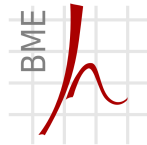




Dinamikus memóriakezelés. Operátorok

A programozás alapjai I.



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

2020. október 26.

1. fejezet

Dinamikus memóriakezelés

Tartalom

1 Dinamikus memóriakezelés

- Memória foglалás és -felszabadítás
- Sztring példa

2 Operátorok

- Definíciók
- Operátorok
- Precedencia
- Szinkronizálás

3 Típuskonverzió

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 használunk, 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;

```

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

```

1  int *p; /* int adat címe */
2  /* Memória foglalás 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 értéként
- 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 *mемblock, size_t size);
```

- korábban lefoglalt memóriaterületet 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.

```
1 /* concatenate -- két sztring összefűzése
2    dinamikusan foglal, az eredmény címét adja vissza
3 */
4 char *concatenate(char *s1, char *s2) {
5     size_t l1 = strlen(s1);
6     size_t l2 = strlen(s2);
7     char *s = (char *)malloc((l1+l2+1)*sizeof(char));
8     if (s != NULL) {
9         strcpy(s, s1);
10        strcpy(s+l1, s2); /* vagy strcat(s, s2) */
11    }
12    return s;
13 }
```



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);
```

```
lódarázs
varázsló
```

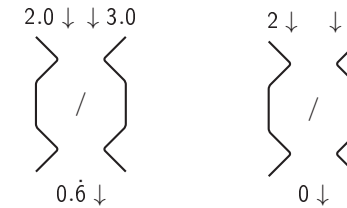


2. fejezet

Operátorok

Operációk (műveletek)

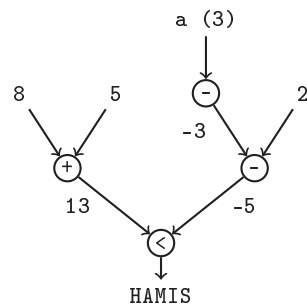
- Műveleti jellel jelöljük őket
- Operandusokon dolgoznak
- Típusos adatot hoznak létre
- Többalakúak: eltérő típusú operandusokra eltérő működés



Kifejezések és operátorok

■ Kifejezések

- pl. $8 + 5 < -a - 2$
- Konstansokból, változóhivatkozásokból és műveletekből épülnek fel



- kiértékelésük során egy típusos adatot kapunk eredményként.



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 ismételés)

- 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
	<kifejezés> < <kifejezés>
	<kifejezés> <= <kifejezés>
relációs operátorok	<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 }
```



Logikai operátorok

művelet	szintaxis
logikai és	<kifejezés> && <kifejezés>
logikai vagy	<kifejezés> <kifejezés>

Logikai rövidzár: Az operandusokat balról jobbra értékeljük ki. Csak addig, míg az eredmény nem egyértelmű. Ezt gyakran ki is használjuk

```
1 int a[5] = {1, 2, 3, 4, 5};
2 int i = 0;
3 while (i < 5 && a[i] < 20)
4     i = i+1; /* nincs túlindexelés */
```



További operátorok

Már használtunk ilyeneket, csak nem mondtuk ki, hogy operátorok

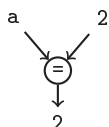
művelet	szintaxis
függvényhívás	<függvény>(<aktuális paraméterek>)
tömbhivatkozás	<tömb>[<index>]
struktúratag-hivatkozás	<struktúra>.<tag>

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



Operátorok mellékhatással

- Bizonyos operátoroknak mellékhatásuk is van
 - főhatás: a kiértékelés során kapott eredmény kiszámítása
 - 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 int a;
2 int b = a = 2;
```

- b-t az a=2 mellékhatásos kifejezés értékével inicializáljuk, melynek mellékhatásaként a is megváltozik.



Balérték

- 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 a = 2
 - tömbelem array[3] = 2
 - dereferált mutató *p = 2
 - struktúratag v.x = 2
- Nem balérték kifejezések (példák)
 - konstans 3 = 2 hiba
 - aritmetikai kifejezés a+4 = 2 hiba
 - logikai kifejezés a>3 = 2 hiba
 - függvényérték sin(2.0) = 2 hiba



Kifejezés vagy utasítás?

Mellékhatásos kifejezés utasításként is szerepelhet a programban

Kifejezésutasítás

<Kifejezés>;

- A kifejezést kiértékeljük, és értékét eldobjuk.

```
1 a = 2 /* kifejezés, az értéke 2, mellékhatásos */
```

```
1 a = 2; /* utasítás, nincs értéke */
2 /* a mellékhatást hajtja végre */
```

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

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



Értékadó operátorok

művelet	szintaxis
	<balérték> += <kifejezés>
	<balérték> -= <kifejezés>
viszonyított értékadás	<balérték> *= <kifejezés>
	<balérték> /= <kifejezés>
	<balérték> %= <kifejezés>

- Körülbelül: <balérték>=<balérték><op><kifejezés>

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

A balértéket csak egyszer értékeljük ki.



Egyéb mellékhatásos operátorok

művelet	szintaxis
utótagos inkrementálás	<balérték> ++
utótagos dekrementálás	<balérték> --
kiértékelés után növeljük/csökkentjük eggyel	
előtagos inkrementálás	++<balérték>
előtagos dekrementálás	--<balérték>
kiértékelés előtt növeljük/csökkentjük eggyel	

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

```
1 for (i = 0; i < 5; ++i) { /* ötször */ }
```



Mutatókhoz kapcsolódó operátorok

művelet	szintaxis
dereferencia	*<mutató>
címképzés	&<balérték>
dereferencia és struktúrhtag-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)	<code>(<típus>)<kifejezés></code>
tárolás helyigénye (bájtokban) a kifejezést nem értékeljük ki	<code>sizeof <kifejezés></code>

```
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);
```



További operátorok

művelet	szintaxis
feltételes kifejezés	<code><felt.> ? <kif.1> : <kif.2></code>

- ha <felt> igaz, akkor <kif.1>, egyébként <kif.2>
- <kif.1> és <kif.2> közül csak az egyiket értékeljük ki
- Nem helyettesíti az `if` utasítást

```
1 a = a < 0 ? -a : a; /* abszolút érték képzése */
```



További operátorok

művelet	szintaxis
vessző	<code><kifejezés> , <kifejezés></code>

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



Adatokkal végzett operációk (műveletek) jellemzői

Precedencia

eltérő műveletek találkozásakor melyik művelet értéke lesz a másik művelet argumentuma?

```
1 int a = 2 + 3 * 4; /* 2 + (3 * 4) */
```

Asszociativitás

azonos precedenciájú műveletek találkozásakor melyik művelet értéke lesz a másik művelet argumentuma?
(Balról jobbra vagy jobbról balra köt?)

```
1 int b = 11 - 8 - 2; /* (11 - 8) - 2 */
2 int a = b = 3; /* a = (b = 3) */
```

A szabályok megjegyzése helyett inkább zárójelezzünk!



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 */

```



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 végrehajtható tevékenység mellékhatása sem kezdődhet el.

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



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 */

```

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` \rightarrow `float` \rightarrow `long` \rightarrow `short`

- `int` \rightarrow `float` kerekítés, ha a szám nagy
- `float` \rightarrow `long` túlcsordulhat, egészre kerekítés
- `long` \rightarrow `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)
- Kétooperandusú konverzió (pl. 2/3.4)
 - műveletvégzéskor



Kétooperandusú konverzió

A két operandus azonos típusú alakítása az alábbi szabályoknak megfelelően

egyik operandus	másik operandus	közös, új típus
<code>long double</code>	bármilyen	<code>long double</code>
<code>double</code>	bármilyen	<code>double</code>
<code>float</code>	bármilyen	<code>float</code>
<code>unsigned long</code>	bármilyen	<code>unsigned long</code>
<code>long</code>	bármilyen (<code>int</code> , <code>unsigned</code>)	<code>long</code>
<code>unsigned</code>	bármilyen (<code>int</code>)	<code>unsigned</code>
<code>int</code>	bármilyen (<code>int</code>)	<code>int</code>



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`