Kvíz

- http://www.menti.com
- mindenki a saját azonosítójával (bbbbnnnn) lépjen be

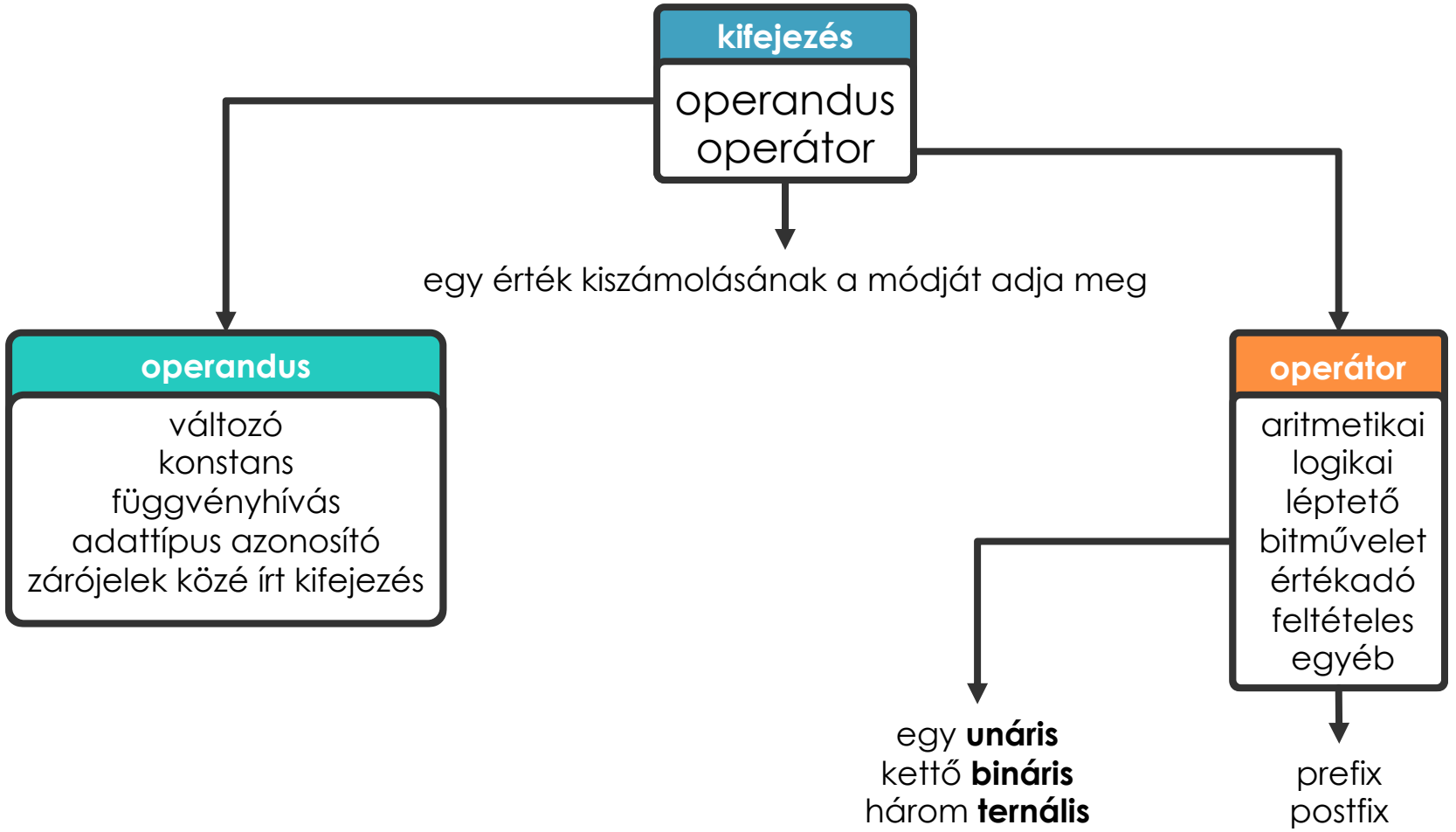
! aki **nem** az azonosítóját használja, nem kapja meg a pontokat a kvízre

3742 3152

Kifejezések és operátorok

KURZUS

Kifejezések és operátorok



Operátorok
aritmetikai

unáris

+	unáris plusz
–	unáris mínusz

→ kizárólag **1** operandusra alkalmazható

→ előjel

bináris

+	összeadás
–	kivonás
*	szorzás
/	osztás
%	maradékszámítás

→ legalább **2** operandusra van szükség

→ matematikai műveletek

alkalmazás

egész `int`, `char`
valós `float`, `double`

operandusokra

KIVÉTEL

a **%** operátor csak **egész** típusú operandusokra alkalmazható

FONTOS

az eredmény típusát az operandusok típusa dönti el

Operátorok

aritmetikai

példa

```
int a, b, c = +3;
b = -4;

a = b - c + 1; //a = -6
a = a * b/2; //a = 12
c = a % 5; //c = 2
```

unáris

bináris

példa

```
int a = 5, b = 2;
float x = 5;

a = a/b; //a = 2
x = x/b; //x = 2.5
x = 5/b; //x = 2.0
```

az eredmény típusát az operandusok típusa dönti el

!!!FONTOS!!!

a **/** és **%** operátorok jobb oldalán szereplő operandus értéke nem lehet **0**

kerekítés

C89 implementálás függő
C99 mindig a **0** irányába

példa

```
e =  7/5; //e =  1 (kerekítve  1.4 értékről)
e = -7/5; //e = -1 (kerekítve -1.4 értékről)
e =  9/5; //e =  1 (kerekítve  1.8 értékről)
e = -9/5; //e = -1 (kerekítve -1.8 értékről)
```

Operátorok

összehasonlító és logikai

összehasonlító

`<`	kisebb
`>`	nagyobb
`<=`	kisebb-egyenlő
`>=`	nagyobb-egyenlő
`==`	egyenlő
`!=`	nem egyenlő

- leggyakoribb felhasználása ciklusokban és feltételekben
- az eredmény mindig `int` típusú — **értéke** →
  - `1` ha a reláció **igaz**
  - `0` ha a reláció **hamis**

logikai

`!`	tagadás
`&&`	logikai **és**
`||`	logikai **vagy**

→ logikai kifejezésekben összetett feltételek megfogalmazására alkalmas

Operátorok

összehasonlító és logikai

példa

```
5 < 10  // 1
10 < 5  // 0
3 > 2.5 // 1

a + b <= c - 1

a < b < c
```

→ összekombinálhatók egész és valós operandusok

→ valójában `(a + b) <= (c - 1)` → betartva a műveletek precedenciáját

→ `(a < b) < c` → a baloldali asszociativitás miatt

FIGYELEM → azt **NEM** ellenőrzi, hogy a **b** értéke **a** és **c** között van-e!!!

példa

```
a == 2
a != b

a < b == b < c
```

→ `1` ha az **a** operandus értéke `2`, `0` minden más esetben

→ `1` ha **a** és **b** értéke különböző, `0` ha megegyezik

→ `(a < b) == (b < c)` → `1` ha mindkét kifejezés egyszerre igaz vagy hamis, `0` ellenkező esetben

Operátorok

összehasonlító és logikai

példa

```
!expr
expr1 && expr2
expr1 || expr2
```

`1` ha az **`expr`** kifejezés logikai értéke `hamis`, `0` ha `igaz`

`1` ha az **`expr1`** `ÉS` **`expr2`** kifejezés logikai értéke egyidőben `igaz`, `0` ha `hamis`

`1` ha az **`expr1`** `VAGY` **`expr2`** kifejezés logikai értéke `igaz`, `0` ha `hamis`

példa

```
(a != 0) && (a%5 == 0)
```

ha egy összetett kifejezés globális eredménye levezethető a részkifejezésből, akkor a jobboldala már nem kerül kiértékelésre

3742 3152

kvíz 1. kérdés

```
a < b < c
```

azt **NEM** ellenőrzi, hogy a **b** értéke **a** és **c** között van-e!!!

hogyan tudnánk felírni azt a feltételt ami mégis azt ellenőrzi?

Operátorok

léptető

léptető

++ 1-vel való növelés
-- 1-vel való csökkentése

- egész és valós operandusokkal egyaránt működik
- hatását az operandus értékére fejti ki
- postfix és prefix alakban is használható
- ha egy kifejezésben egyetlen operandus van, akkor mindegy, hogy postfix vagy prefix alakban használjuk őket
  - egy kifejezés kiértékelésében való használatkor nagyon óvatosnak kell lenni

! C nyelvben nem szokás az **a = a + 1** értékadás, helyette mindig léptetést használunk

példa

```
int a = 4, x, y;
x = ++a;
y = a++;
```

- az értékadás előtt az **a** értéke megnövekszik, ezért **a = 5** és **x = 5**
- az értékadás előtt az **y** felveszi az **a** pillanatnyi értékét, majd az **a** megnövekszik, ezért **y = 5** és **a = 6**

Operátorok

bitművelet

Operátorok

bitművelet

példa

```
a = 10
b = 7
a & b
a | b
a ^ b
~a
~b
```

bináris reprezentáció

```
1  0  1  0
0  1  1  1
0  0  1  0
1  1  1  1
1  1  0  1
0  1  0  1
1  0  0  0
```

→ `1` ha a mindkét bit `1`, minden más esetben `0`
→ `1` ha két bit közül legalább egy `1`, minden más esetben `0`
→ `1` ha csakis az egyik a két bit közül `1`, minden más esetben `0`
→ `1` ahol a bit `0`, `0` ahol `1` volt
→ `1` ahol a bit `0`, `0` ahol `1` volt

példa

```
a = 12
b = 3600
a << 1
a << 2
a << 5
a >> 1
a >> 2
b >> 4
```

bináris reprezentáció

```
0000 0000 0000 1100
0000 1110 0001 0000
0000 0000 0001 1000
0000 0000 0011 0000
0000 0001 1000 0000
0000 0000 0000 0110
0000 0000 0000 0011
0000 0000 1110 0001
```

→ $24 = 12 * 2^1$
→ $48 = 12 * 2^2$
→ $384 = 12 * 2^5$
→ $6 = 12 / 2^1$
→ $3 = 12 / 2^2$
→ $225 = 3600 / 2^4$

a felszabaduló pozíciókat a `0` jelzi

Operátorok
értékadó

értékadó

= egyszerű értékadás

lvalue és **rvalue**

- a memória helyére utal, amely egy operandust azonosít
- az értékadás során elhelyezkedhet bal- vagy jobb oldalon
- más nevén **locator value**
- általában az operandust azonosítja

- az adatra vonatkozik, amit valahol a memóriában tárolunk
- nem lehet neki értéket adni

példa

```
a = 5;  5 = a;
```

! könnyen megjegyezhető **lvalue** csak balról (**left**) és **rvalue** csak jobbról (**right**) helyezkedhet el

kiértékeli az operátor jobb oldalán levő kifejezést, majd ennek eredményét a bal oldalon található operandus veszi fel

példa

```
a = 5;
b = a;
c = a + (b + 9) * 3;
```

hagyományos forma

hagyomásnyostól eltérő forma

az értékadás kifejezése megjelenhet egy másik kifejezésben, mint operandus

példa

```
a = 5;
c = a + ((b = a) + 9) * 3;
a = b = c = 0;
```

zavaros, nehezen áttekinthető kód

Operátorok
értékadó

3742 3152

tömörebb forma

változó = változó operátor kifejezés
helyett
változó operátor= kifejezés

- gyorsabb, áttekinthetőbb kód
- mindkettő használható
- igyekszünk a tömör formát előnyben részesíteni
- összetett kifejezések esetén nem feltétlenül egyértelmű a kibontás

hagyományos	tömör
a = a + b	a += b
a = a - b	a -= b
a = a * b	a *= b
a = a / b	a /= b
a = a % b	a %= b
a = a << b	a <<= b
a = a >> b	a >>= b
a = a & b	a &= b
a = a | b	a |= b
a = a ^ b	a ^= b

példa `a *= b + c;` → kvíz 2. kérdés → `a = a * (b + c);`

!!!FONTOS!!!

ha az értékadás két oldalán nem ugyan olyan típusú operandus van, akkor a C **implicit** átalakítást végez

példa

```
int a;
float b;

a = 12.34;
b = 123;
```

→

```
a = 12
b = 123.000000
```

Operátorok
feltételes és egyéb

feltételes

`kifejezés1 ? kifejezés2 : kifejezés3`

- az egyetlen operátor amelynek `3` operandusra van szüksége
- hasonló az `if` utasításhoz
- először a **`kifejezés1`** értékelődik ki, ami ha nem `0`, tehát **`IGAZ`**, akkor a **`kifejezés2`** adja a kifejezés értékét, különben a **`k`**
- a feltételes kifejezés típusa, mindig a **`kifejezés2`** és a **`kifejezés3`** közül a nagyobb pontosságúval lesz megegyező

példa

```
int a = 3, b = 5, max;
max = a > b ? a : b;
```

példa

```
a % 2 ? printf("páratlan") : printf("páros");
```

egyéb

`[]`	tömb elemeinek elérése

példa

```
int T[100];
T[5] = 9;
```

Operátorok
egyéb

pointer

`&`	cím
`*`	dereferencia

szorosan kapcsolódnak a pointerekhez

példa

```
int a, *p;
p = &a;
*p = 3;
```

- a **p** egy `int` típusú pointer
- a **p** az **a** operandusra mutató pointer
- az **a** értéke `3` lesz

egyenértékű az
`a = 3;`
értékadással

a cím operátornak egy `lvalue` operandusra van szüksége, ami csak jobb oldalon helyezkedhet el

`&n` → ez a kifejezés csakis akkor helyes ha **n** egy `lvalue`

HIBA `&12` → ez a kifejezés helytelen, mert `12` nem egy címezhető objektum

3742 3152

kvíz 3. kérdés

egyéb

`sizeof`

megadja az operandus által a memóriában elfoglalt oktettek számát

példa

```
int a;
printf("%d\n", sizeof(a));
```

kiírja a képernyőre a `4`-es számot

Operátorok
egyéb

típuskonverzió

`(típusnév) kifejezés;`

- implicit (automatikus)

példa

```
int i, j;
float f, m;
m = i + f;
j = i + f;
```

- az **i** `float` típusra konvertálódik
- az **f** változó esetén adatvesztés lép fel, a **j** változóban összeg törtrésze elveszik

- explicit

példa

```
int a = 12, b = 5;
float atlag;

atlag = (float)a/(float)b;
atlag = a/b;
```

- az eredmény `2.4`
- az eredmény `2` **HIBA**

egyéb

`,`

- kifejezések tagolása balról jobbra
- a láncolt kifejezés utolsó elemenének értéke egyenlő az összetett kifejezés értékével

példa

```
int i = (a, b);
int i = (a += 2, a + b);
```

- az **i** felveszi a **b** értékét
- előbb az **a** értéke növekszik `2` majd az **i** felveszi az **a + b** értékét

Operátorok

precedencia és asszociativitás

precedencia	operátor	leírás	asszociativitás

1	`++ --`
	`()`
	`[]`
	`.`
	`->`
	`(típus){lista}`
		postfix növelés/csökkentés
		függvényhívás
		tömbök elemei
		struktúrák és uniók elemeinek elérése
		mint az előző csak pointer által
		összetett literál (**C99**)
			balról jobbra

2	`++ --`
	`+ -`
	`! ~`
	`(típus)`
	`*`
	`&`
	`sizeof`
	`_Alignoff`
		prefix növelés/csökkentés
		unáris plusz/mínusz
		logikai és bitműveleti tagadás
		típuskonverzió
		dereferencia
		cím
		méret
		igazítás követelmény (**C11**)
			jobbról balra

3	`* / %`	szorzás, osztás, maradékos osztás	balról jobbra
4	`+ -`	összeadás és kivonás	
5	`<< >>`	biteltolás balra és jobbra	
6	`< <=` `> >=`	érték összehasonlítás	
7	`== !=`	egyenlőség/nem egyenlőség	
8	`&`	bitszintű ÉS (AND)	
9	`^`	bitszintű kizáró VAGY (XOR)	
10	`|`	bitszintű VAGY (OR)	
11	`&&`	logikai ÉS (AND)	
12	`||`	logikai VAGY (OR)	
13	`?:`	**3** operandusú feltételes	jobbról balra

14	`=`
	`+= -=`
	`*= /= %=`
	`<<= >>=`
	`&= ^= |=`
		értékadás
		értékadás összeadás/kivonás
		értékadás szorzás/hányadós/maradékos osztás
		értékadás biteltolás barla/jobbra
		értékadás bitszintű AND, XOR és OR	

15	`,`	vessző	balról jobbra

Implicit típuskonverzió

- megengedett több, különböző típusú operandus kombinálása egy kifejezésben
- **PROBLÉMA** a bináris operátorok (amelyek két operandusra alkalmazhatók) megkövetelik, hogy az operandusoknak azonos legyen a típusa, ahhoz, hogy a műveletet el lehessen végezni
- **MEGOLDÁS** implicit konvertálás
  - a fordítóprogram észrevehetetlen módon azonos típusúvá alakítja az operandusokat mielőtt gépi kóddá alakítja a forráskódot
  - amennyiben ezt nem tudja elvégezni hibaüzenettel jelez a programozónak
- **ALTERNATÍVA** explicit típuskonverzió

Implicit típuskonverzió

- értékadáskor történő implicit típuskonverzió
- megjelenhetnek problémák
  - értékvesztéssel járhat, hogyha a bal oldalon lévő operandus típusa nem elég befogadó
  - nem lehet elvégezni az átalakítást
- aritmetikai típusok implicit konverziójának rangsora

egész `_Bool` `char` `short` `int` `long` `long long`

valós `float` `double` `long double`

  - az átalakítás arra a legközelebbi típusra történik amelyik mindkét operandus értékét megfelelően ábrázolni tudja

példa

```
int a = 12, b = 5;
float atlag;

atlag = (float)a/(float)b;
atlag = a/b;
```

az eredmény `2.4`

az eredmény `2` HIBA

Implicit típuskonverzió
szabályok és megjegyzések

- annak a típusnak nagyobb a rangja, amely több biten van reprezentálva
- ugyan azon típusnak az előjel nélküli változata nagyobb rangú, mint az előjeles
- a valós típusnak nagyobb a rangja, mint az egész típusnak
- ha egy kifejezésben az összes operandus egész értéket tartalmaz, akkor **`int`** típussá alakítódnak
  - **C99** szabványban előbb a kisebb rangú egészeket (**`_Bool char short`**) „lépteti elő” **`int`** vagy **`unsigned int`** típusúra

példa

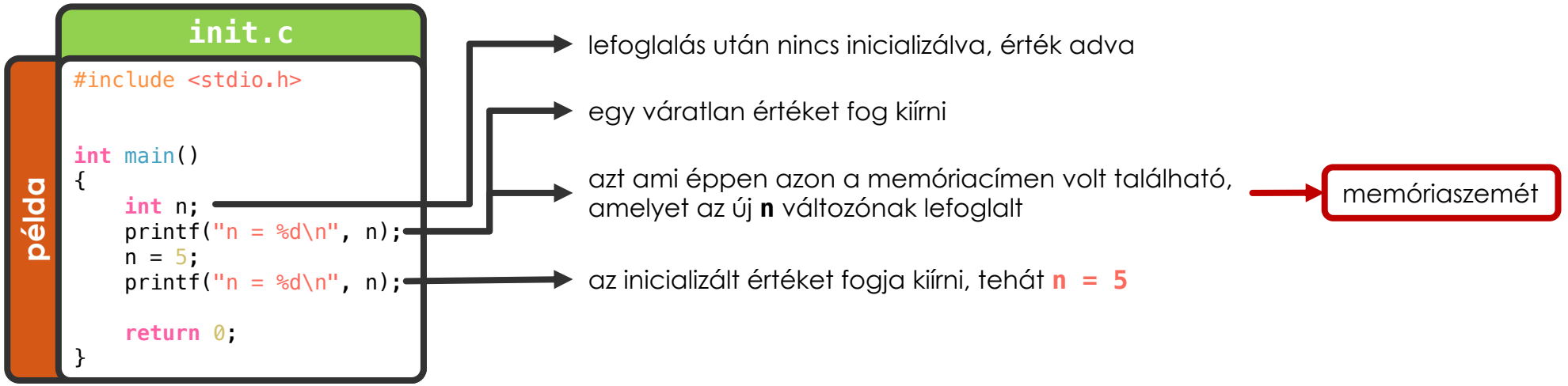
```
char c = 'b';
short sh = 256;
int a = 5, b;
unsigned int u = 1234567;
long l = 300;
float f = 59.65;
double d = 5.75, g;
```

```
b = a + sh;
a = sh - c;
g = d + f;
f = l + u;
```

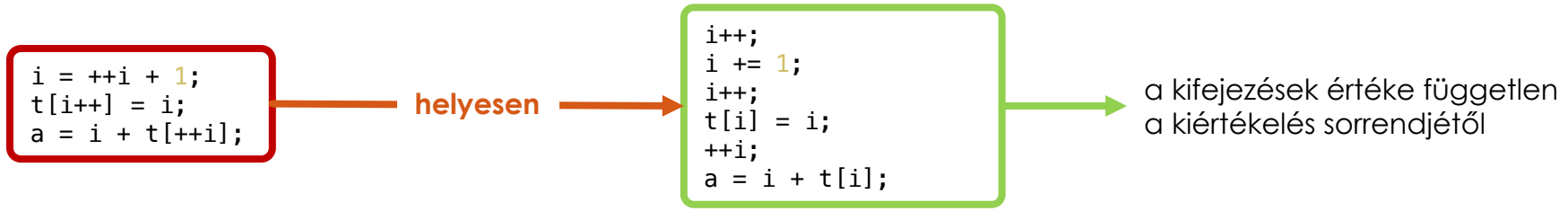
- az **sh** értéke **`int`** típusúvá alakul
- az **c** és **sh** értéke **`int`** típusúvá alakul
- az **f** értéke **`double`** típusúvá alakul
- az **u** értéke **`long`** típusúvá alakul, az eredmény pedig **`float`** lesz

Szabályok és javaslatok

- a lokális változókat implicit inicializálni kell az első használat előtt



- a kifejezések eredménye mindig független kell legyen az operátorok kiértékelési sorrendjétől



Szabályok és javaslatok

- használjunk zárójelezést az összetett kifejezések felírásában
  - a C nyelvben rengeteg operátor van és a precedencia szabályok nem mindig egyértelműek
  - megfelelő zárójelezés segít elkerülni a kifejezések helytelen kiértékelését, elősegíti a kód könnyebb megértését és esetleges későbbi javítását/átírását

példa az **x** változó utolsó bitjének ellenőrzése

az **==** operátornak nagyobb a precedenciája, mint a **&** operátornak

```
x & 1 == 0
```

helyesen

```
(x & 1) == 0
```

a zárójelek biztosítják, hogy megfelelő sorrendben értékelődnek ki a kifejezések

```
x & (1 == 0)
```

```
x & 0
```

3742 3152

kvíz 4. kérdés

KIVÉTEL

az aritmetikai operátorok minden esetben követik az operátorok által jól ismert kiértékelési sorrendet

További részletek
bibliográfia

- K. N. King C programming – A modern approach, 2nd edition, W. W. Norton & Co., 2008
  - 4. fejezet
- Deitel & Deitel C How to Program, 6th edition, Pearson, 2009
  - 2., 3. és 4. fejezet
- CERT – Secure Coding, Rules for expressions
  - https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152200

3742 3152

kvíz 5. kérdés