# Programozás Alapjai

Dr. Gergely Tamás
Dr. Jász Judit

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2021

(v0901)

# Tartalom


1. Bemutatkozás
   - Kurzus információk
   - A SZTE és az informatikai képzés
2. Linux
   - Alapfogalmak
   - Linux parancsok
   - Linux shell
   - Felhasználók
   - Hálózat
3. Gyors C áttekintés
   - Bevezető
   - Pénzváltás (1. verzió)
   - Pénzváltás (2. verzió)
   - Röppálya számítás
   - Röppálya szimuláció
   - Az év napja
   - Csúszóátlag adott elemszámra
   - Csúszóátlag parancssorból
   - Basename standard inputról
   - Basename parancssorból
   - Tér legtávolabbi pontjai
   - A nappalis gyakorlat értékelése
4. Alapok
   - Alapfogalmak
   - A programozás fázisai
   - Algoritmus vezérlése
   - A C nyelvű program
   - Szintaxis
   - A C nyelv elemi adattípusai
   - A C nyelv utasításai
5. Vezérlési szerkezetek
   - Bevezetés
   - Szekvenciális vezérlés
   - Függvények
   - Szelekciós vezérlések
   - Ismétléses vezérlések 1.
   - Eljárásvezérlés
   - Ismétléses vezérlések 2.
6. Folyamatábra és struktúradiagram
7. Adatszerkezetek
   - Az adatkezelés szintjei
   - Elemi adattípusok
   - Pointer adattípus
   - Tömb adattípus
   - Sztringek
   - Pointerek és tömbök C-ben
   - Rekord adattípus
   - Függvény pointer
   - Halmaz adattípus
   - Flexibilis tömbök
   - Láncolt listák
   - Típusokról C-ben
8. IO
   - Alapok
   - Adatállományok
9. C fordítás
   - A fordítás folyamata
   - A preprocesszor
   - A C fordító
   - Assembler
   - Linker és modulok
10. Gyakorlati kérdések
    - Memóriahasználat
    - Gyakori C hibák
    - `where.c` felboncolva

# Tartalom

1 Bemutatkozás
- Kurzus információk
- A SZTE és az informatikai képzés

2 Linux
- Alapfogalmak
- Linux parancsok
- Linux shell
- Felhasználók
- Hálózat

3 Gyors C áttekintés
- Bevezető
- Pénzváltás (1. verzió)
- Pénzváltás (2. verzió)
- Röppálya számítás
- Röppálya szimuláció
- Az év napja
- Csúszóátlag adott elemszámra
- Csúszóátlag parancssorból
- Basename standard inputról
- Basename parancssorból
- Tér legtávolabbi pontjai
- A nappalis gyakorlat értékelése

4 Alapok
- Alapfogalmak
- A programozás fázisai
- Algoritmus vezérlése
- A C nyelvű program
- Szintaxis
- A C nyelv elemi adattípusai
- A C nyelv utasításai

5 Vezérlési szerkezetek
- Bevezetés
- Szekvenciális vezérlés
- Függvények
- Szelekciós vezérlések
- Ismétléses vezérlések 1.
- Eljárásvezérlés
- Ismétléses vezérlések 2.

6 Folyamatábra és struktúradiagram

7 Adatszerkezetek
- Az adatkezelés szintjei
- Elemi adattípusok
- Pointer adattípus
- Tömb adattípus
- Sztringek
- Pointerek és tömbök C-ben
- Rekord adattípus
- Függvény pointer
- Halmaz adattípus
- Flexibilis tömbök
- Láncolt listák
- Típusokról C-ben

8 IO
- Alapok
- Adatállományok

9 C fordítás
- **A fordítás folyamata**
- A preprocesszor
- A C fordító
- Assembler
- Linker és modulok

10 Gyakorlati kérdések
- Memóriahasználat
- Gyakori C hibák
- `where.c` felboncolva

# A C forrás fordításának folyamata

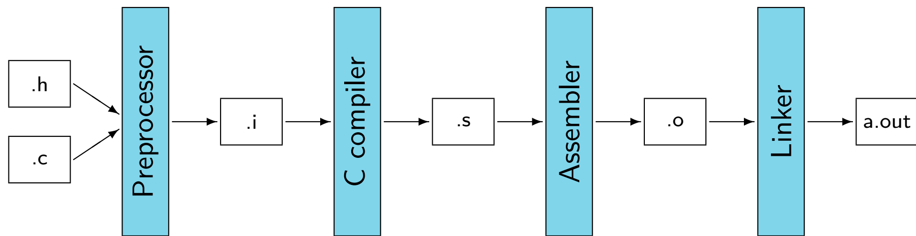
- A fordítási folyamat sok lépésből is állhat, de 4 olyan lépés van, ahol a folyamatot elkezdeni illetve befejezni lehet.
  - preprocessing – előfeldolgozás
  - compilation – fordítás (assembly nyelvre)
  - assembly – fordítás (gépi kódra)
  - linking – szerkesztés
- A fordítás közben többféle fájllal dolgozunk. A szabványos végződések:

| | |
|---|---|
| `file.c` | C source file – C forrásfájl |
| `file.h` | C header file – C fejléc fájl |
| `file.i` | preprocessed C file – előfeldolgozott C fájl |
| `file.s` | assembly source file – assembly nyelvű forrásfájl |
| `file.o` | object file – gépi kódú fájl |
| `a.out` | linked executable – szerkesztett futtatható fájl |

- A fájl végződése utal a programozási nyelvre és arra, hogy mit kell vele csinálni.

# A C forrás fordításának folyamata

## Egyszerűbb programok esetén




p.c
gcc -E p.c >p.i
p.i
gcc -S p.i
p.s
gcc -c p.s
p.o
gcc -o p p.o
p
gcc -S p.c
gcc -o p p.s
gcc -c p.c
gcc -c p.i
gcc -o p p.i
gcc -o p p.c

# Tartalom

1 Bemutatkozás
- Kurzus információk
- A SZTE és az informatikai képzés

2 Linux
- Alapfogalmak
- Linux parancsok
- Linux shell
- Felhasználók
- Hálózat

3 Gyors C áttekintés
- Bevezető
- Pénzváltás (1. verzió)
- Pénzváltás (2. verzió)
- Röppálya számítás
- Röppálya szimuláció
- Az év napja
- Csúszóátlag adott elemszámra
- Csúszóátlag parancssorból
- Basename standard inputról
- Basename parancssorból
- Tér legtávolabbi pontjai
- A nappalis gyakorlat értékelése

4 Alapok
- Alapfogalmak
- A programozás fázisai
- Algoritmus vezérlése
- A C nyelvű program
- Szintaxis
- A C nyelv elemi adattípusai
- A C nyelv utasításai

5 Vezérlési szerkezetek
- Bevezetés
- Szekvenciális vezérlés
- Függvények
- Szelekciós vezérlések
- Ismétléses vezérlések 1.
- Eljárásvezérlés
- Ismétléses vezérlések 2.

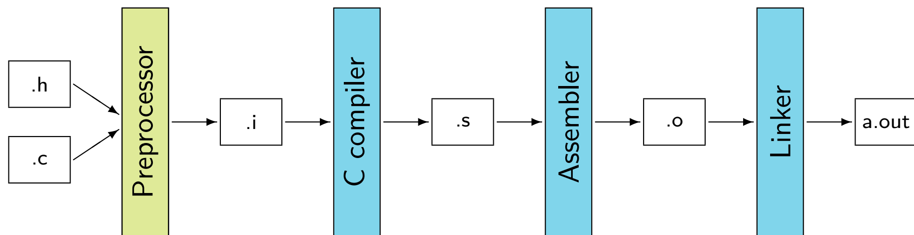
6 Folyamatábra és struktúradiagram

7 Adatszerkezetek
- Az adatkezelés szintjei
- Elemi adattípusok
- Pointer adattípus
- Tömb adattípus
- Sztringek
- Pointerek és tömbök C-ben
- Rekord adattípus
- Függvény pointer
- Halmaz adattípus
- Flexibilis tömbök
- Láncolt listák
- Típusokról C-ben

8 IO
- Alapok
- Adatállományok

9 C fordítás
- A fordítás folyamata
- A preprocesszor
- A C fordító
- Assembler
- Linker és modulok

10 Gyakorlati kérdések
- Memóriahasználat
- Gyakori C hibák
- `where.c` felboncolva




p.c — `gcc -E p.c >p.i` → p.i — `gcc -S p.i` → p.s — `gcc -c p.s` → p.o — `gcc -o p p.o` → p

`gcc -S p.c` (p.c → p.s)

`gcc -o p p.s` (p.s → p)

`gcc -c p.c` (p.c → p.o)

`gcc -c p.i` (p.i → p.o)

`gcc -o p p.i` (p.i → p)

`gcc -o p p.c` (p.c → p)

# A C előfeldolgozó

- A C előfeldolgozó (preprocesszor) egy makroprocesszor, melyet a C fordítóprogram automatikusan meghív, mielőtt a tényleges fordításhoz hozzákezdene.
- Azért nevezzük makroprocesszornak, mert lehetőségünk van úgynevezett makrók definiálására, hogy a hosszú konstrukciókat lerövidíthessük.
- A C előfeldolgozó nincs tisztában a C nyelv szintaxisával, csupán egy sororientált szövegszerkesztőről van szó.
- Főbb feladatai:
  - Kódtisztítás.
  - Fájlok (általában header fájlok) bemásolása.
  - Makro behelyettesítés.
  - Feltételes fordítás.
  - Sor vezérlés.

# Kódtisztítás

- Eltünteti a kommenteket.
  - A /* és */ közötti részeket kicseréli egy darab szóközre.
  - A // utáni részt a sor végéig törli.
- Tördelt sorok összevonása.
  - Sorvégi \ karakter esetén ezt és a sorvége jelet törli, vagyis a két sort egy sorrá vonja össze.

# Az előfeldolgozónak szóló utasítások

- A program forráskódjában elhelyezhetünk az előfeldolgozónak szóló utasításokat.
  - Ezek az utasítások a # karakterrel kezdődnek.
  - Ezt nem szükséges az első pozícióra írni, de a # kell legyen az első értelmes (nem whitespace) karakter a sorban.
  - A # és a kulcsszó között szintén lehetnek whitespace karakterek.
- A preprocesszor ezeket az utasításokat még a C nyelvű fordítás előtt végrehajtja.
  - A C fordító tehát már egy „átdolgozott" forráskódot tartalmaz, amely szigorúan megfelel a C nyelv szintaxisának.

# Header fájlok

## #include

- Az előfordítónak az `#include` parancs segítségével tudjuk megmondani, hogy a forráskód adott helyére egy másik fájl tartalmát másolja be.
- Ilyenkor az előfordító megkeresi a megadott nevű fájlt, a tartalmát bemásolja a parancs helyére, majd a bemásolt fájl elejétől folytatja a feldolgozást.
    - Ilyen módon a bemásolt fájlok tartalma is feldolgozásra kerül, tehát például a bennük található újabb `#include` parancsok hatására újabb fájlok lesznek bemásolva.
- Az `#include` paranccsal leginkább *header* fájlokat szoktunk bemásolni. Ezek olyan fájlok, melyekben gyakran használt konstans- és típus definíciók, függvény- és esetleg változó deklarációk szerepelnek, melyek a program fordításához szükségesek. Az `#include` segítségével nem kell ezeket a definíciókat minden egyes forráshoz külön-külön kézzel bemásolni.

# Header fájlok

- Technikailag kétféle header fájlt lehet megkülönböztetni:
  - A fordítókörnyezethez, operációs rendszerhez, szabványos függvénykönyvtárakhoz tartozó header fájlok.

```
#include <header.h>
```

    - Az ilyen fájlok előre kijelölt helyeken találhatók (pl. `/usr/inlcude`), az előfeldolgozó először itt keresi őket. Ha nem találja, akkor a `-I` fordítási kapcsolóval megadott útvonalakat nézi végig.
  - A saját programrendszerünkhöz tartozó header fájlok, melyekben az általunk deklarált vagy definiált, de több helyen felhasznált programelemek találhatóak.

```
#include "header.h"
```

    - Az ilyen fájlokat elsősorban az aktuális könyvtárban, a forráskód mellett keresi a preprocesszor, és ha nem találja, akkor nézi végig a `-I` fordítási kapcsolóval megadott útvonalakat.

# Makrók

## `#define`

- A preprocesszor számára a `#define` parancs segítségével tudunk szöveg-behelyettesítési szabályokat (makrókat) definiálni.
  - Az előfeldolgozó ilyenkor egy adott szöveg összes további (a `#define` utáni) előfordulását kicseréli egy másik szövegre.
  - A preprocesszor nincs tisztában a C szintaktikájával, ténylegesen csak szövegbehelyettesítést végez.
- A fordítóprogram `-D` kapcsolójának segítségével parancssorból is adhatunk meg makrókat, ezek úgy viselkednek, mintha közvetlenül a program elején definiáltuk volna őket.
- Az `#undef` parancs segítségével lehetőség van a makrók (adott ponttól érvényes) törlésére is.
- Az „üres” makró is definiáltnak számít.

# Makrók

## Egyszerű és paraméteres makrók

- Az egyszerű makró egy szimpla szöveg behelyettesítés.
  - Ennek segítségével lehet például valódi konstansokat készíteni.

```
#define TOMB_MERET 1020
```

  - A fenti `#define` esetén például az előfordító a `TOMB_MERET` minden előfordulását az 1020-as számleírásra cseréli, így a C fordító már csak a sok 1020 értéket fogja látni.
- Paraméteres makróval bonyolultabb konstrukciókat is lehet készíteni.
  - A makró ilyenkor a függvényhíváshoz hasonlóan viselkedik, azaz más-más paraméterekkel meghívva más és más kódot kapunk. (De ez nem valódi függvényhívás, hanem még a fordítás előtt megtörténik!)

```
#define min(X,Y)  ((X)<(Y)?(X):(Y))
```

  - A fenti makró esetén például a `min(a,b)` forráskód-részlet `((a)<(b)?(a):(b))`-vel helyettesdítődik.

- A standard előre definiált makrókat az ANSI szabvány rögzíti.
- Léteznek előre definiált makrók, melyek az operációs rendszer, architektúra, fordítóprogram jellemzőire utalnak.

| | |
|---|---|
| `unix` | Minden UNIX rendszerben |
| `m88k` | Motorola 88000 processzornál |
| `sparc` | Sparc processzornál |
| `sun` | Minden SUN számítógépen |
| `svr4` | System V Release 4 szabványú UNIX operációs rendszerben |

- Ezek a makrók platformfüggő kódok esetén feltételes fordításnál lesznek hasznosak.

# Feltételes fordítás

- Itt is az *if* alapszó szolgál az egyszerű szelekció megvalósítására.
- Míg a C programban szereplő `if` utasítás a program végrehajtása közben érvényesül, addig az előfeldolgozónál használt `#if` még a fordítás előtt értékelődik ki és ennek megfelelően más és más forráskód kerül lefordításra.
- Miért használunk feltételes fordítást?
  - A géptől és az operációs rendszertől függően más-más forráskódra van szükségünk.
  - Ugyanazt a programot több célból is használni szeretnénk, pl. teszünk bele nyomkövető utasításoka, de a végső változatba már nem.
- A feltételes fordítás direktívái:
  - `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`

# Feltételes fordítás

- Egyszerű szelekció, ha a *kifejezés* igaz, a kód része lesz a programnak, ha hamis, akkor nem:

```
#if kifejezés
...
#endif /* kifejezés */
```

- Többszörös szelekciónak speciális kulcsszava van, és feltétel nélküli egyébként ág is alkalmazható.

```
#if kifejezés1
... /* első ág */
#elif kifejezés2
... /* második ág */
#else
... /* harmadik ág */
#endif
```

# Szinusz(x) kiszámítása, feltételes nyomkövetéssel [1/2]

## szinusz-dbg.c [1–27]

```
1  /* sin(x) közelítő értékének kiszámítása a beépített sin(x)
2   *   függvény alkalmazása nélkül.
3   * 1997. Október 25.  Dévényi Károly, devenyi@inf.u-szeged.hu
4   * 2014. Május 19. Gergely Tamás, gertom@inf.u-szeged.hu
5   */
6
7  #include <stdio.h>
8  #include <math.h>
9
10 #define EPSZ    1e-10                          /* a közelítés pontossága */
11
12 double szinusz(double x) {
13     double osszeg = 0.0;                      /* a végtelen sor kezdőösszege */
14     double tag    = x;                      /* a végtelen sor aktuális tagja */
15     double xx     = x * x;                                        /* sqr(x) */
16     int    j      = 2;                             /* a nevező kiszámításához */
17
18     do {                                                  /* ciklus kezdete */
19         osszeg += tag;
20 #if DEBUG > 0
21         fprintf(stderr, "[LOG] Tag=% 30.10lf Osszeg=% 30.10lf\n", tag, osszeg);
22 #endif
23         tag = -(tag * xx / j / (j + 1));                    /* következő tag */
24         j += 2;
25     } while (fabs(tag) >= EPSZ);                /* végfeltétel, ciklus vége */
26     return osszeg;
27 }
```

# Szinusz(x) kiszámítása, feltételes nyomkövetéssel [2/2]

szinusz-dbg.c [29–49]

```
29 int main() {
30     double x;                                          /* argumentum */
31
32     printf("Kérem␣sin(x)-hez␣az␣argumentumot\n");
33     scanf("%lg%*[^\n]", &x); getchar();         /* egész sort olvasunk */
34 #ifdef DEBUG
35     printf("sin(%8.5f)=␣%13.10f\n", x, szinusz(x));
36 #endif
37
38     double x_orig = x;                          /* eredeti argumentum */
39
40     while (x < -M_PI) {                               /* transzformálás */
41         x += 2 * M_PI;
42     }
43     while (M_PI < x) {
44         x -= 2 * M_PI;
45     }
46     printf("sin(%8.5f)=␣%13.10f\n", x_orig, szinusz(x));
47
48     return 0;
49 }
```

# Szinusz(x) kiszámítása, feltételes nyomkövetéssel

## Fordítás

- „Debug" verzió fordítása

```
$ gcc -Wall -DDEBUG=1 -o szinusz szinusz-dbg.c -lm
$ ./szinusz
Kérem sin(x)-hez az argumentumot
20
[LOG] Tag=                20.0000000000 Osszeg=                20.0000000000
...
...
[LOG] Tag=                -0.0000000003 Osszeg=                 0.9129452492
sin(20.00000)=  0.9129452492
[LOG] Tag=                 1.1504440785 Osszeg=                 1.1504440785
...
...
[LOG] Tag=                 0.0000000010 Osszeg=                 0.9129452507
sin(20.00000)=  0.9129452507
```

- „Release" verzió fordítása

```
$ gcc -Wall -o szinusz szinusz-dbg.c -lm
$ ./szinusz
Kérem sin(x)-hez az argumentumot
20
sin(20.00000)=  0.9129452507
```

# Sor vezérlés

- A C forráskód több helyről másolódhat össze.
  - A preprocesszor által elvégzett „összemásolás” esetén nyilván van tartva, hogy melyik sor honnan származik.
  - A preprocesszor „előtt” elvégzett másolás, vagy forráskód-generálás során a #line direktívával adhatjuk meg, hogy eredetileg honnan származik a kód.

```
#line sorszám
```

# Sor vezérlés

```
$ cat -n preproc.c
     1  #define N 30
     2
     3  #ifdef DEBUG
     4  #define STRING "Debug"
     5  #else
     6  #define STRING "Release"
     7  #endif
     8
     9  #line 200
    10  int main() {
    11      int unix;
    12      char tomb[N] = STRING;
    13      for (unix = N - 1; unix && tomb[unix]; --unix) {
    14          tomb[unix] = 0;
    15      }
    16      return 0;
    17  }
```

```
$ gcc preproc.c
preproc.c: In function ‘main’:
preproc.c:201:9: error: expected identifier or ‘(’ before numeric constant
preproc.c:203:15: error: lvalue required as left operand of assignment
preproc.c:203:44: error: lvalue required as decrement operand
```

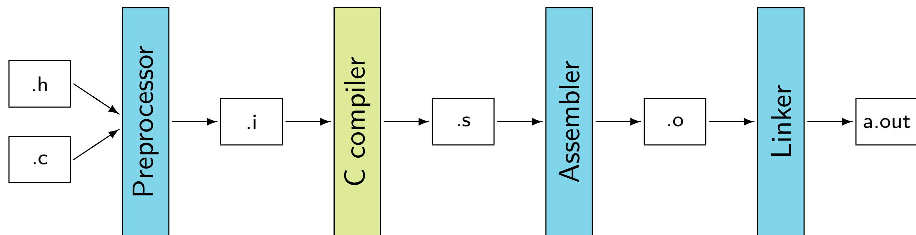
# Sor vezérlés

```
$ gcc -E preproc.c
# 1 "preproc.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "preproc.c"
# 200 "preproc.c"
int main() {
    int 1;
    char tomb[30] = "Release";
    for (1 = 30 - 1; 1 && tomb[1]; --1) {
        tomb[1] = 0;
    }
    return 0;
}
```

```
$ gcc -DDEBUG -E preproc.c
# 1 "preproc.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "preproc.c"
# 200 "preproc.c"
int main() {
    int 1;
    char tomb[30] = "Debug";
    for (1 = 30 - 1; 1 && tomb[1]; --1) {
        tomb[1] = 0;
    }
    return 0;
}
```

# Tartalom


1. Bemutatkozás
   - Kurzus információk
   - A SZTE és az informatikai képzés
2. Linux
   - Alapfogalmak
   - Linux parancsok
   - Linux shell
   - Felhasználók
   - Hálózat
3. Gyors C áttekintés
   - Bevezető
   - Pénzváltás (1. verzió)
   - Pénzváltás (2. verzió)
   - Röppálya számítás
   - Röppálya szimuláció
   - Az év napja
   - Csúszóátlag adott elemszámra
   - Csúszóátlag parancssorból
   - Basename standard inputról
   - Basename parancssorból
   - Tér legtávolabbi pontjai
   - A nappalis gyakorlat értékelése
4. Alapok
   - Alapfogalmak
   - A programozás fázisai
   - Algoritmus vezérlése
   - A C nyelvű program
   - Szintaxis
   - A C nyelv elemi adattípusai
   - A C nyelv utasításai
5. Vezérlési szerkezetek
   - Bevezetés
   - Szekvenciális vezérlés
   - Függvények
   - Szelekciós vezérlések
   - Ismétléses vezérlések 1.
   - Eljárásvezérlés
   - Ismétléses vezérlések 2.
6. Folyamatábra és struktúradiagram
7. Adatszerkezetek
   - Az adatkezelés szintjei
   - Elemi adattípusok
   - Pointer adattípus
   - Tömb adattípus
   - Sztringek
   - Pointerek és tömbök C-ben
   - Rekord adattípus
   - Függvény pointer
   - Halmaz adattípus
   - Flexibilis tömbök
   - Láncolt listák
   - Típusokról C-ben
8. IO
   - Alapok
   - Adatállományok
9. C fordítás
   - A fordítás folyamata
   - A preprocesszor
   - A C fordító
   - Assembler
   - Linker és modulok
10. Gyakorlati kérdések
    - Memóriahasználat
    - Gyakori C hibák
    - `where.c` felboncolva




p.c
p.i
p.s
p.o
p

```
gcc -E p.c >p.i
gcc -S p.i
gcc -c p.s
gcc -o p p.o
gcc -S p.c
gcc -o p p.s
gcc -c p.c
gcc -c p.i
gcc -o p p.i
gcc -o p p.c
```

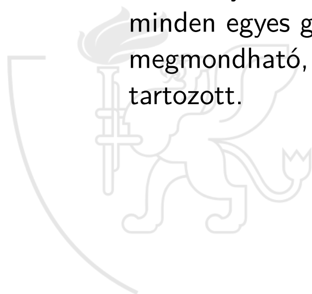
# A C fordító

- A C fordító tehát tulajdonképpen már egy preprocesszált fájlt kap, ami tisztán C nyelvi elemekből áll, és tartalmaz minden deklarációt ahhoz, hogy önmagában le lehessen fordítani.
  - A fordítás `gcc` esetében nem közvetlenül gépi kódra, hanem assembly nyelvre történik.
- A C fordító nem csak „szó szerint” tudja lefordítani a C programot, hanem különféle optimalizálások elvégzésére is képes.

# A C fordító

## Optimalizálás [1/3]

- A gcc fordítónak a `-O` kapcsolóval tudjuk megmondani, hogy milyen optimalizálásokat alkalmazzon.
  - `-O0` **Nincs optimalizálás:** A C forrás minden egyes művelete le lesz fordítva assemblyre, még akkor is, ha az nyilvánvalóan felesleges. Ez általában nagyon nagy és lassú gépi kódot eredményez, viszont hibakeresésnél igen hasznos, hiszen minden egyes gépi kódú utasításról egyértelműen megmondható, hogy az eredetileg melyik C-beli művelethez tartozott.



- 01 **Alapvető optimalizálás:** A C fordító elvégzi azokat az optimalizálásokat amik mind a méretet, mind a futási időt csökkentik, de a fordítási időt nem növelik meg jelentős mértékben.
- 02 **Optimalizálás méretre és futásidőre:** A C fordító elvégzi azokat az optimalizálásokat amik sem a méretre, sem a futásidőre nincsenek jelentős negatív hatással.
- 03 **Optimalizálás futásidőre:** A C fordító elvégzi azokat a futásidőre optimalizáló algoritmusokat is, amik jelentősen növelhetik a kód méretét.

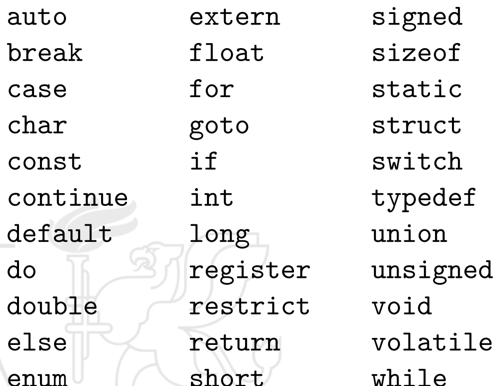
`-Ofast` **Aggresszív optimalizálás sebességre:** A C fordító az O3-as optimalizálásokon felül a standardoknak nem megfelelő átalakításokat is elvégez.

`-Os` **Méretre optimalizálás:** A C fordító az O2-es optimalizálások közül kihagyja azokat amelyek tipikusan méretet növelnek, de elvégez olyan plusz átalakításokat amik csökkentik a kód méretét, tekintet nélkül a futási időre gyakorolt hatásuktól.

`-Og` **Optimalizálás debuggolásra:** A C fordító azokat az átalakításokat végzi csak el, amikkel a program még szépen debuggolható marad.

# Alapszavak a C nyelvben

- Ismétlésképpen foglaljuk össze az alapszavakat:



| | | |
|---|---|---|
| `auto` | `extern` | `signed` |
| `break` | `float` | `sizeof` |
| `case` | `for` | `static` |
| `char` | `goto` | `struct` |
| `const` | `if` | `switch` |
| `continue` | `int` | `typedef` |
| `default` | `long` | `union` |
| `do` | `register` | `unsigned` |
| `double` | `restrict` | `void` |
| `else` | `return` | `volatile` |
| `enum` | `short` | `while` |

`char` Karakter típus, 1 bájt = 8 bit.

`int` Egész típus, 4 bájt = 32 bit (32 és valószínűleg 64 bites rendszereken is).

`float` Valós típus, 4 bájt = 32 bit.

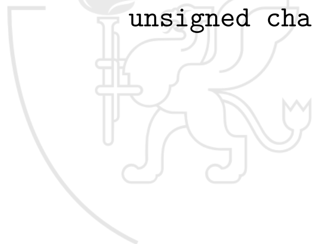
`double` Valós típus, 8 bájt = 64 bit.

`long` „hosszú” – a típus értéktartományának bővítése:
`long int` (4/8), `long long int` (8), `long double` (8/12)

`short` „rövid” – a típus értéktartományának szűkítése:
`short int` (2)

`signed` „előjeles” – negatív és pozitív értékek tárolása:
`signed char`, `signed ... int`

`unsigned` „előjeltelen” – csak nemnegatív értékek tárolása:
`unsigned char`, `unsigned ... int`



- `enum` Felsorolás típus: a típus értékhalmazát a programozó adja meg. A fordító a típust végső soron `int`-ként, az értékhalmaz elemeit konstans azonosítóként kezeli, de a program olvashatóságát jelentősen javíthatja a használata.
- `struct` Szorzat rekord típus: több akár különböző típusú érték egységben kezelése. A típus egyes mezői egyszerre tárolnak értékeket.
- `union` Egyesített rekord típus: több akár különböző típusú érték egységben kezelése. A típus egyes mezői közül egyidejűleg csak egy tárolhat értéket.

`typedef` Típusképzés kulcsszava: a deklarációban megadott azonosító nem egy adott típusú változót, hanem magát a típust fogja azonosítani.

`void` „Üres” típus: értékkel vissza nem térő függvények (eljárások) és típustalan pointerek megvalósításához.

`sizeof` Típus méretének lekérdezésére: az adott típus egy értéke hány bájton tárolódik.

`auto` „Automatikus”: A globálisan deklarált változóknak hely foglalódik a program teljes futási idejére, a blokkokban lokálisan deklarált változóknak pedig az adott blokk végrehajtásának idejére. A gobális változók valamelyik *adatszegmens*ben, a lokális változókat a *verem*ben lesznek tárolva.

`static` „Állandó”: Az ilyen lokálisan deklarált változónak mindenképpen „statikus”, vagyis állandó helyet foglalunk a program teljes futási idejére. A változó az adatszegmensben lesz elhelyezve.
Meg lehet jelölni globális programelemeket (változókat, függvényeket) is a `static` kulcsszóval, ezek az adott fordítási egységre nézve lokálisak lesznek, más egységből nem lesznek elérhetők még a szerkesztési lépésben (linker) sem.

# Alapszavak a C nyelvben

## Tárolási osztályok – hely

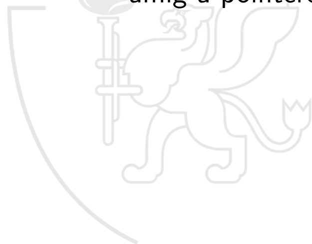
`register` „Regiszter”: A fordító megpróbálja mindenképpen regiszterben tartani a változót (ez nem biztos, hogy sikerül). Ez gyorsabb elérést biztosít, cserében a változónak nem lesz memóriacíme. (Akkor sem, ha a fordító nem tudja megoldani a végig regiszterben tárolást, és az értéket időnként elmenti a memóriába.)

`extern` „Külső”: Ezzel jelezzük a fordítónak, hogy a változó létezik, használni fogjuk, de ne foglaljon neki helyet, mert azt valahol máshol tesszük meg. A változóhivatkozások feloldása majd a linker feladata lesz. Ez akkor hasznos, ha egy változót más fordítási egységben is el szeretnénk érni. Érdekesség, hogy ha egy globális változót egy blokkban új deklarációval elfedünk, akkor egy ennek alárendelt blokkban `extern`-ként ismét újradeklarálva hozzáférhetünk a globális változóhoz.

`const` „Konstans” változó: A változó (beleértve a függvény paraméter) értékét a programban nem változtathatjuk meg. A fordítóprogram nem engedi, hogy (az inicializáláson kívül, ami konstans változónál kötelező, függvényparaméternél pedig automatikus) értéket adjunk a változónak, de a változó értéke továbbra is a memóriában tárolódik, így kerülővel, pointerek segítségével azért módosítható.

`volatile` „Illékony” változó: Ez a kulcsszó azt jelzi a fordítónak, hogy a változó értékét más is módosíthatja. Tehát a fordító nem számíthat a korábban regiszterbe töltött értékre (még ha maga a program nem is változtatott rajta). Ezért a fordító minden egyes műveletnél használni fogja a változóhoz rendelt memóriaterületet, azaz erre a változóra nem végez optimalizálást.

`restrict` „Egyedi” pointer (C99): Pointer típusú függvény paramétereknél jelezhetjük a fordító számára, hogy a pointer által mutatott memóriaterületet csak és kizárólag a megadott pointer segítségével lehet elérni. Ez segítheti a fordítót az optimalitálásban, hiszen ha pl. egyszer regiszterbe töltjük a pointer által mutatott értéket, akkor ez a regiszterben tárolt érték (újbóli betöltés nélkül) mindaddig újrafelhasználható, amíg a pointeren keresztül nem végzünk értékmódosítást.



`if` Az egyszerű szelekciós vezérlés kulcsszava.

`else` A szelekciós vezérlés egyébként ágának kulcsszava.

`switch` Az esetkiválasztásos szelekciós vezérlés kulcsszava.

`case` Az esetkiválasztásos szelekciós vezérlés blokkjában egy belépési pont kulcsszava.

`default` Az esetkiválasztásos szelekciós vezérlés blokkjában az egyébként fel nem sorolt esetek belépési pontja.

# Alapszavak a C nyelvben

## Vezérlés – ismétlés

`for` A ciklusszervezés egyik kulcsszava, általában számlálásos ismétlésekhez.

`while` A ciklusszervezés másik kulcsszava, általában kezdő- vagy végfeltételes ismétlésekhez.

`do` A végfeltételes ciklus kezdetét jelző kulcsszó.

`break` Kiugrás az adott ciklusból vagy `switch` utasításból, az azt követő első utasításra.

`continue` A ciklusmag aktuális végrehajtásának befejezése, ugrás az inkrementálásra vagy a feltétel-kiértékelésre.

`goto` A feltétel nélküli vezérlésátadás megvalósításának kulcsszava. (**Ne használjuk!**)

`return` Visszatérés a függvényből a hívás helyére, és a visszatérési érték megadása.

# Tartalom


1 Bemutatkozás
- Kurzus információk
- A SZTE és az informatikai képzés

2 Linux
- Alapfogalmak
- Linux parancsok
- Linux shell
- Felhasználók
- Hálózat

3 Gyors C áttekintés
- Bevezető
- Pénzváltás (1. verzió)
- Pénzváltás (2. verzió)
- Röppálya számítás
- Röppálya szimuláció
- Az év napja
- Csúszóátlag adott elemszámra
- Csúszóátlag parancssorból
- Basename standard inputról
- Basename parancssorból
- Tér legtávolabbi pontjai
- A nappalis gyakorlat értékelése

4 Alapok
- Alapfogalmak
- A programozás fázisai
- Algoritmus vezérlése
- A C nyelvű program
- Szintaxis
- A C nyelv elemi adattípusai
- A C nyelv utasításai

5 Vezérlési szerkezetek
- Bevezetés
- Szekvenciális vezérlés
- Függvények
- Szelekciós vezérlések
- Ismétléses vezérlések 1.
- Eljárásvezérlés
- Ismétléses vezérlések 2.

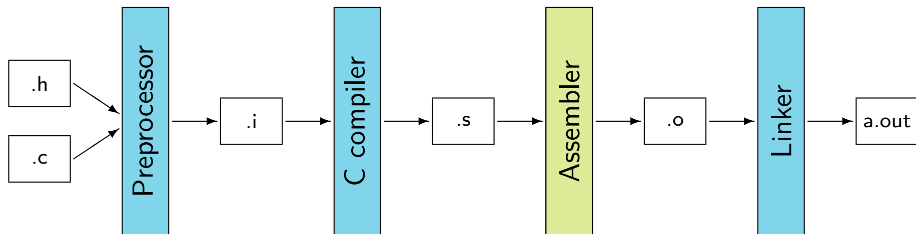
6 Folyamatábra és struktúradiagram

7 Adatszerkezetek
- Az adatkezelés szintjei
- Elemi adattípusok
- Pointer adattípus
- Tömb adattípus
- Sztringek
- Pointerek és tömbök C-ben
- Rekord adattípus
- Függvény pointer
- Halmaz adattípus
- Flexibilis tömbök
- Láncolt listák
- Típusokról C-ben

8 IO
- Alapok
- Adatállományok

9 C fordítás
- A fordítás folyamata
- A preprocesszor
- A C fordító
- **Assembler**
- Linker és modulok

10 Gyakorlati kérdések
- Memóriahasználat
- Gyakori C hibák
- `where.c` felboncolva




p.c → p.i → p.s → p.o → p

| Honnan | Hová | Parancs |
| --- | --- | --- |
| p.c | p.i | `gcc -E p.c >p.i` |
| p.i | p.s | `gcc -S p.i` |
| p.s | p.o | `gcc -c p.s` |
| p.o | p | `gcc -o p p.o` |
| p.c | p.s | `gcc -S p.c` |
| p.s | p | `gcc -o p p.s` |
| p.c | p.o | `gcc -c p.c` |
| p.i | p.o | `gcc -c p.i` |
| p.i | p | `gcc -o p p.i` |
| p.c | p | `gcc -o p p.c` |

# Az assembler

- A C fordító által generált assembly kódot fogja gépi kóddá fordítani, és egy úgynevezett *object* fájlt fog létrehozni.
- Egy ilyen object bináris formában tartalmazza az assembly kódból keletkezett gépi kódot, a programban található konstans értékeket (például a programban leírt számokat vagy sztringeket), illetve sok olyan technikai információt, amire szükség lesz ahhoz, hogy az object-et össze lehessen szerkeszteni más objectekkel (pl. szimbólumtábla).

# Tartalom

1 Bemutatkozás
- Kurzus információk
- A SZTE és az informatikai képzés

2 Linux
- Alapfogalmak
- Linux parancsok
- Linux shell
- Felhasználók
- Hálózat

3 Gyors C áttekintés
- Bevezető
- Pénzváltás (1. verzió)
- Pénzváltás (2. verzió)
- Röppálya számítás
- Röppálya szimuláció
- Az év napja
- Csúszóátlag adott elemszámra
- Csúszóátlag parancssorból
- Basename standard inputról
- Basename parancssorból
- Tér legtávolabbi pontjai
- A nappalis gyakorlat értékelése

4 Alapok
- Alapfogalmak
- A programozás fázisai
- Algoritmus vezérlése
- A C nyelvű program
- Szintaxis
- A C nyelv elemi adattípusai
- A C nyelv utasításai

5 Vezérlési szerkezetek
- Bevezetés
- Szekvenciális vezérlés
- Függvények
- Szelekciós vezérlések
- Ismétléses vezérlések 1.
- Eljárásvezérlés
- Ismétléses vezérlések 2.

6 Folyamatábra és struktúradiagram

7 Adatszerkezetek
- Az adatkezelés szintjei
- Elemi adattípusok
- Pointer adattípus
- Tömb adattípus
- Sztringek
- Pointerek és tömbök C-ben
- Rekord adattípus
- Függvény pointer
- Halmaz adattípus
- Flexibilis tömbök
- Láncolt listák
- Típusokról C-ben

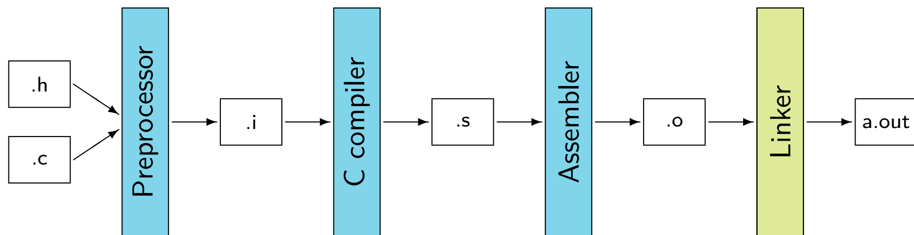
8 IO
- Alapok
- Adatállományok

9 C fordítás
- A fordítás folyamata
- A preprocesszor
- A C fordító
- Assembler
- **Linker és modulok**

10 Gyakorlati kérdések
- Memóriahasználat
- Gyakori C hibák
- `where.c` felboncolva

# A C forrás fordításának folyamata

## Linker




p.c — `gcc -E p.c >p.i` → p.i — `gcc -S p.i` → p.s — `gcc -c p.s` → p.o — `gcc -o p p.o` → p

- `gcc -S p.c` (p.c → p.s)
- `gcc -o p p.s` (p.s → p)
- `gcc -c p.c` (p.c → p.o)
- `gcc -c p.i` (p.i → p.o)
- `gcc -o p p.i` (p.i → p)
- `gcc -o p p.c` (p.c → p)

# A linker

- Mint látható, eddig a pontig minden C forrásnak „egyenes” az útja: a header fájlokat nem számolva egy C forrásból egy object keletkezik.
- Eddig a pontig kivétel nélkül bármelyik helyes C forrás eljuttatható. Futtatható program azonban csak olyan forrásból készíthető, amelyben van `main()` függvény.
- A linker feladata, hogy az object fájlból – amely csak a C forrásfájlban szereplő deklarációkat és definíciókat tartalmazza – előállítsa a futtatható programot.
- Első megközelítésben a linker olyan object fájlból tud futtatható programot csinálni, amiben van `main()` függvény.
- De miért kell ehhez linker? És mire jó a többi object (amiben nincs `main`)?
- Ahhoz, hogy ezt megértsük, meg kell ismerkednünk a modulok fogalmával.

# A modul fogalma

- A korszerű programozási nyelvek lehetővé teszik programok mellett programrendszerek nyelvi szinten történő kezelését is.
- A programrendszer nem más, mint modulok hierarchiája.
- A modul olyan programozási egység, amely programelemek (konstans, típus, változó, függvény) együtteséből áll.
- A modul két részre osztható:
  - *Interfészre*, ami a modul más modulok (programok) által is látható és felhasználható közösségi része, és
  - *Implementációra*, ami a modul privát, kívülről nem látható része.
- A modulnak e két részre osztása biztosítja a felhasznált modulok stabilitását és a privát elemek védelmét.

# A modulok megjelenési formái

- A modulok természetes egységei a fordításnak, tehát minden modul külön fordítható.
  - Fontos hangsúlyozni, hogy a *külön* fordítás nem *független* fordítást jelent.
- Következésképpen a modulok két formában léteznek:
  - Forrásnyelvi forma.
  - Lefordított forma.
- C-ben alapvető egységként adódik az egy C forrás/object, mint legkisebb lehetséges modul.
  - A `.o` kiterjesztésű object fájl ekkor a modul lefordított formája lesz,
  - a `.c` kiterjesztésű forrás a modul privát részének nyelvi formája,
  - a modul közösségi részét pedig a már említett `.h` kiterjesztésű header fájlok valósítják meg.

# Modulok C-ben

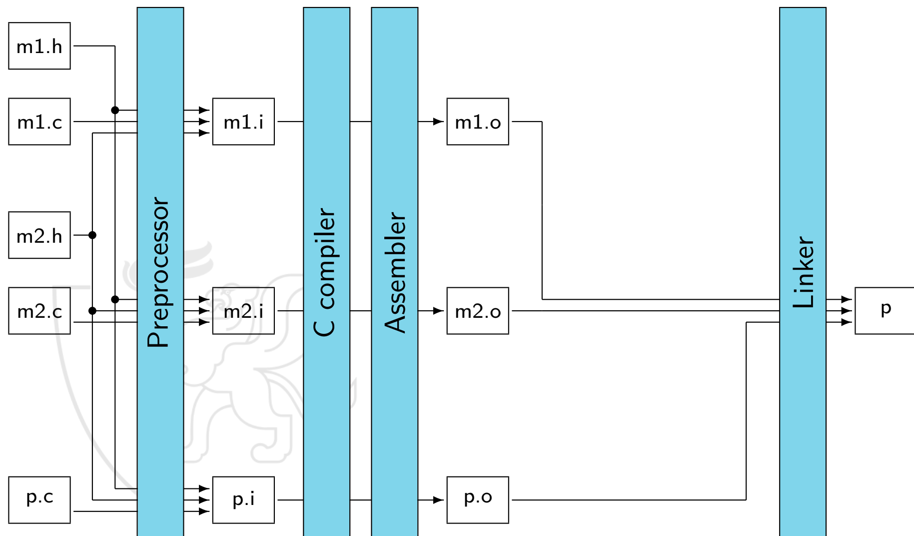
- A header és forrás fájlok tehát általában párosan léteznek:
  - A `.h` kiterjesztésű header fájl tartalmazza az összes olyan változó és függvény deklarációját, illetve konstans és típus definícióját, ami a modul interfészét, közösségi részét képezi. Ha valaki használni akarja a modult, csak include-olnia kell ezt a header fájlt, és máris használhatja az ebben deklarált dolgokat.
  - A `.c` kiterjesztésű forrás pedig tartalmazza a közösségi részben deklarált függvények definícióit, illetve a modul teljes privát részét, tehát az implementációt.
- Ahhoz, hogy olyan programot (vagy másik modult) írjunk, ami a mi modulunkat használja, elegendő a modulunk header fájlját ismerni, azaz a programban include-olni.
  - Az adott program forrásába a megfelelő `#include` helyére a preprocesszor bemásolja a mi header fájlunk tartalmát, így a program ismerni fogja az abban szereplő programelemeket, és object szintig lefordítható lesz.

# Linker

- Egy-egy programot vagy modult le tudunk fordítani object szintig, függetlenül attól, hogy milyen más modulokat használ.
- A `main()` nélküli objectekkel többet már nem nagyon tehetünk, maximum egy *archiver* segítségével össze tudjuk őket gyűjteni egy úgynevezett függvénykönyvtárban ami valójában egy olyan fájl, ami objecteket tartalmaz.
- A `main()` függvénnyel rendelkező objectekből viszont programok készíthetők. De egy-egy ilyen object önmagában eléggé hiányos lehet, hiszen a program a modul felhasznált függvényeinek csak a deklarációját ismeri, a függvény megvalósítása a modul object fájljában van.
- A linker feladata, hogy az ilyen objecteken átívelő hivatkozásokat feloldja, és egyetlen futtatható programot állítson elő sok `.o` fájlból.
- A sok `.o` közül pontosan egynek tartalmaznia kell a `main()` függvényt, hiszen az operációs rendszer majd ezt „hívja meg” a program indításakor.

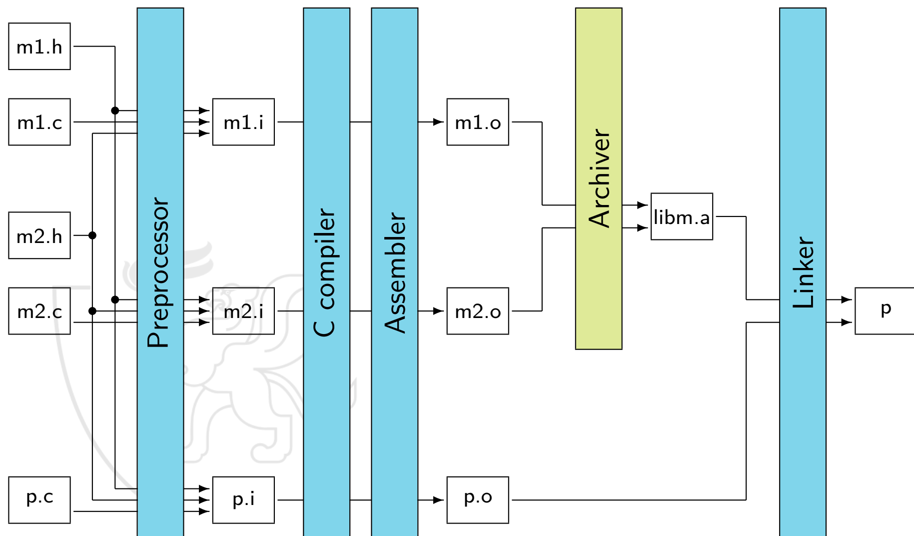
# A C forrás fordításának folyamata

## Komplexebb programok esetén



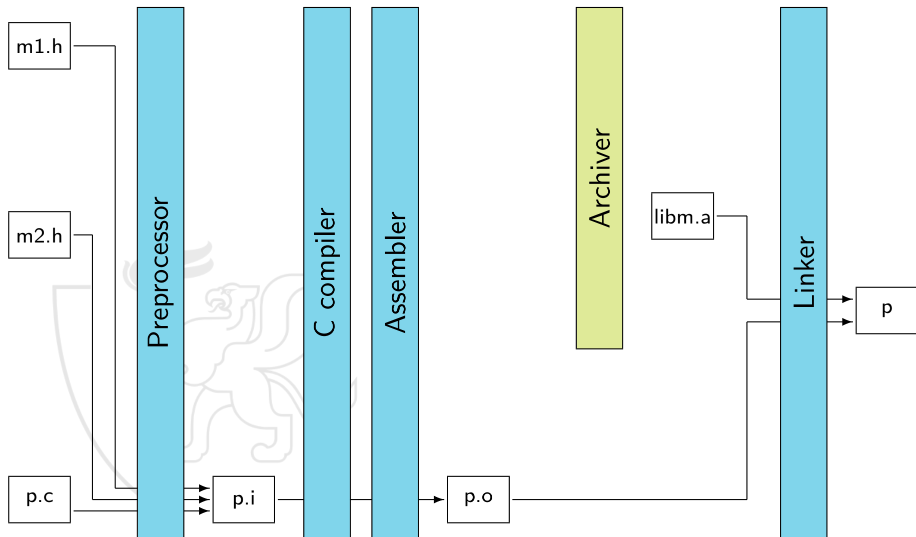
# A C forrás fordításának folyamata

## Komplexebb programok esetén



# A C forrás fordításának folyamata

## Komplexebb programok esetén



# Modulok C-ben

- Látható tehát, hogy egy-egy modul több programhoz is felhasználható, gondoljunk csak például a `math.h`-ra. Az ebben deklarált függvények (pl.: `sin()`) megvalósítása egy `libm.a` függvénykönyvtárban van tárolva.
- Az `#include <math.h>` „csak” a függvények deklarációit tartalmazza, amellyel már fordítható a forráskód, de önmagában még nem készíthető belőle futtatható program.
- A fordításkor megadott `-lm` kapcsoló mondja meg a linkernek, hogy a `libm.a` fájlban is keresgéljen függvények után.
- Meg kell jegyeznünk, hogy a modul kifejezés általában nem csupán egyetlen `.o` fájlt, hanem egy vagy akár több függvénykönyvtárat is takarhat.
- A függvénykönyvtárakból csak azok az eljárások kerülnek bele a programba, amiket valóban meg is hívunk.

# Header fájlok

- Hogyan szokás tehát C-ben modulokat írni?
- Nagyobb programrendszerekben gyakran előfordul, hogy ugyanazt a header fájlt (tranzitíven) többször is includoljuk ugyanabból a forrásból.
- Mivel a C nyelv nem engedi például típusok újradeklarálását, ez fordítási hibát okozhat.
- Ezért a header fájlok tartalmát feltételes fordítási direktívákkal szokás (kell) védeni.

# Modulok C-ben

## Forrásfájlok

### Modul:

**m.h**

```c
#ifndef M_H
#define M_H
int fuggveny(const char*);
#endif
```

**m.c**

```c
#include "m.h"
int fuggveny(const char *str) {
    int  cs;
    for (cs = 0; *str; ++str) {
        cs ^= *str;
    }
    return cs;
}
```

### Program:

**p.c**

```c
#include <stdio.h>
#include "m.h"
int main(int argc, char *argv[]) {
    printf("%d\n", fuggveny(argv[0]));
    return 0;
}
```

### Modul:

`m.i`

```
# 1 "m.c" 1
# 1 "m.h" 1
int fuggveny(const char*);
# 2 "m.c" 2
int fuggveny(const char *str) {
    int cs;
    for (cs = 0; *str; ++str) {
        cs ^= *str;
    }
    return cs;
}
```

### Program:

`p.i`

```
...
# 2 "p.c" 2
# 1 "m.h" 1
int fuggveny(const char*);
# 3 "p.c" 2
int main(int argc, char *argv[]) {
    printf("\%d\n", fuggveny(argv[0]));
    return 0;
}
```

# Prímszámok keresése
## Halmaz modullal

- Ezek után készítsük el a prímszámkereső programunkat úgy, hogy a halmaz kezelését egy külön forrásban valósítjuk meg.

# Halmaz modul interfész [1/2]

## halmaz.h [1–23]

```c
/* Header fájl a halmaz modulhoz. A logikai típust most a
 * preprocesszor segítségével definiáljuk.
 * 2006. Augusztus 16. Gergely Tamás, gertom@inf.u-szeged.hu
 * 2014. Március 31. Gergely Tamás, gertom@inf.u-szeged.hu
 */

#ifndef HALMAZ_H
#define HALMAZ_H

#include <stdbool.h>

#define K (8 * sizeof(kishalmaz_t))
#define HALMAZELEM_PRINT_FORMAT "lld"

typedef long long int kishalmaz_t;

typedef long long int halmazelem_t;

typedef struct {
        halmazelem_t  n;                /* Az univerzum a [0..N) */
        long long int m;                  /* A kishalmazok száma */
        kishalmaz_t  *tar;               /* A kishalmazok tömbje */
} halmaz_t;
```

# Halmaz modul interfész [2/2]

## halmaz.h [25–39]

```
25 halmaz_t letesit(halmazelem_t);
26 void megszuntet(halmaz_t);
27 void uresit(halmaz_t);
28 void bovit(halmaz_t, halmazelem_t);
29 void torol(halmaz_t, halmazelem_t);
30 bool eleme(halmaz_t, halmazelem_t);
31 void egyesites(halmaz_t, halmaz_t, halmaz_t);
32 void metszet(halmaz_t, halmaz_t, halmaz_t);
33 void kulonbseg(halmaz_t, halmaz_t, halmaz_t);
34 bool egyenlo(halmaz_t, halmaz_t);
35 bool resz(halmaz_t, halmaz_t);
36 bool ures_e(halmaz_t);
37 void ertekadas(halmaz_t, halmaz_t);
38
39 #endif /* HALMAZ_H */
```

# Halmaz modul implementáció [1/4]

## halmaz.c [1–25]

```
1  /* A halmaz modul függvényeinek megvalósítása.
2   * 2006. Augusztus 16. Gergely Tamás, gertom@inf.u-szeged.hu
3   */
4
5  #include <stdlib.h>
6  #include "halmaz.h"
7
8  halmaz_t letesit(halmazelem_t n) {
9      halmaz_t ret;
10     ret.n = n;
11     if (n) {
12         ret.m   = (n - 1) / K + 1;
13         ret.tar = (kishalmaz_t*)malloc(ret.m * sizeof(kishalmaz_t));
14     } else {
15         ret.m   = 0;
16         ret.tar = NULL;
17     }
18     return ret;
19 }
20
21 void megszuntet(halmaz_t h) {
22     free(h.tar);
23     h.n = h.m = 0;
24     h.tar = NULL;
25 }
```

# Halmaz modul implementáció [2/4]

## halmaz.c [27–49]

```c
27 void uresit(halmaz_t h) {
28     long long int i;
29     for (i = 0; i < h.m; ++i) {
30         h.tar[i] = 0;
31     }
32 }
33
34 void bovit(halmaz_t h, halmazelem_t x) {
35     if (x < h.n) {
36         h.tar[x / K] |= (1LL << (x % K));
37     }
38 }
39
40 void torol(halmaz_t h, halmazelem_t x) {
41     if (x < h.n) {
42         h.tar[x / K] &= ~(1LL << (x % K));
43     }
44 }
45
46 bool eleme(halmaz_t h, halmazelem_t x) {
47     return (x < h.n) &&
48            (h.tar[x / K] & (1LL << (x % K)));
49 }
```

# Halmaz modul implementáció [3/4]

## halmaz.c [51–70]

```
51 void egyesites(halmaz_t lhs, halmaz_t rhs, halmaz_t dst) {
52     long long int i;
53     for (i = 0; i < dst.m; ++i) {
54         dst.tar[i] = lhs.tar[i] | rhs.tar[i];
55     }
56 }
57
58 void metszet(halmaz_t lhs, halmaz_t rhs, halmaz_t dst) {
59     long long int i;
60     for (i = 0; i < dst.m; ++i) {
61         dst.tar[i] = lhs.tar[i] & rhs.tar[i];
62     }
63 }
64
65 void kulonbseg(halmaz_t lhs, halmaz_t rhs, halmaz_t dst) {
66     long long int i;
67     for (i = 0; i < dst.m; ++i) {
68         dst.tar[i] = lhs.tar[i] & ~(rhs.tar[i]);
69     }
70 }
```

# Halmaz modul implementáció [4/4]

## halmaz.c [72–95]

```
72 bool egyenlo(halmaz_t lhs, halmaz_t rhs) {
73     long long int i;
74     for (i = 0; (i < lhs.m) && (lhs.tar[i] == rhs.tar[i]); ++i);
75     return i == lhs.m;
76 }
77
78 bool resz(halmaz_t lhs, halmaz_t rhs) {
79     long long int i;
80     for (i = 0; (i < lhs.m) && !(lhs.tar[i] & ~(rhs.tar[i])); ++i);
81     return i == lhs.m;
82 }
83
84 bool ures_e(halmaz_t h) {
85     long long int i;
86     for (i = 0; (i < h.m) && !(h.tar[i]); ++i);
87     return i == h.m;
88 }
89
90 void ertekadas(halmaz_t dst, halmaz_t src) {
91     long long int i;
92     for (i = 0; i < dst.m; ++i) {
93         dst.tar[i] = src.tar[i];
94     }
95 }
```

# Prímszámok meghatározása a Halmaz modullal [1/2]

## prim3.c [1–21]

```c
/* Határozzuk meg az adott N természetes számnál nem nagyobb
 *   prímszámokat.
 * 1997. December 6.  Dévényi Károly, devenyi@inf.u-szeged.hu
 * 2006. Augusztus 16.  Gergely Tamás, gertom@inf.u-szeged.hu
 */

#include <stdio.h>
#include "halmaz.h"

#define N 16777216LL

int main() {
    halmaz_t szita;
    halmazelem_t p, s, i;
    long long int lepes, j;

    szita = letesit(N);
    bovit(szita, 2);                          /* a szita inicializálása */
    for (i = 3; i <= N; i += 2) {
        bovit(szita, i);
    }
```

# Prímszámok meghatározása a Halmaz modullal [2/2]

prim3.c [23–50]

```c
23     p = 3;                                          /* az első szitálandó prím */
24     while (p * p  <= N) {                   /* P többszöröseinek kiszitálása */
25         lepes = 2 * p;                                       /* lépésköz = 2*p */
26         s = p * p;                                     /* s az első többszörös */
27         while (s <= N) {
28             torol(szita, s);
29             s += lepes;
30         }
31         do {                                     /* a következő prím keresése */
32             p += 2;
33         } while ((p < N) && (! eleme(szita, p)));
34     }
35 
36     j = 0;                               /* a prímszámok kiíratása képernyőre */
37     printf("A␣prímszámok␣%lld-ig:\n", N);
38     for (p = 2; p < N; ++p) {
39         if (eleme(szita, p)) {
40             printf("%9" HALMAZELEM_PRINT_FORMAT, p);
41             if (++j == 8) {
42                 putchar('\n');
43                 j = 0;
44             }
45         }
46     }
47     putchar('\n');
48     megszuntet(szita);
49     return 0;
50 }
```

# Megvalósítás elrejtése

- Mire jók még a modulok?
- Programrendszer készítésekor az egyes modulokat célszerű úgy tervezni, hogy a header-ben ne legyenek láthatók azok a programelemek, amelyek csak a megvalósításhoz kellenek.
- Ennek két oka is van:
  - Egyrészt ezzel biztosítani tudjuk, hogy a modult felhasználó csak azokhoz a programelemekhez tud hozzáférni, amit a műveletek szabályos használata megenged, ezzel védettséget biztosítunk a lokális elemek számára.
  - Másrészt a megvalósításhoz használt elemek elrejtése az implementációs részbe azt eredményezi, hogy a megvalósítás módosítása, megváltoztatása esetén nem kell a programrendszer más modulját változtatni, pl. újrafordítani.

# Függvények megvalósításának elrejtése

- Nyilvánvalónak tűnik, hogy ha egy függvény deklarációja nem szerepel a modul header fájljában, akkor az a függvény más modulok által nem használható (közvetlenül nem hívható). Ez viszont így nem igaz.
- A linker nem tudja, honnan származik a deklaráció. Ha valaki saját magának deklarálja a függvényt, a linker akkor is megtalálja azt a modul object fájljában.
- Ha viszont egy függvényt a `static` kulcsszóval deklarálunk, akkor azt a linker nem fogja látni, hiába kerül bele az object fájlba. Vagyis, a headerben (`.h`) nem szereplő, kizárólag a megvalósításhoz tartozó függvényeket a modul megvalósításában (`.c`) ajánlott a `static` kulcsszóval deklarálni, hogy valóban el legyenek rejtve a „külvilág” elől.

# Adattípusok megvalósításának elrejtése

- A többféle programelem közül az adattípusok elrejtése okozhat még problémát.
  - Azoknak az eljárásoknak a deklarációjában ugyanis, amelyek a probléma megoldását adják – tehát a headerben kell lenniük – meg kell adni a paraméterek típusát, jóllehet a típus definícióját csak a modul `.c` forrásában kellene megadni, mert ezek megadása már a megvalósításhoz tartozik.
- Adattípus megadásának elhalasztása, vagyis elrejtése a `void*` pointer felhasználásával megoldható.
  - Ezt mutatja a következő séma, amelyben a `modh.h` a `modp.h` header `t` adattípusának elrejtését mutatja.
  - Mivel az elrejtést pointerek segítségével valósítjuk meg, a modulban lennie kell olyan eljárásnak, ami `t` típusú dinamikus változót létesít.

# Modulok C-ben

## Példa

Nyilvános:

- A `modp.h` header fájl (jól látható a `d` típus)

```
...
typedef d t;
void fgv(t*);
...
```

- A `modp.c` forrásfájl

```
...

void fgv(t *x) {
    ...

    (*x)
    ...
}
...
```

Rejtett:

- A `modh.h` header fájl (csak egy `void*` típus látszik)

```
...
typedef void *t;
void fgv(t);
...
```

- A `modh.c` forrásfájl

```
...
typedef d *tt;
void fgv(t x) {
    ...
    tt xx=x;
    (*xx)
    ...
}
...
```

# Megvalósítás elrejtése

- Az elrejtés technikája lehetővé teszi, hogy modul tervezésekor meg tudjuk adni az interfész végleges alakját, anélkül, hogy a megvalósítás bármely részletét is ismernénk.
- Ez különösen hasznos absztrakt adattípusok tervezése és megvalósítása esetén.
- Lássuk például a gráf adattípust, kétféle implementációval.

# Gráf interfész [1/1]

## graf.h [1–27]

```
1 /* Gráf megvalósítása típuselrejtéssel. Közös header fájl.
2  * 2006. Augusztus 17. Gergely Tamás, gertom.inf.u-szeged.hu
3  * 2017. Augusztus 31. Gergely Tamás, gertom.inf.u-szeged.hu
4  */
5
6 #ifndef GRAF_H
7 #define GRAF_H
8
9 #include <stdbool.h>
10
11 /* Típusdefiníciók */
12
13 typedef void *graf_t;
14 typedef int pont_t;
15
16 graf_t gr_letesit(int);
17 int  gr_pontok_szama(graf_t);
18 void gr_el_beszur(graf_t, pont_t, pont_t);
19 void gr_el_torol(graf_t, pont_t, pont_t);
20 bool gr_van_e_el(graf_t, pont_t, pont_t);
21 int  gr_pont_ki_fok(graf_t, pont_t);
22 int  gr_pont_ki_elek(graf_t, pont_t, pont_t[]);
23 int  gr_pont_be_fok(graf_t, pont_t);
24 int  gr_pont_be_elek(graf_t, pont_t, pont_t[]);
25 void gr_megszuntet(graf_t);
26
27 #endif /* GRAF_H */
```

# Gráf megvalósítás 1. verzió (mátrix) [1/6]

graf1.c [1–16]

```
/* Gráf megvalósítása típuselrejtéssel.
 * 2006. Augusztus 17. Gergely Tamás, gertom.inf.u-szeged.hu
 * 2017. Augusztus 31. Gergely Tamás, gertom.inf.u-szeged.hu
 */

#include <stdlib.h>
#include "graf.h"

typedef struct _graf_t {
    int    n;
    char *mx;
} _graf_t;

static bool jo_pont(_graf_t *g, pont_t p) {
    return 0 <= p && p < g->n;
}
```

# Gráf megvalósítás 1. verzió (mátrix) [2/6]

graf1.c [18–39]

```
18 graf_t gr_letesit(int n) {
19     _graf_t *ptr;
20     ptr = (_graf_t*)malloc(sizeof(_graf_t));
21     if (ptr) {
22         ptr->n = n;
23         ptr->mx = (char*)malloc(n * n * sizeof(char));
24         if (!ptr->mx) {
25             free(ptr);
26             ptr = NULL;
27         } else {
28             int i, nn;
29             for (i = 0, nn = n * n; i < nn; ++i) {
30                 ptr->mx[i] = 0;
31             }
32         }
33     }
34     return (void*)ptr;
35 }
36
37 int gr_pontok_szama(graf_t g) {
38     return ((_graf_t*)g)->n;
39 }
```

# Gráf megvalósítás 1. verzió (mátrix) [3/6]

graf1.c [41–60]

```
41 void gr_el_beszur(graf_t g, pont_t f, pont_t t) {
42     _graf_t *gg = (_graf_t*)g;
43     if (jo_pont(gg, f) && jo_pont(gg, t)) {
44         gg->mx[gg->n * f + t] = 1;
45     }
46 }
47
48 void gr_el_torol(graf_t g, pont_t f, pont_t t) {
49     _graf_t *gg = (_graf_t*)g;
50     if (jo_pont(gg, f) && jo_pont(gg, t)) {
51         gg->mx[gg->n * f + t] = 0;
52     }
53 }
54
55 bool gr_van_e_el(graf_t g, pont_t f, pont_t t) {
56     _graf_t *gg = (_graf_t*)g;
57     return jo_pont(gg, f) &&
58            jo_pont(gg, t) &&
59            gg->mx[gg->n * f + t];
60 }
```

# Gráf megvalósítás 1. verzió (mátrix) [4/6]

graf1.c [62–84]

```
62 int gr_pont_ki_fok(graf_t g, pont_t p) {
63     int i, fok = -1;
64     _graf_t *gg = (_graf_t*)g;
65     if (jo_pont(gg, p)) {
66         for (fok = i = 0; i < gg->n; ++i) {
67             fok += gg->mx[gg->n * p + i];
68         }
69     }
70     return fok;
71 }
72
73 int gr_pont_ki_elek(graf_t g, pont_t p, pont_t l[]) {
74     int i, fok = -1;
75     _graf_t *gg = (_graf_t*)g;
76     if (jo_pont(gg, p)) {
77         for (fok = i = 0; i < gg->n; ++i) {
78             if (gg->mx[gg->n * p + i]) {
79                 l[fok++] = i;
80             }
81         }
82     }
83     return fok;
84 }
```

# Gráf megvalósítás 1. verzió (mátrix) [5/6]

graf1.c [86–108]

```
86 int gr_pont_be_fok(graf_t g, pont_t p) {
87     int i, fok = -1;
88     _graf_t *gg = (_graf_t*)g;
89     if (jo_pont(gg, p)) {
90         for (fok = i = 0; i < gg->n; ++i) {
91             fok += gg->mx[gg->n * i + p];
92         }
93     }
94     return fok;
95 }
96
97 int gr_pont_be_elek(graf_t g, pont_t p, pont_t l[]) {
98     int i, fok = -1;
99     _graf_t *gg = (_graf_t*)g;
100    if (jo_pont(gg, p)) {
101        for (fok = i = 0; i < gg->n; ++i) {
102            if (gg->mx[gg->n * i + p]) {
103                l[fok++] = i;
104            }
105        }
106    }
107    return fok;
108 }
```

# Gráf megvalósítás 1. verzió (mátrix) [6/6]

graf1.c [110–116]

```
110 void gr_megszuntet(graf_t g) {
111     _graf_t *gg = (_graf_t*)g;
112     if (gg) {
113         free(gg->mx);
114     }
115     free(gg);
116 }
```

# Gráf megvalósítás 2. verzió (éllista) [1/6]

graf2.c [1–18]

```
 1 /* Gráf megvalósítása típuselrejtéssel.
 2  * 2006. Augusztus 17. Gergely Tamás, gertom.inf.u-szeged.hu
 3  * 2017. Augusztus 31. Gergely Tamás, gertom.inf.u-szeged.hu
 4  */
 5
 6 #include <stdlib.h>
 7 #include "graf.h"
 8
 9 typedef struct _graf_t {
10     int    n;
11     int   *be;
12     int   *ki;
13     pont_t *mx;
14 } _graf_t;
15
16 static bool jo_pont(_graf_t *g, pont_t p) {
17     return 0 <= p && p < g->n;
18 }
```

# Gráf megvalósítás 2. verzió (éllista) [2/6]

graf2.c [20–44]

```
20 graf_t gr_letesit(int n) {
21     _graf_t *ptr;
22     ptr = (_graf_t*)malloc(sizeof(_graf_t));
23     if (ptr) {
24         ptr->n =n;
25         ptr->be = (int*)malloc(n * sizeof(int));
26         ptr->ki = (int*)malloc(n * sizeof(int));
27         ptr->mx = (pont_t*)malloc(n * n * sizeof(pont_t));
28         if (!(ptr->mx && ptr->be && ptr->ki)) {
29             free(ptr->mx); free(ptr->be); free(ptr->ki);
30             free(ptr); ptr = NULL;
31         } else {
32             int i;
33             for (i = 0; i < n; ++i) {
34                 ptr->be[i] = 0;
35                 ptr->ki[i] = 0;
36             }
37         }
38     }
39     return (void*)ptr;
40 }
41
42 int gr_pontok_szama(graf_t g) {
43     return ((_graf_t*)g)->n;
44 }
```

# Gráf megvalósítás 2. verzió (éllista) [3/6]

graf2.c [46–67]

```
46 void gr_el_beszur(graf_t g, pont_t f, pont_t t) {
47     _graf_t *gg = (_graf_t*)g;
48     if (jo_pont(gg, f) && jo_pont(gg, t) && !gr_van_e_el(g, f, t)) {
49         gg->mx[gg->n * f + gg->ki[f]++] = t;
50         ++(gg->be[t]);
51     }
52 }
53
54 void gr_el_torol(graf_t g, pont_t f, pont_t t) {
55     int i;
56     _graf_t *gg = (_graf_t*)g;
57     if (jo_pont(gg, f) && jo_pont(gg, t)) {
58         for (i = 0; (i < gg->ki[f]) && (gg->mx[gg->n * f + i] != t); ++i);
59         if (i < gg->ki[f]) {
60             for (; (i < gg->ki[f]); ++i) {
61                 gg->mx[gg->n * f + i] = gg->mx[gg->n * f + i + 1];
62             }
63             --(gg->ki[f]);
64             --(gg->be[t]);
65         }
66     }
67 }
```

# Gráf megvalósítás 2. verzió (éllista) [4/6]

graf2.c [69–85]

```
69 bool gr_van_e_el(graf_t g, pont_t f, pont_t t) {
70     int i;
71     _graf_t *gg = (_graf_t*)g;
72     if (jo_pont(gg, f) && jo_pont(gg, t)) {
73         for (i = 0; (i < gg->ki[f]) && (gg->mx[gg->n * f + i] != t); ++i);
74         return i < gg->ki[f];
75     }
76     return false;
77 }
78 
79 int gr_pont_ki_fok(graf_t g, pont_t p) {
80     _graf_t *gg = (_graf_t*)g;
81     if (jo_pont(gg, p)) {
82         return gg->ki[p];
83     }
84     return -1;
85 }
```

# Gráf megvalósítás 2. verzió (éllista) [5/6]

graf2.c [87–105]

```
87 int gr_pont_ki_elek(graf_t g, pont_t p, pont_t l[]) {
88     int i, *ptr;
89     _graf_t *gg = (_graf_t*)g;
90     if (jo_pont(gg, p)) {
91         for (i = 0, ptr = gg->mx + p * gg->n + i; i < gg->ki[p]; ++i, ++ptr) {
92             l[i] = *ptr;
93         }
94         return gg->ki[p];
95     }
96     return -1;
97 }
98
99 int gr_pont_be_fok(graf_t g, pont_t p) {
100     _graf_t *gg = (_graf_t*)g;
101     if (jo_pont(gg, p)) {
102         return gg->be[p];
103     }
104     return -1;
105 }
```

# Gráf megvalósítás 2. verzió (éllista) [6/6]

graf2.c [107–127]

```
107 int gr_pont_be_elek(graf_t g, pont_t p, pont_t l[]) {
108     int i, *ptr;
109     _graf_t *gg = (_graf_t*)g;
110     if (jo_pont(gg, p)) {
111         for (i = 0, ptr = gg->mx + p * gg->n + i; i < gg->ki[p]; ++i, ++ptr) {
112             l[i] = *ptr;
113         }
114         return gg->be[p];
115     }
116     return -1;
117 }
118
119 void gr_megszuntet(graf_t g) {
120     _graf_t *gg = (_graf_t*)g;
121     if (gg) {
122         free(gg->mx);
123         free(gg->be);
124         free(gg->ki);
125     }
126     free(gg);
127 }
```

`-flto` **link-time optimizer** A külön fordított modulok hátránya, hogy bizonyos észszerű optimalizálásokat (mint például a *function inlining*) nem lehet fordítási időben elvégezni, ha azok több modult érintenek. A `-flto` kapcsoló hatására a linker utólag lefuttatja ezeket az optimalizáló algoritmusokat. A kapcsolót már **fordításkor is meg kell adni**, különben a fordító nem teszi bele a link-time optimalizáláshoz szükséges információt az object fájlokba.

# Linker

## Statikus és dinamikus linkelés

- Statikus linkelés esetén (ami egyes rendszereken a default, de a `-static` kapcsolóval kikényszeríthető) a linker minden szükséges függvényt belepakol a programba, hogy az önállóan futtatható legyen.
- Egyes operációs rendszereken lehetőség van dinamikus linkelésre is. Ekkor a könyvtári függvényeket az operációs rendszer rakja a programba *a program betöltésekor*. A különböző objectekben lévő saját függvényeket ilyenkor is a linker teszi bele a programba, de a könyvtári függvényeket nem, ezáltal kisebb lesz a program.
  - A dinamikus linkelés előnye, hogy ha egyszerre több program fut, ami ugyanazt a függvényt használja, a függvény kódja akkor is csak egyetlen példányban lesz meg a rendszer memóriájában, és a programok közösen használják ezt a kódot, osztoznak rajta (shared object (`.so`), dynamic link library (`.dll`)).