Programozás 2

2.

HALMAZ ADATTIPUS

Halmaz típus

- A programozásban számtalan formában előfordulhatnak halmazok és halmazokkal végzett műveletek.
- Legyen **`U`** egy egész típus, amit univerzumnak nevezünk. Tekintsük azt az értékhalmazt, amelynek elemei az **`U`** univerzum részhalmazai, ez lesz a **`Halmaz(U)`** új adattípus értékhalmaza.

Halmaz műveletei

- (A **`Halmaz(U)`** adattípusra a továbbiakban a **`Halmaz`** azonosítót használjuk.)
- Üresít(<- H : Halmaz);
  - A művelet végrehajtása után a **`H`** változó értéke az üres halmaz.
- Bővít(<-> H : Halmaz; -> x : U);
  - A művelet a **`H`** változó értékéhez hozzáveszi az **`x`** elemet.
- Töröl(<-> H : Halmaz; -> x : U);
  - A művelet a **`H`** változó értékéből törli az **`x`** elemet.

Halmaz műveletei

- Eleme(-> x : U; -> H : Halmaz) : bool;
  - A függvényművelet akkor és csak akkor ad igaz értéket, ha **x** eleme a **H** halmaznak.
- Egyesítés(-> H_1,H_2 : Halmaz; <- H : Halmaz);
  - A művelet eredményeként a **H** változó értéke a $\mathbf{H_1}$ és $\mathbf{H_2}$ halmaz egyesítése lesz.
- Metszet(-> H_1,H_2 : Halmaz; <- H : Halmaz);
  - A művelet eredményeként a **H** változó értéke a $\mathbf{H_1}$ és $\mathbf{H_2}$ halmaz közös része lesz.

Halmaz műveletei

- Különbség(-> H_1,H_2 : Halmaz; <- H : Halmaz);
  - A művelet eredményeként a **`H`** változó azokat és csak azokat az `x in U` értékeket tartalmazza, amelyre **`x in`** $\mathbf{H_1}$, de **`x`** nem eleme a $\mathbf{H_2}$ halmaznak.
- Egyenlő(-> H_1,H_2 : Halmaz) : bool;
  - Az egyenlőség relációs művelet.
- Rész(-> H_1,H_2 : Halmaz) : bool;
  - Akkor és csak akkor ad igaz értéket, ha a $\mathbf{H_1}$ halmaz minden eleme a $\mathbf{H_2}$ halmaznak is eleme.

Halmaz műveletei

- Ürese(-> H : Halmaz) : bool;
  - Akkor és csak akkor ad igaz értéket, ha **H** értéke az üres halmaz.
- Értékadás(<- H_1, -> H_2 : Halmaz);
  - A művelet hatására a $\mathbf{H_1}$ változó felveszi a $\mathbf{H_2}$ értékét.

Halmaz megvalósítása

- A C nyelvben a halmaznak nincs nyelvi megvalósítása.
- A halmazok reprezentálásához induljunk ki abból, hogy tetszőleges **U** univerzum esetén az **U** részhalmazai megadhatók karakterisztikus függvényükkel.
- Ha **H** az **U** részhalmaza, akkor karakterisztikus függvénye:

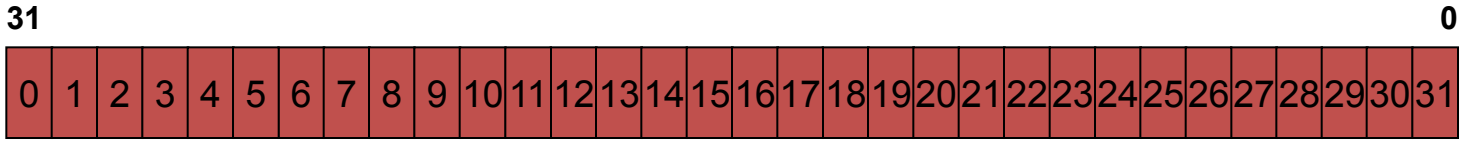
$$k_H : U \rightarrow \{0,1\},\ k_H(x)=1 \iff x \in H$$

Halmaz megvalósítása

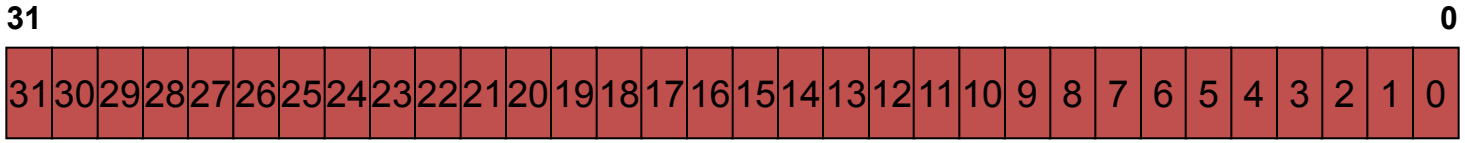
- A karakterisztikus függvények és **`U`** részhalmazai között kölcsönösen egyértelmű megfeleltetést ad az előbbi összefüggés.
- Ha **`U`** egész típus, a karakterisztikus függvényeket meg tudnánk valósítani a **`bool[U]`** típussal, de ez nem hatékony megoldás, mert **`U`** minden eleméhez a logikai **`bool`** típus megvalósításától függően legalább egy (de inkább 4) byte szükséges.
- Az 1 bit/elem hatékonyságot el is tudjuk viszont érni, a bitműveletek segítségével.

Halmaz megvalósítása

- A módszer lényege, hogy például egy **`int`** típusú változóban 32 bitet tárolhatunk, azaz egy 32 elemű kis halmaz reprezentálására ideális.



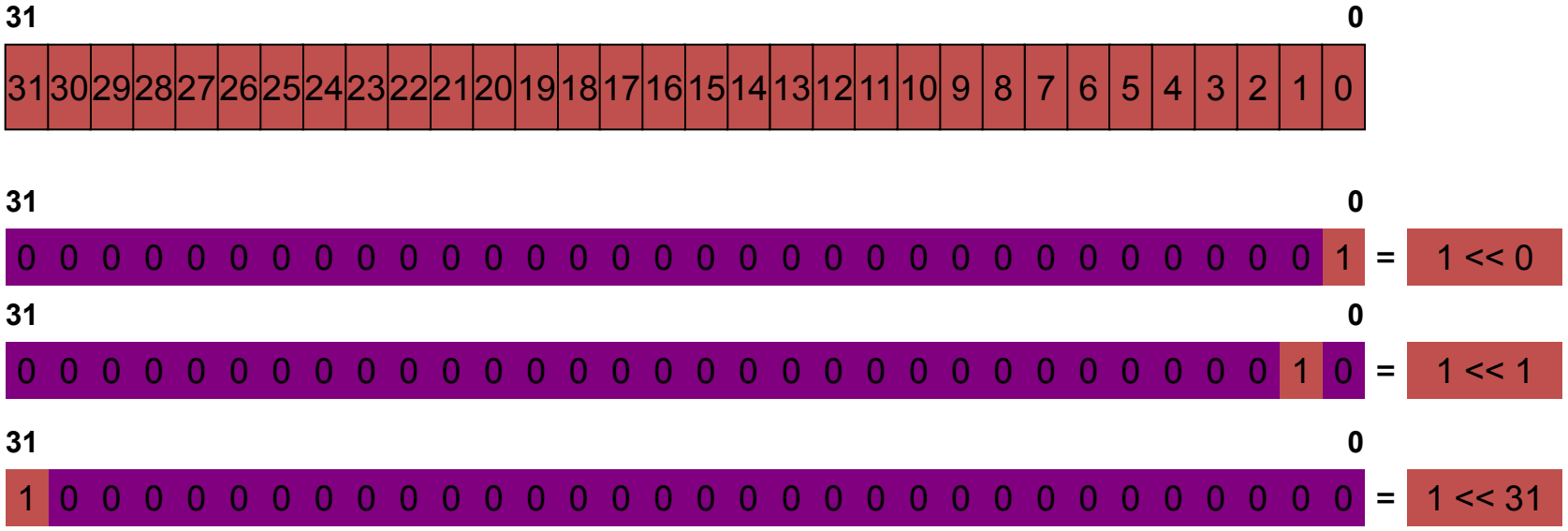
vagy inkább:



- Melyik jobb?

Halmaz megvalósítása

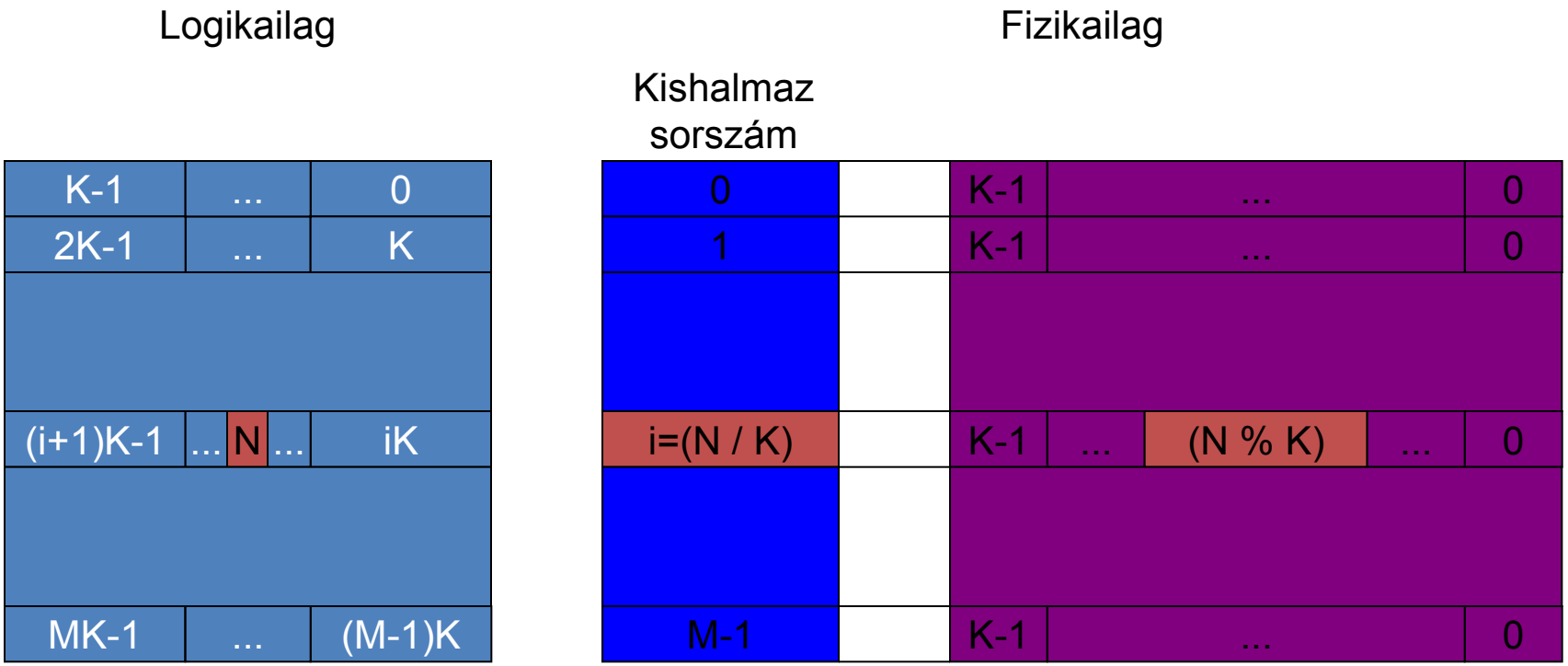
- Az egyes biteket a bitműveletekkel érhetjük el.



- A bit törlése, beállítása és lekérdezése az **&**, **|** és **~** műveletek segítségével történik.

Halmaz megvalósítása

- Nagyobb halmazt összerakhatunk kis halmazokból a következőképpen.



Halmaz megvalósítása

- Képezzük az alábbi típusokat:

```
#define K (8*sizeof(KisHalmaz))
#define M ???
#define H_MAX (M*K)

typedef int KisHalmaz;
typedef KisHalmaz Halmaz[M];
```

Halmaz megvalósítása

- Minden **x** természetes szám egyértelműen meghatározott az

  (x / K, x % K)

  számpárral.
- Tehát **x** akkor és csak akkor eleme a **H** változó által reprezentált halmaznak, ha teljesül, hogy a **H[x / K]** KisHalmaznak eleme az **(x % K)**, azaz a megfelelő KisHalmaz megfelelő bitje 1.
- Az univerzum halmaz ekkor a **0..H_MAX-1** intervallum lesz.

Halmaz műveletei

- Üresít(<- H : Halmaz);

```
void Uresit(Halmaz H)
{
  long int i;
  for(i = 0; i < M; ++i)
      H[i] = 0;
}
```

Halmaz műveletei

- Bővít(<-> H : Halmaz; -> x : U);

```
void Bovit(Halmaz H,
                        unsigned long
  int x)
{
   if(x < H_MAX)
     H[x / K] |=  (1 << (x % K));
}
```

Halmaz műveletei

- Töröl(<-> H : Halmaz; -> x : U);

```
void Torol(Halmaz H,
                        unsigned long
  int x)
{
   if(x < H_MAX)
     H[x / K] &= ~(1 << (x % K));
}
```

Halmaz műveletei

- Eleme(-> x : U; -> H : Halmaz) : bool;

```
int Eleme(unsigned long int x,
  Halmaz H)
{
   return (x < H_MAX) &&
          (H[x / K] & (1 << (x %
  K)));
}
```

Halmaz műveletei

- Egyesítés(-> H1,H2 : Halmaz; <- H : Halmaz);

```
void Egyesites(Halmaz H1, Halmaz H2,
                                  Halmaz H)
{
  long int i;
  for(i = 0; i < M; ++i)
      H[i] = H1[i] | H2[i];
}
```

Halmaz műveletei

- Metszet(-> H1,H2 : Halmaz; <- H : Halmaz);

```
void Metszet(Halmaz H1, Halmaz H2,
                                Halmaz H)
{
  long int i;
  for(i = 0; i < M; ++i)
      H[i] = H1[i] & H2[i];
}
```

Halmaz műveletei

- Különbség(-> H1,H2 : Halmaz; <- H : Halmaz);

```
void Kulonbseg(Halmaz H1, Halmaz H2,
                                  Halmaz H)
{
  long int i;
  for(i = 0; i < M; ++i)
      H[i] = H1[i] & ~(H2[i]);
}
```

Halmaz műveletei

- Egyenlő(-> H1,H2 : Halmaz) : bool;

```
int Egyenlo(Halmaz H1, Halmaz H2)
{
  long int i;
  for(i = 0;
      (i<M) && (H1[i]==H2[i]);
      ++i);
  return i==M;
}
```

Halmaz műveletei

- Rész(-> H1,H2 : Halmaz) : bool;

```
int Resz(Halmaz H1, Halmaz H2)
{
  long int i;
  for(i = 0;
      (i<M) && !(H1[i] & ~(H2[i]));
      ++i);
  return i==M;
}
```

Halmaz műveletei

- Ürese(-> H : Halmaz) : bool;

```
int Urese(Halmaz H)
{
  long int i;
  for(i = 0; (i<M) && !(H[i]);
 ++i);
  return i==M;
}
```

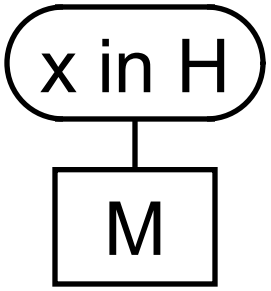
Halmaz műveletei

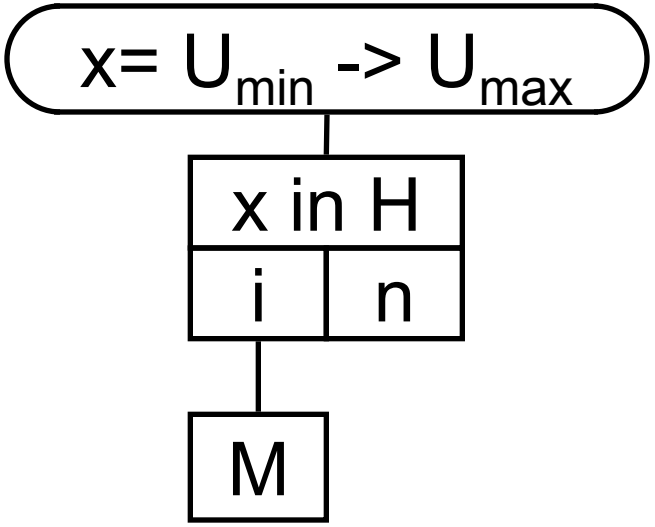
- Értékadás(<- H1, -> H2 : Halmaz);

```
void Ertekadas(Halmaz H1, Halmaz
  H2)
{
   long int i;
   for(i = 0; i < M; ++i)
        H1[i] = H2[i];
}
```

Halmaz típus

- Azáltal, hogy az univerzum egész típus, lehetővé válik a diszkrét ismétléses vezérlés megfogalmazása ezekre a halmazokra.
- Legyen U_{min}, U_{max} az univerzum minimális illetve maximális eleme.





Halmaz típus

- Ebben az esetben tehát a diszkrét ismétléses vezérlés megvalósítása C nyelven:

```
for(x=Umin; x<=Umax; ++x) {
     if(Eleme(x,H)) {
          M;
     }
}
```

Klikkek

- Problémafelvetés:
  - Egy közösségben az emberek közötti barátsági kapcsolat alapján meg kell határoznunk a klikkeket.
- Specifikáció:
  - Sorszámozzuk 1-től N-ig az embereket és olvassuk be az R barátsági relációt.
  - Input:
    - Számpárokat olvasunk be. Az input vége: 0 0. Az i,j pár azt jelenti, hogy az i. személy barátságban van a j. személlyel.
  - Output:
    - Soroljuk fel a baráti csoportokat.

Klikkek

- Algoritmustervezés:
  - A matematika nyelvén: Tegyük fel, hogy a barátság reflexív, szimmetrikus és tranzitív reláció. Meg kell határozni az R relációt tartalmazó legszűkebb ekvivalencia reláció szerinti osztályozást.
  - Induljunk ki abból, hogy minden személy csak saját magával van barátságban; tehát képezzük az {i} egyelemű halmazokat. Ha az input számpárokat valameddig feldolgozva meghatároztuk az osztályozást, a következő (i,j) számpárt beolvasva össze kell vonni azt a két részhalmazt, amelybe i illetve j tartozik, hisz mindenki, aki i-vel barátságban van, az barátságban van mindenkivel, akivel j barátságban van.

Klikkek

- Az algoritmus C megvalósítása tehát az absztrakt Halmaz(U) adattípus műveleteivel.

```
/* A beolvasott R relációt tartalmazó legszűkebb ekvivalencia
   reláció szerinti osztályozást határozzuk meg.
   Vázlat, nem fordítható C program.
   2006. Augusztus 14. Gergely Tamás, gertom@inf.u-szeged.hu
 */

#include <stdio.h>

#define N 10                                  /* maximális elemszám */

typedef Halmaz(U) Halmaz;                       /* EZ ÍGY NEM C !!!*/
```

>>>

Klikkek

```
int main() {
  Halmaz H[N];
  unsigned short i,j, ti,tj;

  for(i = 1; i <= N; ++i) {                    /* inicializálás */
      Uresit(H[i-1]); Bovit(H[i-1], i);
  }

  printf("Kérem a relációban lévő számpárokat!\n");
  scanf("%hd%hd%*[^\n]", &i, &j);
  getchar();
```

>>>

Klikkek

```
while(i != 0) {

  for(ti=0; !Eleme(i, H[ti]); ti++);
                          /* amelyik H[ti] halmazban van i ? */

  for(tj=0; !Eleme(j, H[tj]); tj++);
                          /* amelyik H[tj] halmazban van j ? */

  if(ti != tj) {                /* H[ti] és H[tj] összevonása */
    Egyesites(H[ti], H[tj], H[ti]);
    Uresit( H[tj] );
  }

  scanf("%hd%hd%*[^\n]", &i, &j);
  getchar();
}
```

>>>

Klikkek

```
  printf("Az osztályok: \n");
  for (i = 1; i <= N; i++) {          /* az osztályok kiíratása */
    if (! Urese(H[i - 1])) {
      for (j = 1; j <= N; j++) {
        if (Eleme(j, H[i - 1]))
          printf("%4hd,", j);
      }
      putchar('\n');
    }
  }
}
```

Klikkek

- Mivel N < 32, elég egy KisHalmaz.

```
/* A beolvasott R relációt tartalmazó legszűkebb ekvivalencia
   reláció szerinti osztályozást határozzuk meg.
   1997. December 6.  Dévényi Károly, devenyi@inf.u-szeged.hu
   2006. Augusztus 14. Gergely Tamás, gertom@inf.u-szeged.hu
 */

#include <stdio.h>

#define N  10                                /* maximális elemszám */

typedef int Halmaz;          /* N kicsi, ezért elegendő az int */
```

>>>

Klikkek

```
main(int argc, char *argv[])
{
  Halmaz H[N];
  unsigned short i, j, ti, tj;

  for (i = 1; i <= N; i++)                    /* inicializálás */
    H[i - 1] = 1 << i;

  printf("Kérem a relációban lévő számpárokat!\n");
  scanf("%hd%hd%*[^\n]", &i, &j);
  getchar();
```

>>>

Klikkek

```
while (i != 0) {                                          /* While */

  for(ti = 0; ((1 << i) & H[ti]) == 0; ti++); /* azon ti index
                    keresése, amelyik H[ti] halmazban van i */

  for(tj = 0; ((1 << j) & H[tj]) == 0; tj++); /* azon tj index
                    keresése, amelyik H[tj] halmazban van j */

  if (ti != tj) {              /* H[ti] és H[tj] összevonása */
    H[ti] |= H[tj];
    H[tj] = 0;
  }

  scanf("%hd%hd%*[^\n]", &i, &j);
  getchar();
}
```

>>>

Klikkek

```
  printf("Az osztályok: \n");
  for (i = 1; i <= N; i++) {          /* az osztályok kiíratása */
    if (H[i - 1] != 0) {
      for (j = 1; j <= N; j++) {
        if (j < 32 && ((1 << j) & H[i - 1]) != 0)
          printf("%4hd,", j);
      }
      putchar('\n');
    }
  }
}
```

Prímszámok

- Problémafelvetés:
  - Határozzuk meg az adott N természetes számnál nem nagyobb prímszámokat.
- Specifikáció:
  - Az N legyen konstans.
  - Input:
    - Nincs
  - Output:
    - Soroljuk fel a prímszámokat N-ig

Prímszámok

- Algoritmustervezés:
  - A jól ismert Erathosztenészi szita algoritmust valósítjuk meg.
  - A halmazt kezdetben feltöltjük az egynél nagyobb páratlan számokkal.
  - Megkeressük a halmaz még nem feldolgozott legkisebb elemét (ez prímszám lesz) és töröljük a többszöröseit.
  - Az előző pontot addig ismételjük, míg el nem érjük az N gyökét.
  - Az eredményhalmaz csak a prímszámokat fogja tartalmazni.

Prímszámok

```
/*
 * Határozzuk meg az adott N természetes számnál nem nagyobb
 * prímszámokat.
 * Készítette: Dévényi Károly, devenyi@inf.u-szeged.hu
 *             1997. December 6.
 * Módosította: Gergely Tamás, gertom@inf.u-szeged.hu
 *             2006. Augusztus 15.
 */
#include <stdio.h>

#define K (8*sizeof(KisHalmaz))
#define M 100
#define N (M*K)

typedef int KisHalmaz;
typedef KisHalmaz Halmaz[M];
```

Prímszámok

```
int main() {
    Halmaz szita;
    KisHalmaz kicsi;          /* a szita inicializálásához */
    long int p,s,lepes,i,j;

    kicsi = 0;                    /* a szita inicializálása */
    for(i = 0; i <= ((K-1) / 2); ++i) {
      kicsi |= (1 << (2*i+1));
    }
    for(i = 0; i < M; ++i) {
      szita[i] = kicsi;
    }
    szita[0] &= ~2;      /* 2 == (1 << 1) */
    szita[0] |= 4;       /* 4 == (1 << 2) */
```

>>>

Prímszámok

```
p = 3;                          /* az első szitálandó prím */
while(p*p  < N) {     /* P többszöröseinek kiszitálása */
    lepes = 2 * p;                      /* lépésköz = 2*p */
    s = p*p;                    /* s az első többszörös */
    while(s < N) {
        szita[s / K] &= ~(1 << (s % K));
        s += lepes;
    }
    do {                    /* a következő prím keresése */
        p += 2;
    } while( (p < N) &&
                    ! (szita[p / K] & (1 << (p % K))) );
}
```

Prímszámok

```
    j = 0;            /* a prímszámok kiíratása képernyőre */
    printf("A prímszámok %d-ig:\n", N);
    for(p = 2; p < N; ++p) {
        if(szita[p / K] & (1 << (p % K))) {
            printf("%8d", p);
            if(++j == 9) {
                j = 0;
                putchar('\n');
            }
        }
    }
    putchar('\n');
}
```

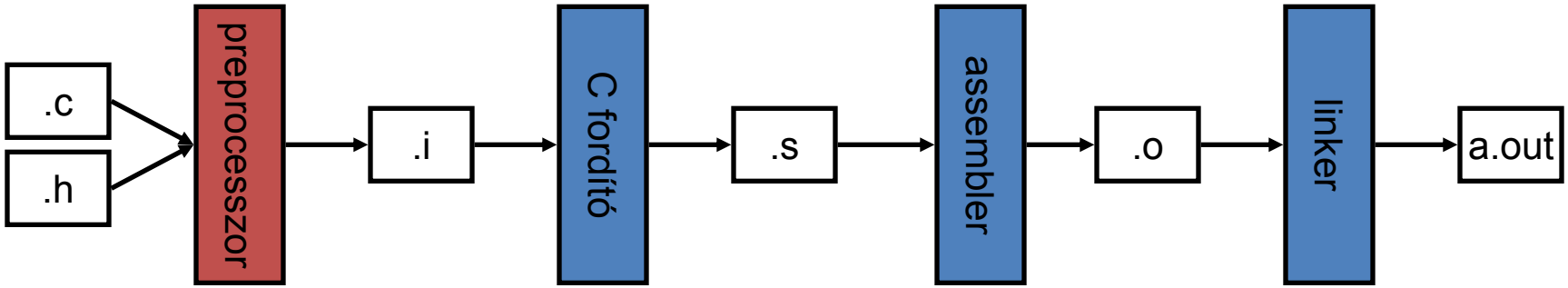
C PROGRAM FORDÍTÁSA

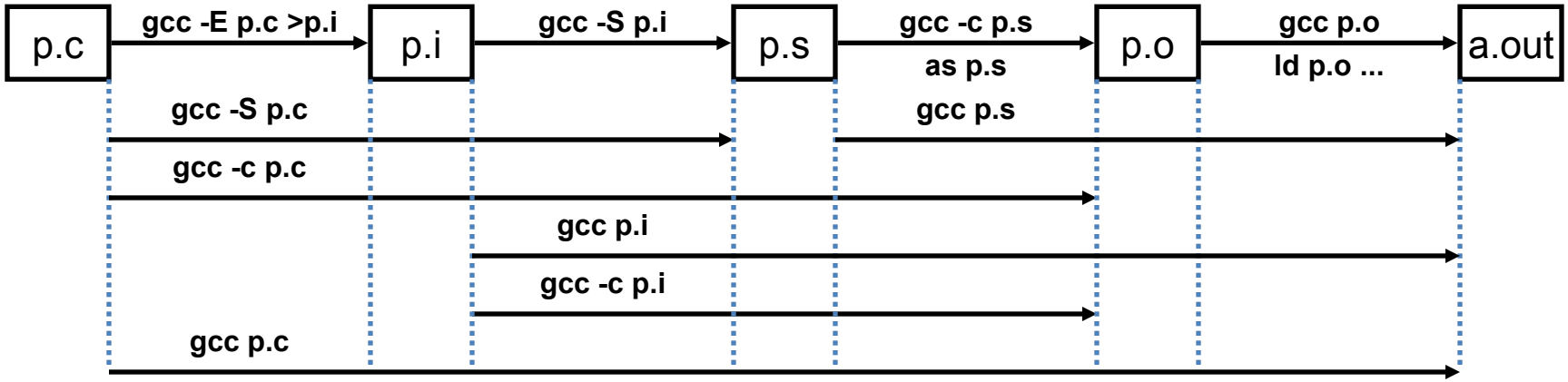
A C forrás fordításának folyamata

- 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 C forrás fordításának folyamata

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





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

- A C előfeldolgozó 4 egymástól független lehetőséget kinál:
  - Fájlok (általában header fájlok) bemásolása.
  - Makrobővítés. (Macro expansion)
  - Feltételes fordítás. (Conditional compilation)
  - Sor vezérlés. (Line control)

A C előfeldolgozó

- A C előfeldolgozó nincs tisztában a C nyelv szintaxisával, csupán egy sororientált szövegszerkesztésről van szó.
- Mindig elvégzi a következő egyszerű műveleteket:
  - Minden kommentet kicserél egy darab szóközre. (A `/*` és `*/` közötti részeket.)
  - `\újsor` karakterkombinációt (azaz sorvégi `\`-t) törli.
  - Az előre definiált makrókat behelyettesíti.

A C előfeldolgozó

- Az előfeldolgozó számára szóló utasítások **#**-kal kezdődnek. Nem szükséges az első pozícióra írni, de a **#** kell legyen az első értelmes karakter a sorban.

Header fájlok

- A header fájlok használata kettős:
  - A rendszer header fájlokban az operációs rendszerrel és a szabványos könyvtárakkal való kapcsolatot definiálják.

```
#include <fájl>
```

  - Ha nem találja a **`fájl`**-t a standard (előre definiált) helyen, akkor keresi a **`-I`** fordítási opcióval megadott könyvtárban.

Header fájlok

- A saját header fájlokban azok a deklarációk szerepelnek, amiket több forrásprogramunkban is fel szeretnénk használni.

```
#include "fájl"
```

- Ha nem találja a **`fájl`**-t az aktuális könyvtárban, akkor keresi a **`-I`** fordítási opcióval megadott könyvtárban.

Header fájlok

- Ha az előfeldolgozó talál egy neki szóló **`#include`** utasítást, akkor a megadott fájlt egyszerűen sorról sorra bemásolja az adott helyre, és a bemásolt rész elejéről folytatja a munkát.

Makrók

- Az egyszerű makró egy rövidítés, amit bevezetünk és a későbbiekben használunk.
- A C-ben a konstansokat például így definiáljuk:

```
#define TOMB_MERET 1020
```

- Mindenütt, ahol mostantól kezdve a **`TOMB_MERET`** megjelenik, az előfeldolgozó helyettesíti ezt az 1020-al, tehát a C fordító már a sok 1020 értéket fogja látni.

Makrók

- Az egyszerű makrók értékeit be tudjuk állítani a fordító **-D** opciójával is.
- Az egyszerű makrónál mindig ugyanazt a szöveget helyettesíti az előfeldolgozó.

Paraméteres makrók

- Ha a makrónak paraméterei is vannak, akkor a függvényhíváshoz hasonlóan más-más paraméterekkel is meghívhatjuk, így a helyettesítés is más és más lehet.

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

- Ha ezek után a programban szerepel a

```
min(a,b)
```

szöveg, akkor az előfeldolgozó behelyettesíti erre:

```
((a)<(b)?(a):(b))
```

Előre definiált makrók

- A standard előre definiált makrókat az ANSI szabvány rögzíti.
- Nem standard előre definiált makrók pl.
  - **`unix`** minden UNIX rendszerben
  - **`m88k`** Motorola 88000 processzornál
  - **`sparc`** Sparc processzornál
  - **`sun`** minden SUN számítógépen
  - **`svr4`** System V Release 4 szabványú UNIX operációs rendszerben
- A makró definiciót hatástalanítani lehet az **`#undef`**-fel.

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.

Feltételes fordítás

- 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. van benne nyomkövetés, míg a végső változatban már nincs.

Feltételes fordítás

- A feltételes fordítás direktivái:
  - **`#if`**, **`#ifdef`**, **`#ifndef`**
- Egyszerű szelekció:

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

- Egyszerű szelekció egyébként ággal:

```
#if kifejezés
... /* text-if-true */
#else
... /* text-if-false */
#endif
```

Feltételes fordítás

- Például:

```
#if defined(DEBUG)
printf("%d\n", v);
#endif
```

vagy:

```
#ifdef(DEBUG)
printf("%d\n", v);
#endif
```

esetleg

```
#if DEBUG > 2
printf("%d\n", v);
#endif
```

Feltételes fordítás

- Többszörös szelekció (egyébként ággal)

```
#if kifejezes1
... /* első ág */
#elif kifejezes2
... /* második ág */
#elif kifejezes3
... /* harmadik ág */
#else
... /* negyedik ág */
#endif
```

Sor vezérlés

- A C forráskód több helyekről másolodik össze. A **`#line`** direktivával megadhatjuk, hogy ez legyen most a forrássor sorszáma a könnyebb beazonosíthatóság végett.

```
#line sorszám
```

Sor vezérlés

```
$ cat preproc.c
#define N 30

#ifdef DEBUG
#define STRING "Debug"
#else
#define STRING "Release"
#endif
#line 200
main()
{
    int unix;
    char tomb[N] = STRING;
    for(unix=N-1; unix && tomb[unix]; unix--) {
        tomb[unix] = 0;
    }
}
```

Sor vezérlés

```
$ gcc -E preproc.c
# 1 "preproc.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "preproc.c"
# 200 "preproc.c"
main()
{
    int 1;
    char tomb[30] = "Release";
    for(1=30 -1; 1 && tomb[1]; 1 --) {
        tomb[1] = 0;
    }
}
```

Sor vezérlés

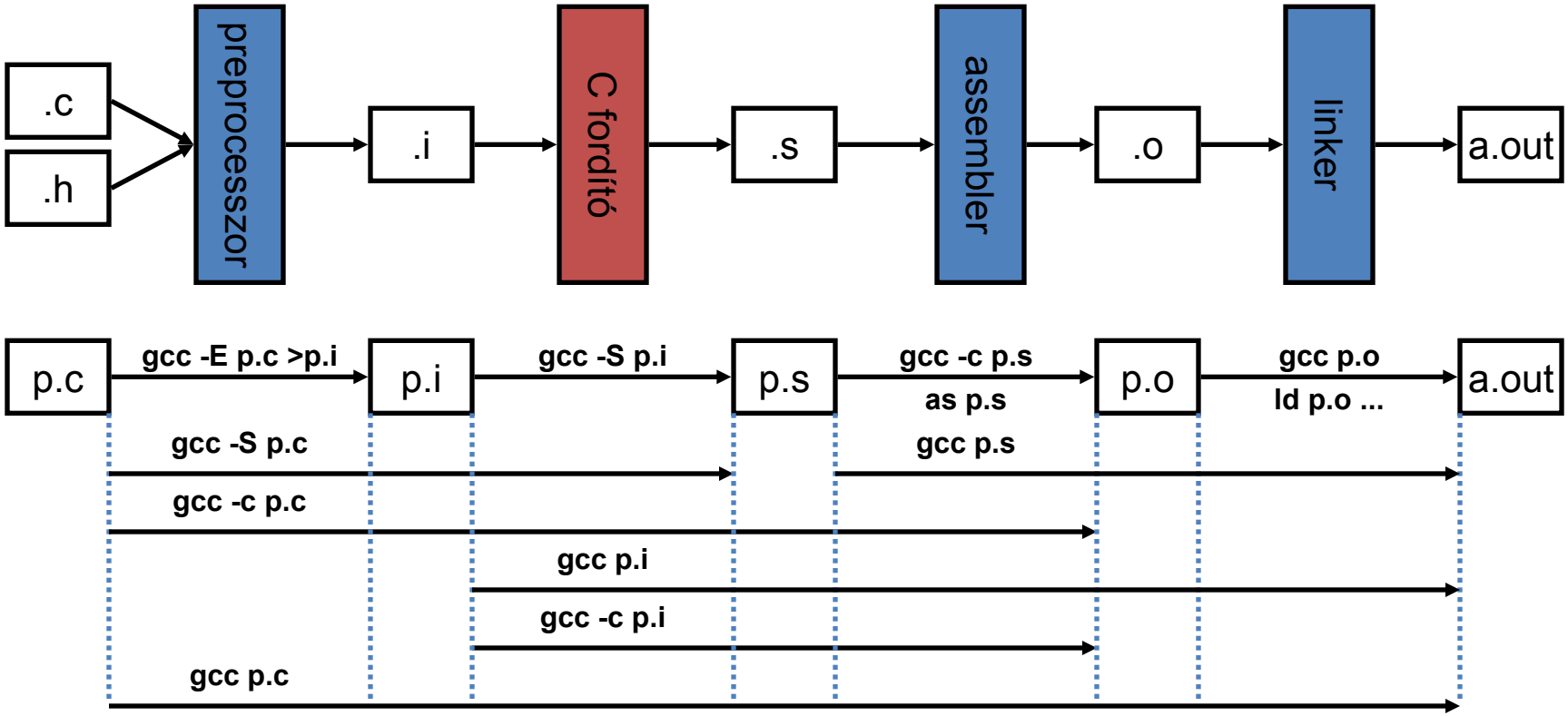
```
$ gcc -DDEBUG -E preproc.c
# 1 "preproc.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "preproc.c"
# 200 "preproc.c"
main()
{
    int 1;
    char tomb[30] = "Debug";
    for(1=30 -1; 1 && tomb[1]; 1 --) {
        tomb[1] = 0;
    }
}
```

Sor vezérlés

```
$ gcc preproc.c
preproc.c: In function 'main':
preproc.c:202: error: syntax error before numeric constant
preproc.c:204: error: invalid lvalue in assignment
preproc.c:204: error: invalid lvalue in decrement
```

A C forrás fordításának folyamata

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



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 C fordító nem csak „szó szerint” tudja lefordítani a C programot, hanem különféle optimalizálások elvégzésére is képes.

A C fordító

- 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.

A C fordító

- **-O1**: optimalizálás
  - A C fordító elvégzi azokat az optimalizálásokat amik mind a méretet, mind a futási időt csökkentik.
- **-O2**: optimalizálás futási időre
  - A C fordító az **O1** –ből indulva elvégzi azokat az optimalizálásokat amik csökkentik a futási időt, de a méreten (**O1**-hez képest) csak keveset növelnek.
- **-O3**: agresszív optimalizálás futásidőre
  - A C fordító elvégzi azokat az futásidőre optimalizáló algoritmusokat is, amik jelentősen növelhetik a kód méretét.

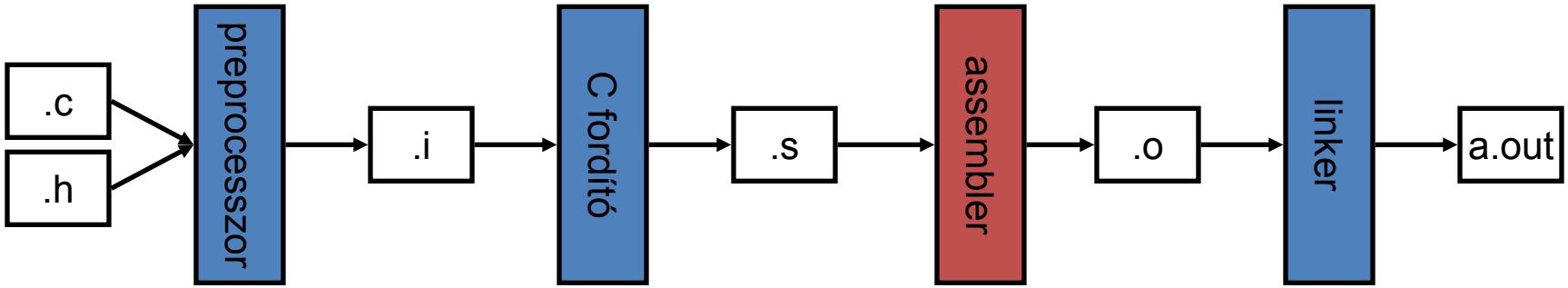
A C fordító

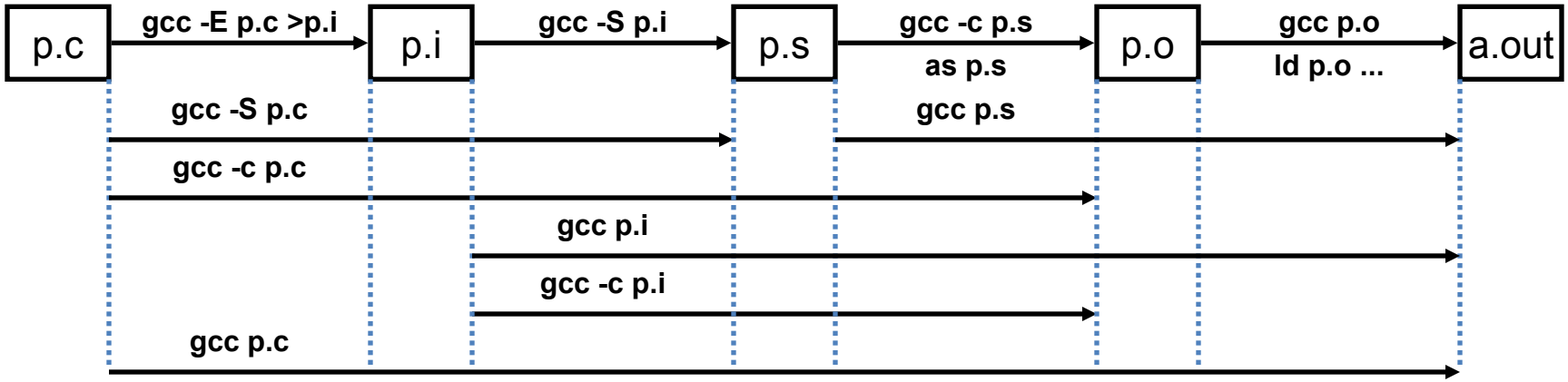
- **`-Os`**: optimalizálás méretre
  - A C fordító az **`O1`** –ből indulva elvégzi azokat az optimalizálásokat amik csökkentik a kód méretét, tekintet nélkül a futási időre gyakorolt hatásuktól.

- A fordítás gcc esetében nem közvetlenül gépi kódra, hanem assembly nyelvre történik.

A C forrás fordításának folyamata

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





Az assembler

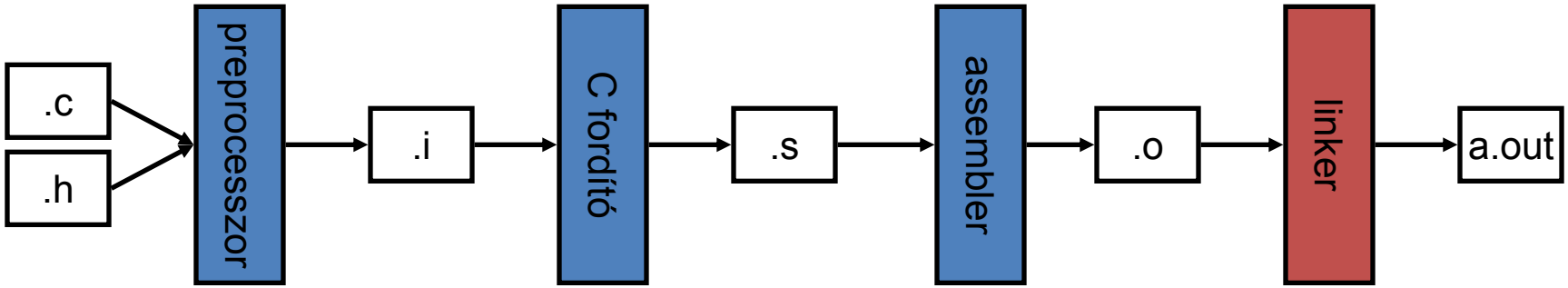
- Az assembly kódot fogja gépi kóddá fordítani, és egy úgynevezett object fájlt fog létrehozni.
- Egy ilyen object bináris formában tartalmazza az assembly kódból keletkezett gépi kódot, a programban található konstans értékeket (például a programban leírt számokat vagy sztringeket), illetve sok olyan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

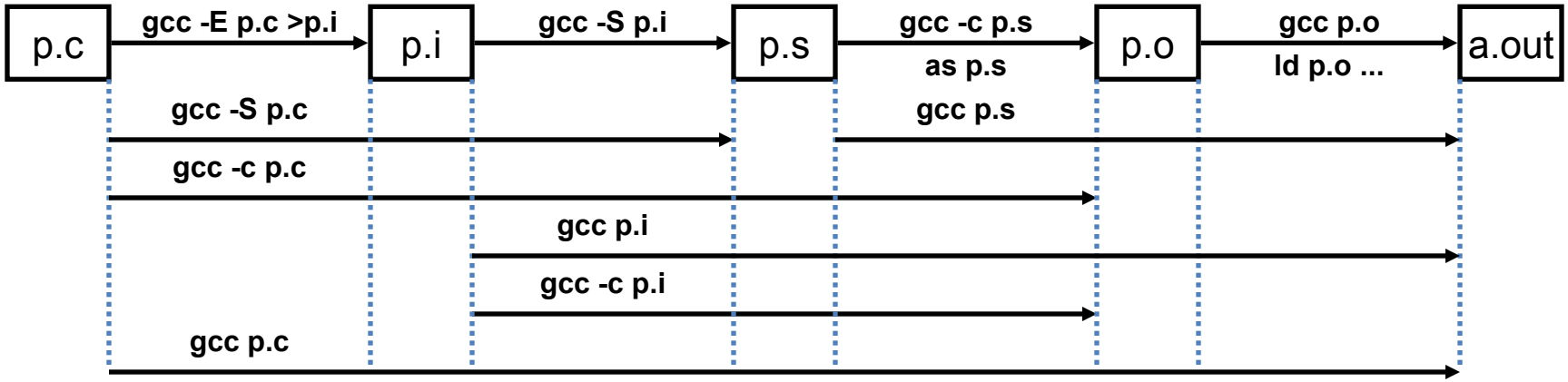
Fordítási egységek

- 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 C forrás fordításának folyamata

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





A linker

- Első megközelítésben a linker olyan object fájlból tud futtatható programot csinálni, amiben van **`main()`** függvény.

- De akkor miért kell ehhez linker?
- Mire jó a többi object (amiben nincs **`main`**)?

- Ahhoz, hogy ezt megértsük, meg kell ismerkednünk a modulok fogalmával.

Modulok

- 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 program elemek (konstans, típus, változó, függvény) együtteséből áll.

Modulok

- Egy modulban szereplő program elemek egy része felhasználható más modulok (programok) által, ezek képezik a modul közösségi, vagy látható részét, míg a modul többi része kívülről nem látható, ez a modul privát része.
- A modulnak e két részre osztása biztosítja a felhasznált modulok stabilitását és a privát elemek védelmét.

Modulok

- 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.

Modulok C-ben

- C-ben alapvető egységként adódik az egy forrás/object, mint legkisebb lehetséges modul.
- A `.o` kiterjesztésű object fájl ekkor a modul „lefordított formája” lesz, a `.c` kiterjesztésű forrás pedig a modul privát részének nyelvi formája.
- A modul közösségi részét pedig a már említett `.h` kiterjesztésű header fájlok valósítják meg.

Header fájlok

- A header és forrás fájlok tehát általában párosan léteznek:
  - A header fájl tartalmazza az összes olyan változó és függvény deklarációját, illetve konstans és típus definícióját, ami a modul 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 forrás pedig tartalmazza a közösségi részben deklarált függvények definícióit, illetve a modul teljes privát részét.

Header fájlok

- Ahhoz tehát, hogy olyan programot vagy másik modult írjunk, ami használja a modulunkat, elegendő a modulunk header fájlját ismerni.

- Az adott program forrásába a megfelelő **`#include`** helyére a preprocesszor bemásolja a mi header fájlunkat, így az object szintig lefordítható lesz.

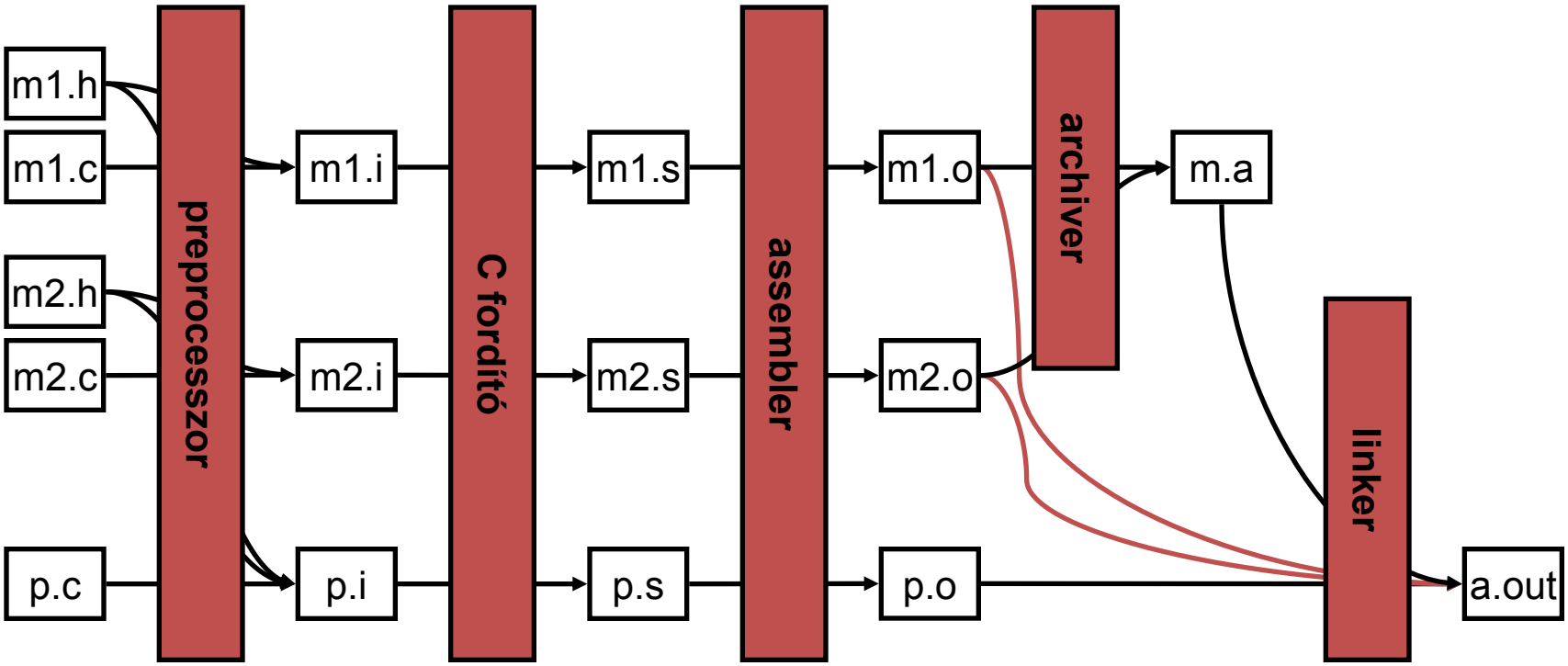
Modulok fordítása

- 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 össze tudjuk őket gyűjteni egy úgynevezett függvénykönyvtárban ami valójában egy olyan fájl, ami objecteket tartalmaz.
- A **`main()`** függvénnyel rendelkező objectekből viszont programok készíthetők, és ez a linker feladata.

A Linker

- Egy-egy program object-je önmagában eléggé hiányos lehet, hiszen ha egy modul valamelyik függvényét használja, annak csak a deklarációját ismeri. A függvény megvalósítása a modul object fájljában van. A linker feladata, hogy az ilyen objecteken átívelő hivatkozásokat feloldja, és egyetlen 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



Modulok

- Látható tehát, hogy egy-egy modul több programhoz is felhasználható, gondoljunk csak például a **`math.h`**-ra. Az ebben deklarált függvények (pl.: **`sin()`**) egy függvénykönyvtárban vannak megvalósítva. A **`-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 egy **`.o`** fájlt, hanem egy vagy akár több függvénykönyvtárat is takarhat.

Modulok C-ben

- Hogyan szokás tehát C-ben modulokat írni?

- Mivel a C nyelv nem engedi például típusok újradeklarálását, viszont bonyolultabb programrendszerekben gyakran előfordul, hogy ugyanazt a header fájlt (tranzitívan) többször is includolnák ugyanabba a forrásba, ezért szokás ezt feltételes preprocesszor direktívákkal kivédeni.

Modulok C-ben

- p.c

```
#include <stdio.h>
#include "m.h"

int main(int argc, char *argv[])
{
    printf("%d\n", fuggveny(argv[0]));
}
```

Modulok C-ben

- m.h

```
#ifndef M_H
#define M_H

int fuggveny(const char*);

#endif
```

Modulok C-ben

- m.c

```
#include "m.h"

int fuggveny(const char *str)
{
    int  cs;
    for(cs=0; *str; str++) {
        cs^=*str;
    }
    return cs;
}
```

Modulok C-ben

- 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]));
}
```

Modulok C-ben

- m.i

```
...

int fuggveny(const char*);
# 2 "m.c" 2

int fuggveny(const char *str)
{
    int cs;
    for(cs=0; *str; str++) {
        cs^=*str;
    }
    return cs;
}
```

Modulok C-ben

- A prímszámok keresését például megírhatjuk úgy, hogy a halmaz kezelését egy külön forrásban valósítjuk meg.

halmaz.h

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

#ifndef HALMAZ_H
#define HALMAZ_H

#ifndef BOOL
#define BOOL  int
#define FALSE 0
#define TRUE  1
#endif

#define K (8*sizeof(KisHalmaz))

typedef int KisHalmaz;

typedef unsigned long int halmazelem;
```

halmaz.h

```
typedef struct halmaztip {
        unsigned long int n;          /* Az univerzum a [0..N) */
        unsigned long int m;            /* A kishalmazok száma */
        KisHalmaz  *tar;               /* A kishalmazok tömbje */
} Halmaz;

Halmaz Letesit(halmazelem);
void Megszuntet(Halmaz);
void Uresit(Halmaz);
void Bovit(Halmaz, halmazelem);
void Torol(Halmaz, halmazelem);
BOOL Eleme(Halmaz, halmazelem);
void Egyesites(Halmaz H1, Halmaz H2, Halmaz H);
void Mettszet(Halmaz H1, Halmaz H2, Halmaz H);
void Kulonbseg(Halmaz H1, Halmaz H2, Halmaz H);
BOOL Egyenlo(Halmaz H1, Halmaz H2);
BOOL Resz(Halmaz H1, Halmaz H2);
BOOL Urese(Halmaz H);
void Ertekadas(Halmaz H1, Halmaz H2);
#endif
```

halmaz.c

```
/* A halmaz modul függvényeinek megvalósítása.
 * 2006. Augusztus 16. Gergely Tamás, gertom@inf.u-szeged.hu
 */

#include <stdlib.h>
#include "halmaz.h"
```

>>>

halmaz.c

```
Halmaz Letesit(unsigned long int n)
{
    Halmaz ret;
    ret.n = n;
    if(n) {
        ret.m   = (n-1)/K + 1;
        ret.tar = (KisHalmaz*)malloc(ret.m * sizeof(KisHalmaz));
    } else {
        ret.m   = 0;
        ret.tar = NULL;
    }
    return ret;
}
```

halmaz.c

```
void Megszuntet(Halmaz H)
{
    free(H.tar);
}

void Uresit(Halmaz H)
{
    long int i;
    for(i = 0; i < H.m; ++i) {
        H.tar[i] = 0;
    }
}
```

halmaz.c

```
void Bovit(Halmaz H, halmazelem x)
{
    if(x < H.n) {
        H.tar[x / K] |= (1 << (x % K));
    }
}

void Torol(Halmaz H, halmazelem x)
{
    if(x < H.n) {
        H.tar[x / K] &= ~(1 << (x % K));
    }
}
```

>>>

halmaz.c

```
BOOL Eleme(Halmaz H, halmazelem x)
{
    return (x < H.n) &&
           (H.tar[x / K] & (1 << (x % K)));
}

void Egyesites(Halmaz H1, Halmaz H2, Halmaz H)
{
    long int i;
    for(i = 0; i < H.m; ++i) {
        H.tar[i] = H1.tar[i] | H2.tar[i];
    }
}
```

>>>

halmaz.c

```
void Mettszet(Halmaz H1, Halmaz H2, Halmaz H)
{
    long int i;
    for(i = 0; i < H.m; ++i) {
        H.tar[i] = H1.tar[i] & H2.tar[i];
    }
}

void Kulonbseg(Halmaz H1, Halmaz H2, Halmaz H)
{
    long int i;
    for(i = 0; i < H.m; ++i) {
        H.tar[i] = H1.tar[i] & ~(H2.tar[i]);
    }
}
```

>>>

halmaz.c

```
BOOL Egyenlo(Halmaz H1, Halmaz H2)
{
    long int i;
    for(i=0; (i<H1.m) && (H1.tar[i] == H2.tar[i]); ++i);
    return i == H1.m;
}

BOOL Resz(Halmaz H1, Halmaz H2)
{
    long int i;
    for(i = 0; (i<H1.m) && !(H1.tar[i] & ~(H2.tar[i])); ++i);
    return i == H1.m;
}
```

>>>

halmaz.c

```
BOOL Urese(Halmaz H)
{
    long int i;
    for(i = 0; (i<H.m) && !(H.tar[i]); ++i);
    return i == H.m;
}

void Ertekadas(Halmaz H1, Halmaz H2)
{
    long int i;
    for(i = 0; i < H1.m; ++i) {
        H1.tar[i] = H2.tar[i];
    }
}
```

prim3.c

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

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

#define N 100000
```

>>>

prim3.c

```
int main() {
    Halmaz szita;
    halmazelem p, s;
    long int lepes, i, j;
    szita=Letesit(N);
    Bovit(szita, 2);                    /* a szita inicializálása */
    for(i = 3; i <= N; i+=2) {
      Bovit(szita, i);
    }
```

>>>

prim3.c

```
    p = 3;                                  /* az első szitálandó prím */
    while(p*p  <= N) {                 /* P többszöröseinek kiszitálása */
        lepes = 2 * p;                                 /* lépésköz = 2*p */
        s = p*p;                                /* s az első többszörös */
        while(s <= N) {
            Torol(szita, s);
            s += lepes;
        }
        do {                                /* a következő prím keresése */
            p += 2;
        } while((p < N) && (! Eleme(szita, p)));
    }
```

>>>

prim3.c

```
    j = 0;                  /* a prímszámok kiíratása képernyőre */
    printf("A prímszámok %ld-ig:\n", N);
    for(p = 2; p < N; ++p) {
        if(Eleme(szita, p)) {
            printf("%8d", p);
            if(++j==10) {
               putchar('\n');
               j=0;
            }
        }
    }
    putchar('\n');
    Megszuntet(szita);
}
```

Megvalósítás elrejtése

- Mire jók még a modulok?

- Programrendszer készítésekor az egyes modulokat célszerű úgy tervezni, hogy a headerben ne legyenek láthatók azok a programelemek, amelyek csak a megvalósításhoz kellenek.

Megvalósítás elrejtése

- 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.

Megvalósítás elrejtése

- Egyedül az adattípusok elrejtése okozhat problémát.
- Ugyanis azoknak az eljárásoknak a deklarációjában, amelyek a probléma megoldását adják – tehát a headerben kell lenniük – meg kell adni a paraméterek típusát, jóllehet a típus definiálását csak a modul `.c` forrásában kellene megadni, mert ezek megadása már a megvalósításhoz tartozik.

Megvalósítás elrejtése

- Adattípus megadásának elhalasztása, vagyis elrejtése a **`void*`** pointer felhasználásával megoldható.
- Ezt mutatja a következő séma, amelyben az **`mm.h`** az **`m.h t`** adattípusának elrejtését mutatja.
- Természetesen az **`mm.h`** -ban kell lennie olyan eljárásnak, amely **`t`** típusú dinamikus változót létesít.

Megvalósítás elrejtése

- m.h

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

- m.c

```
void fgv(t *x)
{

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

Megvalósítás elrejtése

- mm.h

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

- mm.c

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

Megvalósítás elrejtése

- Az elrejtés technikája lehetővé teszi, hogy modul tervezésekor meg tudjuk adni a header rész végleges alakját, anélkül, hogy a megvalósítás bármely részletét is ismernénk.
- Ez különösen hasznos absztrakt adattípusok tervezése és megvalósítása esetén.
- A függvénykönyvtárakból ráadásul csak azok az eljárások kerülnek bele a programba, amiket valóban meg is hívunk.

graf.h

```
/*
 * Gráf megvalósítása típuselrejtéssel. Közös header fájl.
 * 2006. Augusztus 17. Gergely Tamás, gertom.inf.u-szeged.hu
 */

#ifndef GRAF_H
#define GRAF_H

/* Típusdefiníciók */

typedef void *graf;
typedef int pont;
```

>>>

graf.h

```
graf letesit(int);
int pontokszama(graf);
void elbeszur(graf, pont, pont);
void eltorol(graf, pont, pont);
int vanel(graf, pont, pont);
int kifok(graf, pont);
int kielek(graf, pont, pont[]);
int befok(graf, pont);
int beelek(graf, pont, pont[]);
void megszuntet(graf);

#endif
```

graf1.c

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

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

typedef struct _graft {
    int   n;
    char *mx;
} _graft;
```

>>>

graf1.c

```
graf letesit(int n)
{
    _graft *ptr;
    ptr=(_graft*)malloc(sizeof(_graft));
    if(ptr) {
        ptr->n =n;
        ptr->mx=(char*)malloc(n*n*sizeof(char));
        if(!ptr->mx) {
            free(ptr);
            ptr=NULL;
        } else {
            int i, nn;
            for(i = 0, nn = n * n; i < nn; i++) {
                ptr->mx[i] = 0;
            }
        }
    }
    return (void*)ptr;
}
```

graf1.c

```
int pontokszama(graf g)
{
    return ((_graft*)g)->n;
}

void elbeszur(graf g, pont f, pont t)
{
    if(0 <= f && f < ((_graft*)g)->n &&
       0 <= t && t < ((_graft*)g)->n) {
        ((_graft*)g)->mx[((_graft*)g)->n * f + t]=1;
    }
}

void eltorol(graf g, pont f, pont t)
{
    if(0 <= f && f < ((_graft*)g)->n &&
       0 <= t && t < ((_graft*)g)->n) {
        ((_graft*)g)->mx[((_graft*)g)->n * f + t]=0;
    }
}
```

graf1.c

```
int vanel(graf g, pont f, pont t)
{
    return 0 <= f && f < ((_graft*)g)->n &&
           0 <= t && t < ((_graft*)g)->n &&
           ((_graft*)g)->mx[((_graft*)g)->n * f + t];
}

int kifok(graf g, pont p)
{
    int i,fok=-1;
    if(0 <= p && p < ((_graft*)g)->n ) {
        for(fok=i=0; i<((_graft*)g)->n; i++) {
            fok+=((_graft*)g)->mx[((_graft*)g)->n * p + i];
        }
    }
    return fok;
}
```

graf1.c

```
int kielek(graf g, pont p, pont l[])
{
    int i,fok=-1;
    if(0 <= p && p < ((_graft*)g)->n ) {
        for(fok=i=0; i<((_graft*)g)->n; i++) {
            if(((_graft*)g)->mx[((_graft*)g)->n * p + i]) {
                l[fok++]=i;
            }
        }
    }
    return fok;
}
```

>>>

graf1.c

```
int befok(graf g, pont p)
{
    int i,fok=-1;
    if(0 <= p && p < ((_graft*)g)->n) {
        for(fok=i=0; i<((_graft*)g)->n; i++) {
            fok+=((_graft*)g)->mx[((_graft*)g)->n * i + p];
        }
    }
    return fok;
}
```

>>>

graf1.c

```
int beelek(graf g, pont p, pont l[])
{
    int i,fok=-1;
    if(0 <= p && p < ((_graft*)g)->n ) {
        for(fok=i=0; i<((_graft*)g)->n; i++) {
            if(((_graft*)g)->mx[((_graft*)g)->n * i + p]) {
                l[fok++]=i;
            }
        }
    }
    return fok;
}

void megszuntet(graf g)
{
    if(g) { free(((_graft*)g)->mx); }
    free(g);
}
```

graf2.c

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

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

typedef struct _graft {
    int   n;
    int  *be;
    int  *ki;
    pont *mx;
} _graft;
```

>>>

graf2.c

```
graf letesit(int n) {
    _graft *ptr; ptr=(_graft*)malloc(sizeof(_graft));
    if(ptr) {
        ptr->n =n;
        ptr->be=(int*)malloc(n*sizeof(int));
        ptr->ki=(int*)malloc(n*sizeof(int));
        ptr->mx=(pont*)malloc(n*n*sizeof(pont));
        if(!(ptr->mx && ptr->be && ptr->ki)) {
            free(ptr->mx); free(ptr->be);
            free(ptr->ki); free(ptr); ptr=NULL;
        } else {
            int i;
            for(i = 0; i < n; i++) {
                ptr->be[i] = 0; ptr->ki[i] = 0;
            }
        }
    }
    return (void*)ptr;
}
```

graf2.c

```
int pontokszama(graf g)
{
    return ((_graft*)g)->n;
}

void elbeszur(graf g, pont f, pont t)
{
    if(0 <= f && f < ((_graft*)g)->n &&
       0 <= t && t < ((_graft*)g)->n &&
       !vanel(g, f, t)) {
        ((_graft*)g)->mx[ ((_graft*)g)->n * f +
            ((_graft*)g)->ki[f]++ ] = t;
    }
}
```

>>>

graf2.c

```
void eltorol(graf g, pont f, pont t)
{
    int i;
    _graft *gg=(_graft*)g;
    if(0 <= f && f < gg->n &&
       0 <= t && t < gg->n) {
        for(i=0;
            (i<gg->ki[f]) && (gg->mx[gg->n * f + i] != t);
            i++);
        for(; (i<(gg)->ki[f]); i++) {
            (gg)->mx[(gg)->n * f + i] =
                (gg)->mx[(gg)->n * f + i + 1];
        }
    }
}
```

>>>

graf2.c

```
int vanel(graf g, pont f, pont t) {
    int i;
    _graft *gg=(_graft*)g;
    if(0 <= f && f < gg->n &&
       0 <= t && t < gg->n) {
        for(i=0;
            (i<gg->ki[f]) && (gg->mx[gg->n * f + i] != t);
            i++);
        return i<gg->ki[f];
    }
    return 0;
}

int kifok(graf g, pont p) {
    if(0 <= p && p < ((_graft*)g)->n ) {
        return ((_graft*)g)->ki[p];
    }
    return -1;
}
```

graf2.c

```
int kielek(graf g, pont p, pont l[]) {
    int i,*ptr;
    if(0 <= p && p < ((_graft*)g)->n ) {
        for(i=0, ptr=((_graft*)g)->mx + p*((_graft*)g)->n+i;
            i<((_graft*)g)->ki[p]; i++, ptr++) {
            l[i]=*ptr;
        }
        return  ((_graft*)g)->ki[p];
    }
    return -1;
}

int befok(graf g, pont p) {
    if(0 <= p && p < ((_graft*)g)->n ) {
        return ((_graft*)g)->be[p];
    }
    return -1;
}
```

>>>

graf2.c

```
int beelek(graf g, pont p, pont l[]) {
    int i,*ptr;
    if(0 <= p && p < ((_graft*)g)->n ) {
        for(i=0, ptr=((_graft*)g)->mx + p*((_graft*)g)->n+i;
            i<((_graft*)g)->ki[p]; i++, ptr++) {
            l[i]=*ptr;
        }
        return ((_graft*)g)->be[p];
    }
    return -1;
}

void megszuntet(graf g) {
    if(g) {
        free(((_graft*)g)->mx);
        free(((_graft*)g)->be);
        free(((_graft*)g)->ki);
    }
    free(g);
}
```

Felhasznált anyagok

- Dévényi Károly (SZTE): Programozás alapjai
- Simon Gyula (PE): A programozás alapjai
- Pohl László (BME): A programozás alapjai
- B. W. Kernighan - D. M. Ritchie: A C programozási nyelv