05.)

Implementáljuk a grep utilityt-t:

$ grep minta file1 file2 — amelyik sorokban megtalálja a mintát, azt kiírja a képernyőre.

- i flaggel case insensitive legyen!
- v flaggel azokat a sorokat adja, ahol nincs találat
- s-ként tetszőleges kombinációt flagek pl.: -iv és -vi és -i -v is elmegy

1. lépés: main-beli ciklus:

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

int main(int argc, char *argv[])
{
  if (1==argc) usage(argv[0]);
  else if (2==argc) gr(stdin, stdout, argv[1]);
  else
  {
    for(int i=2; i<argc; ++i)
    {
      FILE *fp = fopen(argv[i], "r");
      if(fp)
      {
        gr(fp, stdout, argv[i]);
        fclose(fp);
      }
      else perror("Can't open file");
    }
  }
  return 0;
}
```

void usage(char*)
void gr(FILE*, FILE*, char*)
#define BUFSIZE 1024
#include <string.h>

2. lépés konkrét függvény

```
void usage(char *n)
{
  fprintf(stderr, "usage %s: minta [files] \n", n);
  exit(1);  // kilép a programból a megadott visszatérési értékkel.
}

void gr(FILE* in, FILE* out, char *pattern)
{
  char buffer[BUFSIZE];
  while(fgets(buffer, BUFSIZE, in))
  {
    if(NULL!=strstr(buffer, pattern))  // ez hasonló vagy ugyanaz.
    {
      fputs(buffer, out);
    }
  }
}
```

A 2-es változat.

Jobban szegsorolt:

3. lépés: írunk külön utility-t a flag-ek feldolgozására

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFSIZE 1024
typedef struct PARAM          // egy structban tároljuk el, hogy melyik
{ int iflag                   // paramétert kértük meg.
  int uflag
  char *p;
} param_t;
param_t params;
int procargs (int argc, char * argv[]);  // visszaadja az első nem paramétert
int main (int argc, char * argv[])
{
   int i = procargs (argc, argv);
   :
   :
}
int procargs (int argc, char * argv[])
{
   int i = 1;
   while (i < argc && '-' == * argv[i])
   {
     int j = 1;
     while ('\0' != argv[i][j])
     {
        switch (argv[i][j])
        {
          case 'i': params.iflag = 1; break;
          case 'u': params.uflag = 1; break;
          default: usage(argv);
        }
        ++j;
     }
     ++i;
   }
   if (i >= argc) usage (argv);   // ha nem adott meg mintát, akkor írok ki
   params.p = argv[i];   if (params.iflag)
   ++i;                  {
   return i;               char *s = (char *) malloc (strlen(params.p)+1);
}                          conv2upper (s, params.p);
                           params.p = s
```

4. lépés: A maint kommigáljuk

```
int main (int argc, char * argv[])
{
  int i = procargs (argc, argv);
  if (i >= argc) gr(stdin, stdout);
  else
  {
    for ( ; i < argc; ++i)
       :     // ez a rész u.a.
  }
  return 0;
}
```

5. feladat: A gr_t is bonyolítja

```
void gr(FILE *in, FILE *out)
{
  char buffer[BUFSIZE];
  while ( fgets(buffer, BUFSIZE, in))
  {
    if (params.iflag)
    {
      char upper[BUFSIZE];
      conv2upper(upper, buffer);
    }
    int m = 0;
    m = (strstr(params.iflag ? upper : buffer, params.p) != NULL);
    if (params.vflag) m = !m;
    if (m) fputs(buffer, out);
  }
}

void conv2upper(char *t, char *s)
{
  char *p = s;
  while ('\0' != *p)
  {
    *t++ = toupper(*p++);          // átalakítás a ctype.h-ban található
  }
}
```