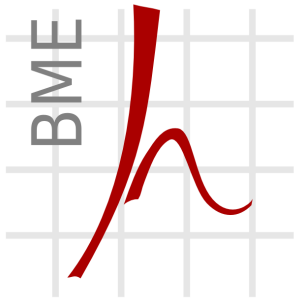
Dinamikus memóriakezelés. Operátorok

A programozás alapjai I.



Hálózati Rendszerek és Szolgáltatások Tanszék
Farkas Balázs, Fiala Péter, Vitéz András, Zsóka Zoltán

2021. október 26.

Tartalom

1. fejezet

Dinamikus memóriakezelés

Dinamikus memóriakezelés

- Olvassunk be egész számokat, és írjuk ki őket fordított sorrendben!
- A beolvasandó egész számok számát is a felhasználótól kérjük be!
- Csak annyi memóriát használjunk, amennyi feltétlenül szükséges!

1. Beolvassuk a darabszámot (`n`)
2. `n` egész szám tárolására elegendő memóriát kérünk az operációs rendszertől
3. Beolvassuk és eltároljuk a számokat, kiírjuk őket fordítva
4. Visszaadjuk a lefoglalt memóriát az operációs rendszernek

Példa

```
1  int n, i;
2  int *p;
3
4  printf("Hányat olvassak be? ");
5  scanf("%d", &n);
6  p = (int*)malloc(n*sizeof(int));
7  if (p == NULL) return;
8
9  printf("Kérek %d számot:\n", n);
10 for (i = 0; i < n; ++i)
11   scanf("%d", &p[i]);
12
13 printf("Fordítva:\n");
14 for (i = 0; i < n; ++i)
15   printf("%d ", p[n-i-1]);
16
17 free(p);
18 p = NULL;
```

link

p:0x0000

```
Hányat olvassak be?  5
Kérek 5 számot:
1 4 2 5 8
Fordítva:
8 5 2 4 1
```

A malloc és free függvények – <stdlib.h>

```
void *malloc(size_t size);
```

- size bájt egybefüggő memóriát foglal, és a lefoglalt terület címét visszaadja void* típusú értékként
- A visszaadott void* „csak egy cím”, ami nem dereferálható. Akkor lesz használható, ha átkonvertáljuk (pl. int*-gá).

```
1 int *p; /* int adat címe */
2 /* Memóriafoglalás 5 int-nek */
3 p = (int *)malloc(5*sizeof(int));
```

- Ha nem áll rendelkezésre elég egybefüggő memória, a visszatérési érték NULL. Ezt mindig ellenőrizni kell.

```
1 if (p != NULL)
2 {
3   /* használat, majd felszabadítás */
4 }
```

A malloc és free függvények – <stdlib.h>

```
void free(void *p);
```

- A p címen kezdődő egybefüggő memóriaterületet felszabadítja
- Méretet nem adjuk meg, mert azt az op.rendszer tudja (felírta a memóriaterület elé, ezért a kezdőcímmel kell hívni)
- free(NULL) megengedett (nem csinál semmit), ezért lehet így is:

```
1 int *p = (int *)malloc(5*sizeof(int));
2 if (p != NULL)
3 {
4   /* használat */
5 }
6 free(p);  /* nem baj, ha NULL */
7 p = NULL; /* ez jó szokás */
```

- Mivel a nullpointer nem mutat sehova, jó szokás felszabadítás után kinullázni a mutatót, így látni fogjuk, hogy nincs használatban.

malloc – free

- a `malloc` és a `free` kéz a kézben járnak
- ahány `malloc`, annyi `free`

```
1 char *WiFi = (char *)malloc(20*sizeof(char));
2 int *Lunch = (int *)malloc(23*sizeof(int));
3 ...
4 free(WiFi);
5 free(Lunch);
```

- Ha a felszabadítás elmarad, memóriaszivárgás (memory leak)
- Jó szokások:
  - Amelyik függvényben foglalunk, abban szabadítsunk
  - A `malloc` által visszaadott mutatót ne módosítsuk, ha lehet, ugyanazon keresztül szabadítsunk
- Van, hogy nem lehet tartani a jó szokásokat, ezt külön (kommentben) jelezzük

A calloc függvény – <stdlib.h>

```
void *calloc(size_t num, size_t size);
```

- egybefüggő memóriát foglal num darab, egyenként size méretű elemnek, a lefoglalt területet kinullázza, és címét visszaadja void* típusú értékként
- Használata szinte azonos a malloc-kal, csak ez elvégzi a num*size szorzást, és kinulláz.
- A lefoglalt területet ugyanúgy free-vel kell felszabadítani

```
1  int *p = (int *)calloc(5, sizeof(int));
2  if (p != NULL)
3  {
4    /* használat */
5  }
6  free(p);
```

A realloc függvény – <stdlib.h>

```
void *realloc(void *memblock, size_t size);
```

- korábban lefoglalt meóriaterületet átméretez `size` bájt méretűre
- új méret lehet kisebb is, nagyobb is, mint a régi
- ha kell, új helyre másolja a korábbi tartalmat, az új elemeket nem inicializálja
- visszatérési értéke az új terület címe

```
1 int *p = (int *)malloc(3*sizeof(int));
2 p[0] = p[1] = p[2] = 8;
3 p = realloc(p, 5*sizeof(int));
4 p[3] = p[4] = 8;
5 ...
6 free(p);
```

Példa

- Írjunk függvényt, mely a paraméterként kapott két sztringet összefűzi. A függvény foglaljon helyet az eredménysztringnek, és adja vissza annak címét.

```c
/* concatenate -- két sztring összefűzése
  dinamikusan foglal, az eredmény címét adja vissza
*/
char *concatenate(char *s1, char *s2) {
    size_t l1 = strlen(s1);
    size_t l2 = strlen(s2);
    char *s = (char *)malloc((l1+l2+1)*sizeof(char));
    if (s != NULL) {
        strcpy(s, s1);
        strcpy(s+l1, s2); /* vagy strcat(s, s2) */
    }
    return s;
}
```

link

Példa

A függvény használata

```
1  char word1[] = "ló", word2[] = "darázs";
2
3  char *res1 = concatenate(word1, word2);
4  char *res2 = concatenate(word2, word1);
5  res2[0] = 'v';
6
7  printf("%s\n%s", res1, res2);
8
9  /* A függvény memóriát foglalt, felszabadítani! */
10 free(res1);
11 free(res2);
```

link

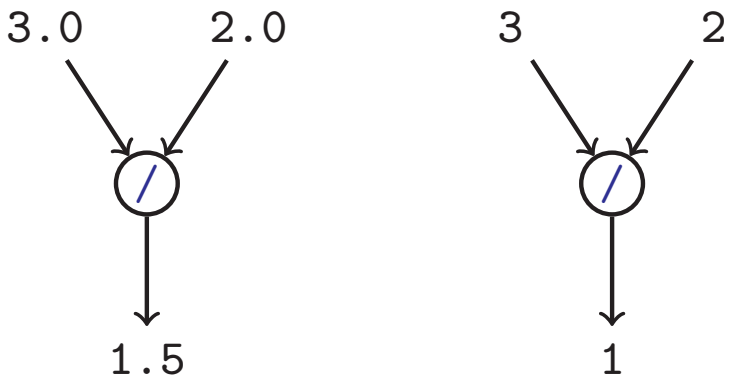
```
lódarázs
varázsló
```

2. fejezet

Operátorok

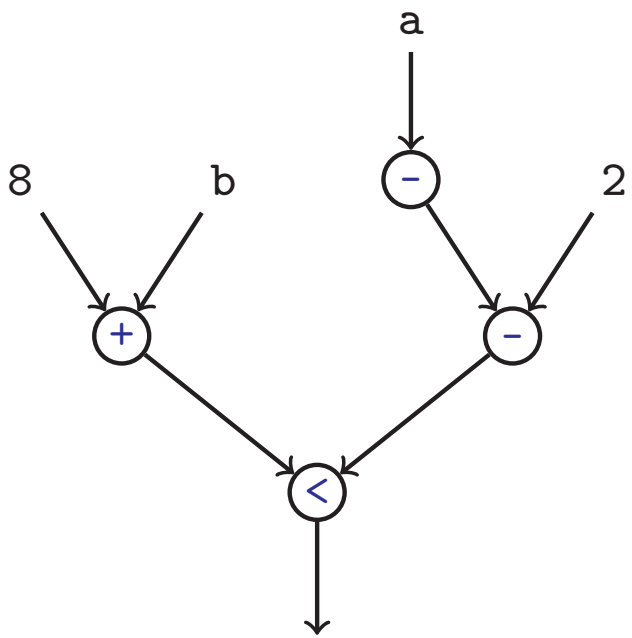
Operációk (műveletek)

- Kifejezések építőkövei
- Műveleti jellel jelöljük őket (+, -, *, &, ...)
- Operandusokon dolgoznak (argumentumok)
- Kiértékeléskor típusos adatot (érték) hoznak létre
- Többalakúak: eltérő típusú operandusokra eltérő működés



Kifejezések és kifejezésfák

```
8 + b < -a - 2
```



- Szerkezete fában ábrázolható
  - Levelek: konstansok és változóhivatkozások
  - Csomópontok: operátorok
  - Gyökér: kifejezés értéke

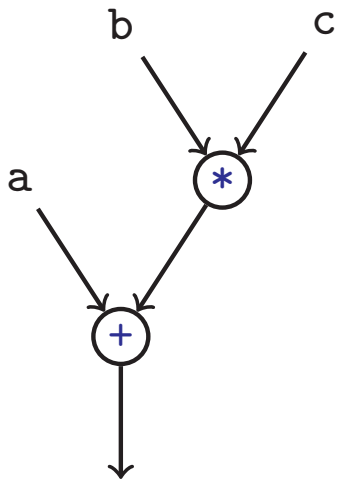
A fa felépítését az operátorok nyelvtana határozza meg:

- Precedencia
- Asszociativitás

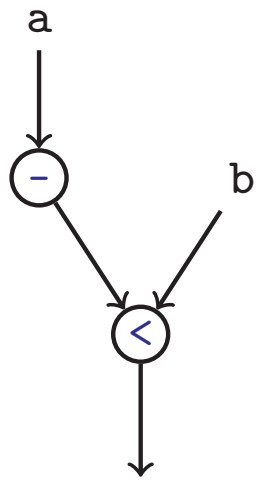
Precedencia

Két operátor találkozásakor
az erősebb precedenciájú operátor értéke lesz
a gyengébb precedenciájú operátor argumentuma

```
a + b * c
```




```
-a < b
```



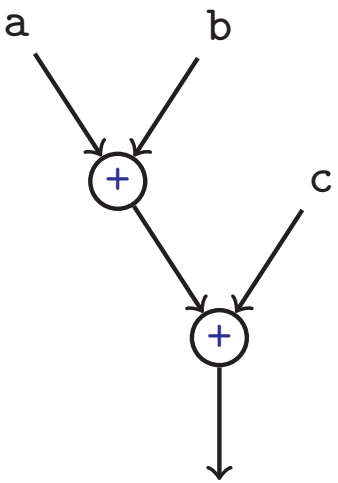
Asszociativitás (csoportosítás)

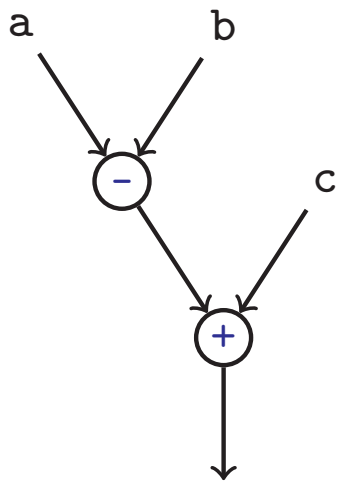
Két azonos precedenciájú operátor találkozásakor . . .

- Balról jobbra csoportosítás
  A bal oldai operátor értéke lesz a jobb oldali operátor argumentuma

```
a + b + c
(a + b) + c
```

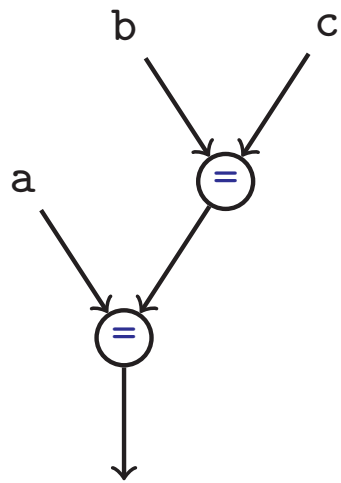
```
a - b + c
(a - b) + c
```



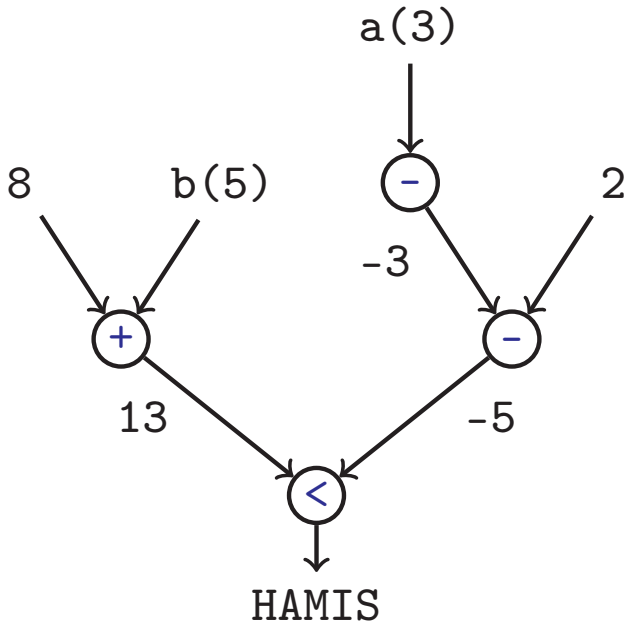


- Jobbról balra csoportosítás
  A jobb oldai operátor értéke lesz a bal oldali operátor argumentuma

```
a = b = c
a = (b = c)
```



Kifejezések kiértékelése



- Egy operátor operandusainak kiértékelési sorrendje
  - általában definiálatlan
    – a hatékonyság kedvéért
  - néhány esetben balról jobbra
    – a hatékonyság kedvéért

Az operátorok fajtái

- Az operandusok száma alapján
  - monadikus (unary) – egyoperandusú
    `-a`
  - diadikus (binary) – kétoperandusú
    `1+2`
- Az operandus értelmezése alapján
  - aritmetikai
  - összehasonlító, rendező
  - logikai
  - bitszintű
  - egyéb

Aritmetikai operátorok

művelet	**szintaxis**
egyoperandusú plusz	`+<kifejezés>`
egyoperandusú mínusz	`-<kifejezés>`
összeadás	`<kifejezés> + <kifejezés>`
kivonás	`<kifejezés> - <kifejezés>`
szorzás	`<kifejezés> * <kifejezés>`
bennfoglalás vagy osztás	`<kifejezés> / <kifejezés>`
az eredmény típusa az operandusok típusától függ, ha mindkettő egész, akkor egész osztás	
maradékképzés	`<kifejezés> % <kifejezés>`

Igazságérték (részben ismétlés)

- Egy érték igazságértékként értelmezve
  - hamis, ha értéke csupa 0 bittel van ábrázolva
  - igaz, ha értéke **nem** csupa 0 bittel van ábrázolva

```
1  while (1)     { /* végtelen ciklus */ }
2  while (-3.0)  { /* végtelen ciklus */ }
3  while (0)     { /* ide egyszer sem lépünk be */ }
```

- Minden igazságérték jellegű eredmény `int` típusú, és értéke
  - 0, ha hamis
  - 1, ha igaz

```
1  printf("%d\t%d", 2<3, 2==3);
```

```
1    0
```

Rendező és összehasonlító operátorok

művelet	szintaxis
relációs operátorok	`<kifejezés> < <kifejezés>`
	`<kifejezés> <= <kifejezés>`
	`<kifejezés> > <kifejezés>`
	`<kifejezés> >= <kifejezés>`
egyenlőség-vizsgálat	`<kifejezés> == <kifejezés>`
nem-egyenlő operátor	`<kifejezés> != <kifejezés>`

Logikai (`int`, 0 vagy 1) értéket adnak eredményként

Logikai operátorok

művelet	szintaxis
tagadás	`!<kifejezés>`

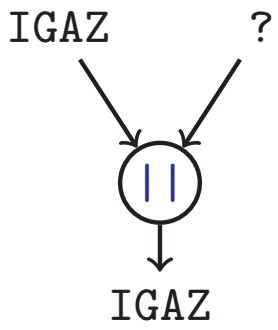
```
1  int a = 0x5c; /* 0101 1100, igaz */
2  int b = !a;   /* 0000 0000, hamis */
3  int c = !b;   /* 0000 0001, igaz */
```

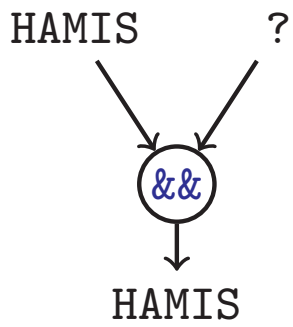
- Tanulság: `!!a` $\neq$ `a`, csak igazságérték szempontjából.

```
1  int vege = 0;
2  while (!vege) {
3    int b;
4    scanf("%d", &b);
5    if (b == 0)
6      vege = 1;
7  }
```

A logikai rövidzár

Az || és && operátorok operandusaikat balról jobbra értékelik ki, de a jobb oldalit csak akkor, ha a teljes kifejezés értéke még nem derült ki.






```
1  char *str = "Horvath Miklos";
2  if (i == 0 || str[i-1] == ' ')
3    /* szó eleje */
```

```
1  if (p != NULL && *p < 2)
2    /* ... */
```

További operátorok

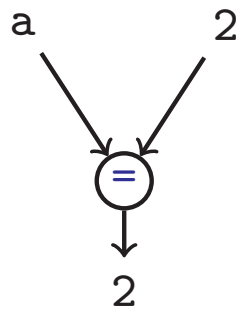
Már használtunk ilyeneket, csak nem mondtuk ki, hogy operátorok

művelet	**szintaxis**
függvényhívás	`<függvény>(<aktuális paraméterek>)`
tömbhivatkozás	`<tömb>[<index>]`
struktúratag-hivatkozás	`<struktúra>.<tag>`

```
1  c = sin(3.2); /* () */
2  a[28] = 3;    /* [] */
3  v.x = 2.0;    /* . */
```

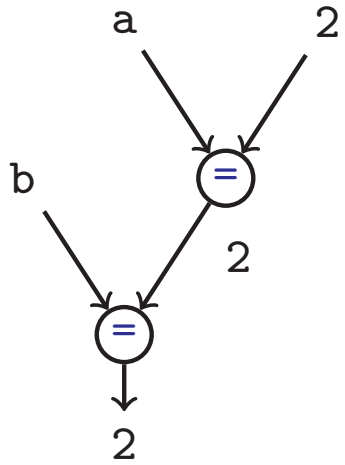
Operátorok mellékhatással

- Bizonyos operátorok kiértékelésének mellékhatása is van
  - főhatás: operátor értékének meghatározása
  - mellékhatás: operandus értéke változik
- Az értékadás operátor =
  - C-ben az értékadás kifejezés!
  - mellékhatása az értékadás (a megváltozik)
  - főhatása a új értéke



- A főhatás miatt ez is értelmes:

```
1    b = a = 2
2  /* b = (a = 2) */
```

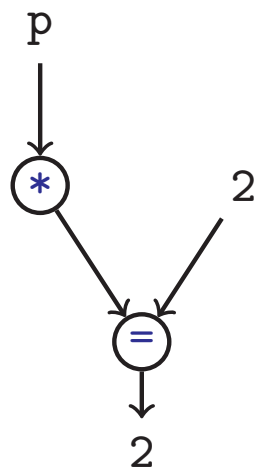


Balérték (lvalue)

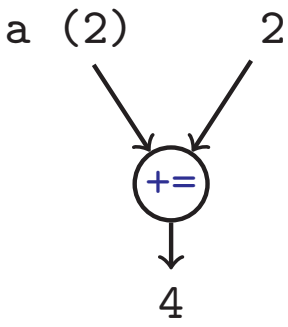
- Az értékadás operátor megváltoztatja a bal operandus értékét. A bal oldalon csak „változtatható dolog” állhat
- Balérték (lvalue): Olyan kifejezés, amely értékadás bal oldalán állhat

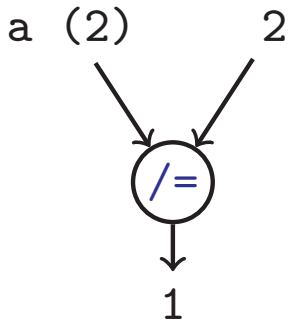
- balérték lehet
  - változóhivatkozás
  - tömbelem
  - dereferált mutató
  - struktúratag
  - struktúratag

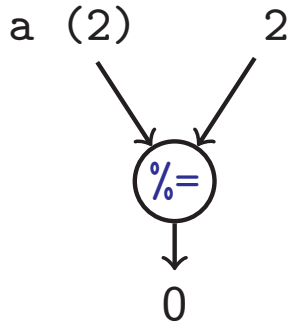
```
a     = 2
t[3]  = 2
*p    = 2
v.x   = 2
q->x  = 2
```



Viszonyított értékadás



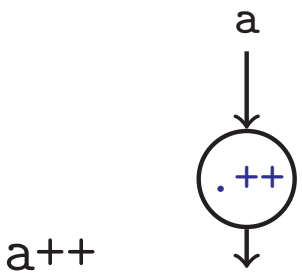


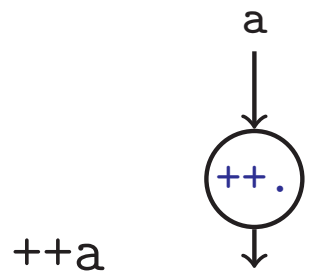


- Körülbelül: `<balérték> = <balérték> <op> <kifejezés>`
  de a balértéket csak egyszer értékeljük ki.

```
1  a += 2;           /* a = a + 2;                        */
2  t[rand()] += 2;   /* NEM t[rand()] = t[rand()] + 2;    */
```

Egyéb mellékhatásos operátorok





- a++ a-t növeli eggyel, visszaadja a eredeti értékét.
- ++a a-t növeli eggyel, visszaadja a új értékét.

```
1  b = a++; /* b = a;  a += 1;     posztinkremens */
2  b = ++a; /* a += 1; b = a;     preinkremens   */
```

```
1  /* tömb feltöltése */
2  int i = 0, a;
3  while (scanf("%d", &a) == 1)
4    t[i++] = a;          /* { t[i] = a; i++; } */
```

Kifejezés vagy utasítás?

Mellékhatásos kifejezés utasításként is szerepelhet a programban

Kifejezésutasítás

```
<Kifejezés>;
```

- A kifejezést kiértékeljük, és értékét eldobjuk.

```
1 a = 2;   /* kifejezésutasítások                          */
2 i++;     /* a főhatást elnyomjuk                         */
3 b %= 8; /* a mellékhatás utasításrangra emelkedik */
```

- Mivel a főhatást elnyomjuk, csak mellékhatásos kifejezésekből van értelme kifejezésutasítást alkotnunk.

```
1 2 + 3; /* helyes utasítás, semmit nem hajt végre */
```

Mutatókhoz kapcsolódó operátorok

művelet	**szintaxis**
dereferencia	`*<mutató>`
címképzés	`&<balérték>`
dereferencia és struktúratag-hivatkozás	`<mutató> -> <tag>`

Dereferencia esetén operandusként egy mutatót eredményül adó kifejezés kell álljon.

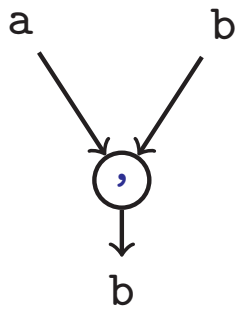
```
1  c = *(t+3); /* * */
2  p = &c;     /* & */
3  sp->a = 2.0;  /* -> */
```

További operátorok

művelet	szintaxis
kényszerített típusmódosítás (casting)	`(<típus>)<kifejezés>`
tárolás helyigénye (bájtokban) a kifejezést nem értékeljük ki	`sizeof <kifejezés>`

```
1  int a1=2, a2=3, meret;
2  double b;
3  b = a1/(double)a2;
4  meret = sizeof 3/a1;
5  meret = sizeof(double)a1;
6  meret = sizeof(double);
```

A vessző operátor

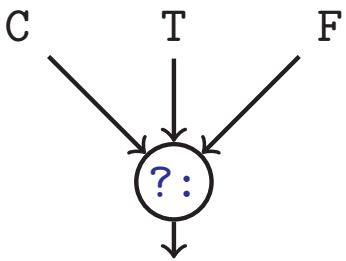


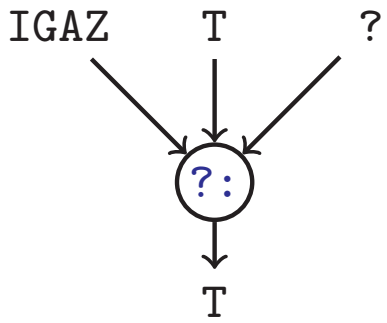
- Az operandusokat balról jobbra értékeli ki
- Az első kifejezés értékét eldobjuk.
- A teljes kifejezés értéke és típusa a második kifejezés értéke illetve típusa lesz.

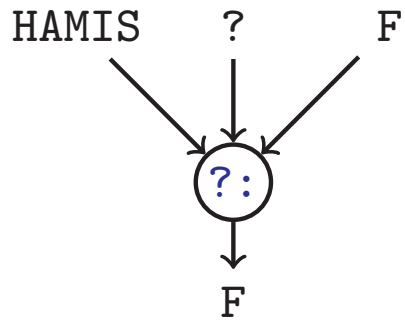
```
1 /* A ketjegyűek növekvő lépésközzel */
2 for(step=1,j=10; j<100; j+=step, step++)
3   printf("%d\n", j);
```

A ternáris operátor

A C nyelv egyetlen 3-operandusú (ternáris) operátora `C ? T : F`







- `C` kiértékelését követően `T` és `F` közül csak az egyiket értékeli ki.

```
1 /* b = abs(a) */
2 b = a < 0 ? -a : a;
```

```
1 /* c = max(a, b) */
2 c = a > b ? a : b;
```

- Nem helyettesíti az `if` utasítást, mivel az `if` két utasítás közül, míg a ternáris operátor két kifejezés közül választ.

A C nyelv operátorainak listája

A precedencia sorrend szerint rendezve (az azonos precedenciájúak egy sorban)

```
1  ()   []   .   -> /* legerősebb */
2  !   ~   ++   --   +   -   *   &   (<type>)   sizeof
3  *   /   %
4  +   -
5  <<   >>
6  <   <=   >   >=
7  ==   !=              /* tilos precedenciát tanulni! */
8  &                    /* tessék zárójelezni! */
9  ^
10 |
11 &&
12 ||
13 ?:
14 =   +=   -=   *=   /=   %=   &=   ^=   |=   <<=   >>=
15 , /* leggyengébb */
```

A C nyelv operátorai

Összefoglalva

- Sok, hatékony operátor
- Egyes operátoroknál a kiértékelés során mellékhatások is fellépnek
- Mindig igyekezzünk szétválasztani a fő- és mellékhatást ehelyett:

```
1  t[++i] = func(c-=2);
```

írjuk inkább ezt:

```
1  c -= 2;              /* ugyanazt jelenti */
2  ++i;                 /* ugyanolyan hatékony */
3  t[i] = func(c);      /* holnap is érteni fogom */
```

Mellékhatások szinkronizálása

A kifejezések kiértékelési sorrendje sokszor definiálatlan.

```
1 int i = 0, array[8];
2 array[++i] = i++; /* array[2] = 0;?  array[1] = 1;? */
```

Definiálatlan működés, „véletlen program”.

Sorrend-határ pont (sequence point)

A program végrehajtásának azon pontja, ahol

- minden előzőleg végrehajtott tevékenység mellékhatásának be kell fejeződnie.
- egyetlen későbbi végrehajtandó tevékenység mellékhatása sem kezdődhet el.

Ha „normális programokat” írunk, és nem keverjük a fő- és mellékhatásokat, nem kell foglalkoznunk vele.

3. fejezet

Típuskonverzió

Mi az?

Bizonyos esetekben a C-programnak konvertálnia kell kifejezéseink típusát.

```
1  long func(float f) {
2    return f;
3  }
4
5  int main(void) {
6    int i = 2;
7    short s = func(i);
8    return 0;
9  }
```

A példában: `int` → `float` → `long` → `short`

- `int` → `float` kerekítés, ha a szám nagy
- `float` → `long` túlcsordulhat, egészre kerekítés
- `long` → `short` túlcsordulhat

Típusok konverziója

- Alapelv
  - érték megőrzése, ha lehet
- Túlcsordulás esetén
  - a kapott érték elvileg definiálatlan

- Egyoperandusú konverzió (ezt láttuk)
  - értékadáskor
  - függvény hívásakor (a formális paraméterek aktualizálásakor)
- Kétoperandusú konverzió (pl. `2/3.4`)
  - műveletvégzéskor

Kétoperandusú konverzió

A két operandus azonos típussá alakítása az alábbi szabályoknak megfelelően

egyik operandus	másik operandus	közös, új típus
`long double`	bármi	`long double`
`double`	bármi	`double`
`float`	bármi	`float`
`unsigned long`	bármi	`unsigned long`
`long`	bármi (`int`, `unsigned`)	`long`
`unsigned`	bármi (`int`)	`unsigned`
`int`	bármi (`int`)	`int`

Típuskonverziók

Példa a konverzióra

```
1 int a = 3;
2 double b = 2.4;
3 a = a*b;
```

1. $3 \rightarrow 3.0$
2. $3.0 * 2.4 \rightarrow 7.2$
3. $7.2 \rightarrow 7$

Köszönöm a figyelmet.