Programozási nyelvek 1

Szathmáry László
Debreceni Egyetem
Informatikai Kar

3. előadás

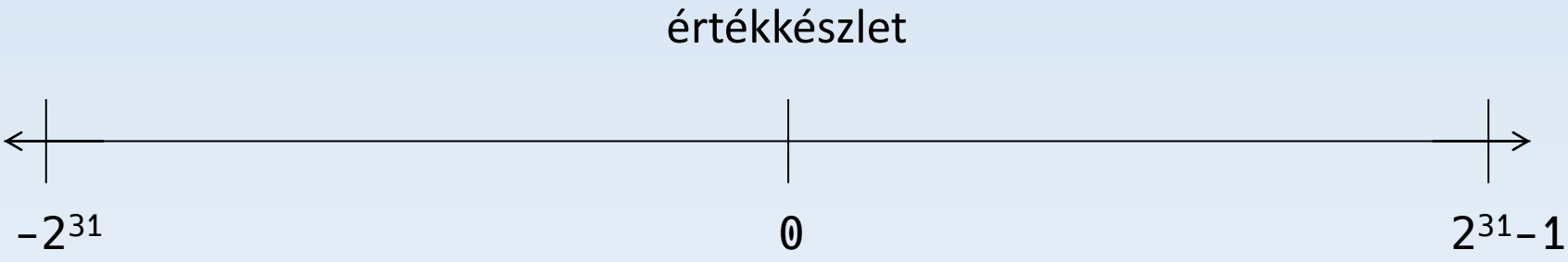
- I. gyakran használt adattípusok, változók
- II. operátorok, logikai kifejezések
- III. feltételes utasítások
- IV. ciklusok

2023-2024, 2. félév

(utolsó módosítás: 2024. jan. 31.)

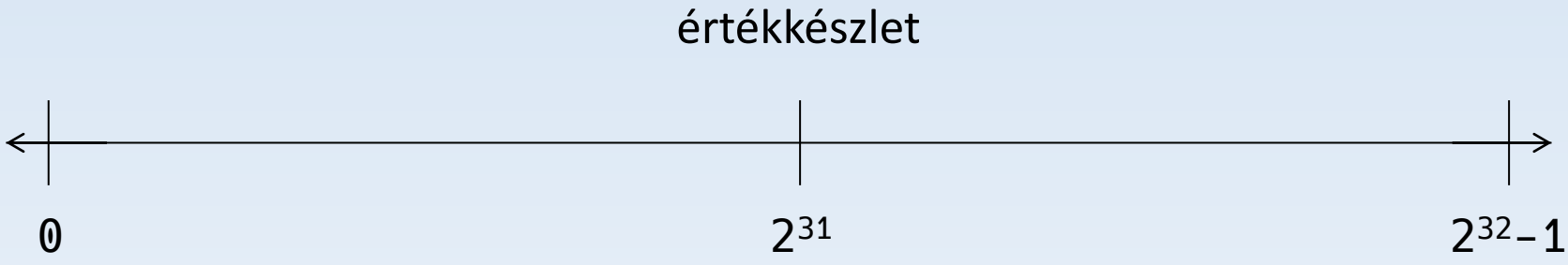
I. Adattípusok

- `int`
  - Az `int` típusú változókban egész értékeket tudunk tárolni. Előjeles, vagyis pozitív és negatív értéket is felvehet.
  - Egy `int` típusú változó mérete 4 byte (32 bit). A memóriában ennyi helyet foglal. Ez a méret (32 bit) meghatározza, hogy mi az a legnagyobb / legkisebb szám, amit tárolni tud.



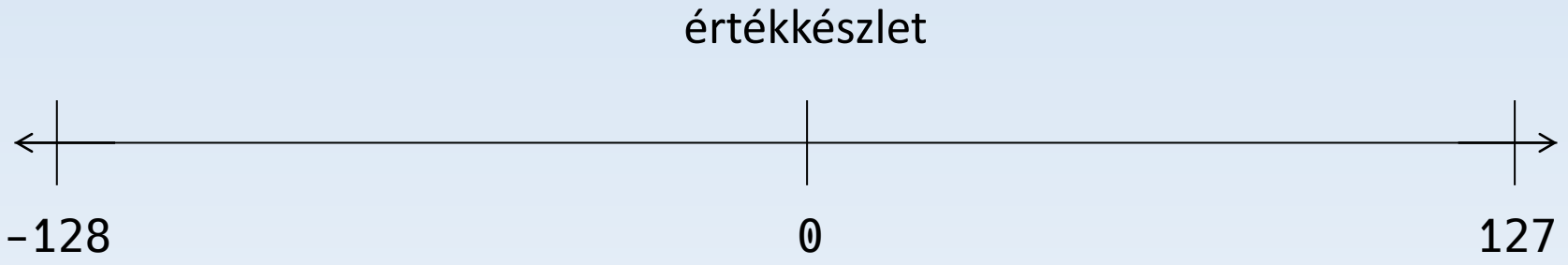
Adattípusok

- `unsigned int`
  - Az `unsigned` egy *módosító jelző*, ami alkalmazható bizonyos típusokra. Lényegében megduplázza a pozitív értékkészletet. Ennek az az ára, hogy negatív értéket nem vehet fel. Jelentése: előjel nélküli.
  - Ritkábban van rá szükség.



Adattípusok

- char
  - Egy char típusú változóban egyetlen karaktert tudunk tárolni.
  - Egy char típusú változó mérete 1 byte (8 bit). A memóriában ennyi helyet foglal.
  - Az ASCII táblázat minden karakterhez hozzárendel egy egész értéket. A táblázat eleje (0-tól 127-ig) minden karakterkészlet esetén azonos.



ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Parancssorból:

```
man ascii
```

- 0-tól 31-ig a nem nyomtatható karakterek szerepelnek
- 32-től a nyomtatható karakterek szerepelnek

implicit típuskonverzió

```c
5     char c = 'A';
6
7     printf("%c\n", c);
8
9     printf("%c, %d\n", c, c);
10
11    printf("%c, %d\n", c, (int)c);
12
13    // -----
14
15    int code = 65;
16
17    printf("%c\n", code);
```

explicit típuskonverzió

Kimenet:

```
A
A, 65
A, 65
A
```

Házi feladat:

ASCII táblázat (link)

Adattípusok

- `float`
  - Egy `float` típusú változóban valós (lebegőpontos) értéket tudunk tárolni, pl. `3.14` .
  - Egy lebegőpontos érték onnan ismerhető fel könnyen, hogy van benne tizedespont (magyarul: tizedesvessző), azaz van egészrésze és törtrésze.
  - Egy `float` típusú változó mérete 4 byte (32 bit). A memóriában ennyi helyet foglal.
  - A 32 biten tárolni kell az egészrészt és a törtrészt is. Itt bejön a pontosság kérdése.
    - Pl. ha túl nagy az egészrész, kevesebb hely marad a törtrészre.
    - Pl. a pí esetén a végtelen sok tizedesjegyből (`3.14159…`) csak valamennyit tudunk tárolni 32 biten.

Adattípusok

- `double`
  - Egy `double` típusú változóban valós (lebegőpontos) értéket tudunk tárolni, pl. 3.14 .
  - Hasonló a `float` típushoz. A különbség, hogy 4 byte helyett ez 8 byte-os (64 bites).
  - A `double` típus dupla pontosságú. Egy lebegőpontos számot nagyobb pontossággal tud tárolni.

Típusok

- `void`
  - Ez egy típus, de nem *adattípus*.
  - Függvények rendelkezhetnek ilyen visszatérési típussal. Ennek a jelentése: a függvény nem ad vissza semmilyen értéket. (Az ilyen függvényt eljárásnak is nevezzük).

Extra típus(ok)

- `string`
  - Láttuk a leggyakrabban használt 5 típust.
  - A tárgyhoz készült egy *library* (függvénykönyvtár), ennek a neve prog1 (letöltés innen). Ebben van egy extra típus, amit `string` –nek neveztünk el.
  - Egy string típusú változóban egy szöveget (karaktersorozatot) tudunk tárolni.
  - A használatához szükség lesz a `prog1.h` és `prog1.c` fájlokra, ill. a forráskódunk elején tüntessük fel az `#include "prog1.h"` sort!
  - Később mi is fogunk készíteni saját típusokat, lásd rekord (`struct`) és típusdefiníció (`typedef`).

Változók

- változó létrehozása
  - Egy változó létrehozásához elegendő megadni a típusát és a nevét. Ezt *deklaráció*nak nevezzük.

```
int counter;
char c;
```

  - Létrehozhatunk egyszerre több azonos típusú változót is.

```
int x, y, z, width, height;
double pi, e;
```

  - Tipp: elegendő ott deklarálni egy változót, ahol szükség van rá.

Változók

- változó használata
  - Miután deklaráltunk egy változót, a későbbi használat során már nem kell megadni a típusát.

```
int counter;              // deklaráció
counter = 0;              // értékadás
char c;                   // deklaráció
c = 'Y';                  // értékadás
```

  - A deklarálást és az értékadást össze is lehet vonni egyetlen lépésbe (ezt *inicializálás*nak is nevezik).

```
int counter = 0;          // inicializálás
char c = 'Y';             // inicializálás
```

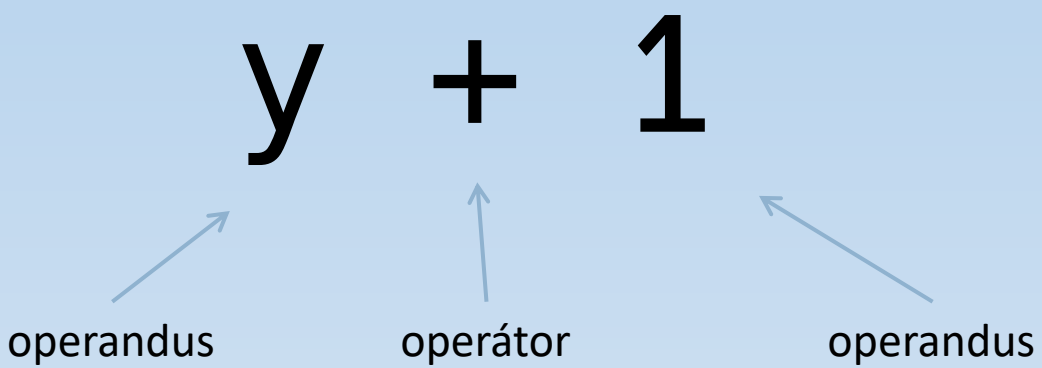
II. Aritmetikai operátorok

- Számokkal a következő matematikai műveleteket tudjuk elvégezni: összeadás (**+**), kivonás (**-**), szorzás (*****), osztás (**/**).

```
int x = y + 1;
x = 2 * x;
```

- A modulus operátor (**%**) az egészosztás maradékát adja.

```
int m = 14 % 3;      // 2
```



kétoperandusú operátorok

```
+
-
*
/
%
```

```
int result = x + 2;
```

egyoperandusú operátorok

```
+
-
```

```
int x = 2;

x = -x;
```

egész osztása egésszel

```
int x;

x = 5 / 2;      // 2

x = 7 / 4;      // 1
```

- valós osztása valóssal
- egész osztása valóssal (vagy fordítva)

```
double result;

result = 5.0 / 2.0;    // 2.5

result = 5 / 2.0;      // 2.5

result = 5.0 / 2;      // 2.5
```

Típuskonverzió

Ha egy operátor operandusai különböző típusúak, akkor a művelet végrehajtása előtt azokat egy közös típusra kell hozni. Az átalakításra van néhány szabály (lásd később). Általában az automatikus (implicit) konverzió csak akkor jön létre, ha egy „keskenyebb” operandust egy „szélesebbre” kell alakítani, mivel így nem lép fel információvesztés. Jelen esetben az egész fog lebegőpontossá alakulni.

Aritmetikai operátorok

- Az aritmetikai operátorok használatának van egy egyszerűbb módja:

```
x = x * 5;
x *= 5;
```

- A fenti rövidebb verzió mind az öt aritmetikai operátorral működik.

- Az 1-gyel való inkrementálásra (növelésre) és dekrementálásra (csökkentésre) van egy még egyszerűbb módszer:

```
++x;
x++;

--x;
x--;
```

Logikai kifejezések

- A logikai kifejezések értékek összehasonlítására szolgálnak.
- A logikai kifejezések kiértékelésekor két lehetséges eredmény jöhet ki: **igaz** vagy **hamis**.
- Hol használjuk fel a logikai kifejezéseket?
  - feltételes utasításoknál (pl.: ha ez a feltétel igaz, akkor ezen az ágon megyek tovább)
  - ciklusoknál (pl. amíg egy feltétel igaz, addig végrehajtok bizonyos műveleteket)
- C-ben nincs igazi logikai (bool) típus.
  A `0` hamisnak számít, minden `0`-tól különböző érték pedig igaznak.
  Ha logikai igaz / hamis értékű változót akarunk létrehozni, akkor a `0` (hamis) és `1` (igaz) értékeket szokás használni.

Logikai kifejezések

- logikai operátorok
  - A logikai ÉS (&&) akkor igaz, ha mindkét operandusa igaz, különben hamis.

x	y	(x && y)
0	0	0
0	1	0
1	0	0
1	1	1

Logikai kifejezések

- logikai operátorok
  - A logikai VAGY (||) akkor igaz, ha (legalább) az egyik operandusa igaz, különben hamis.

x	y	(x || y)
0	0	0
0	1	1
1	0	1
1	1	1

Rövidzár kiértékelés

Az && és a || operátorokkal összekapcsolt kifejezések kiértékelése balról jobbra történik, és a kiértékelés azonnal félbeszakad, ha az eredmény igaz vagy hamis volta ismertté válik.

`x && y`

Ha x hamis, akkor y-t felesleges kiértékelni,
hiszen az egész kifejezés hamis lesz.

`x || y`

Ha x igaz, akkor y-t felesleges kiértékelni,
hiszen az egész kifejezés igaz lesz.

Logikai kifejezések

- logikai operátorok
  - A logikai NEM (`!`) negálja az operandusa logikai értékét. Az igaz értéket hamissá, a hamis értéket igazzá alakítja.

x	!x
0	1
1	0

Ez egy unáris (egyoperandusú) operátor.

Logikai kifejezések

- összehasonlító (relációs) operátorok
  - Értékek összehasonlítására szolgál.
    - kisebb (`x < y`)
    - kisebb vagy egyenlő (`x <= y`)
    - nagyobb (`x > y`)
    - nagyobb vagy egyenlő (`x >= y`)

Logikai kifejezések

- összehasonlító (relációs) operátorok
  - A két operandus értéke egyenlő-e vagy sem.
    - egyenlő (`x == y`)
    - nem egyenlő (`x != y`)
  - Vigyázat! Ne keverjük össze az értékadó operátort (=) az egyenlő operátorral (==)!

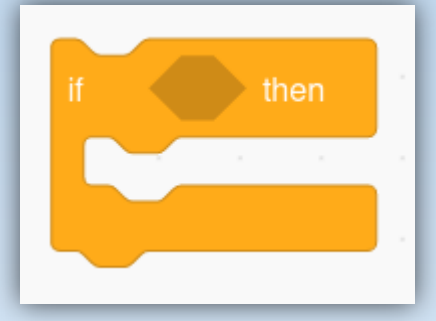
III. Feltételes utasítások

- A feltételes kifejezések elágazásokat tesznek lehetővé a programunkban. Egy kifejezés értékétől függően más-más ágon folytatódhat a program futása.

- A C nyelv többféle feltételes utasítást biztosít.

Feltételes utasítások

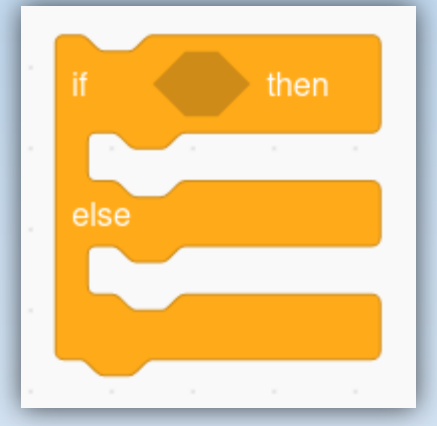
```
if (logikai-kifejezés)
{

}
```



- Ha a `logikai-kifejezés` igaz, akkor a kapcsos zárójelek közötti sorok (*blokk*) le fognak futni.
- Ha a `logikai-kifejezés` hamis, akkor a kapcsos zárójelek közötti sorok **nem** hajtódnak végre.

Feltételes utasítások

```
if (logikai-kifejezés)
{
    // 1. ág
}
else
{
    // 2. ág
}
```



- Ha a `logikai-kifejezés` igaz, akkor az első ágban lévő sorok hajtódnak végre.
- Ha a `logikai-kifejezés` hamis, akkor a második ágban lévő sorok hajtódnak végre.

Feltételes utasítások

```
if (logikai-kif1)
{
    // 1. ág
}
else if (logikai-kif2)
{
    // 2. ág
}
else if (logikai-kif3)
{
    // 3. ág
}
else
{
    // 4. ág
}
```

- C-ben össze lehet kapcsolni több `if-else` részt egyetlen utasítássá.
  - Scratch-ben ezt csak egymásba ágyazással lehetett megoldani.
- Csak az egyik ág fog lefutni! Az ágak kölcsönösen kizárják egymást.
- Tetszőleges számú `else if` ág adható meg.

Feltételes utasítások

```
if (logikai-kif1)
{
    // 1. ág
}
if (logikai-kif2)
{
    // 2. ág
}
if (logikai-kif3)
{
    // 3. ág
}
else
{
    // 4. ág
}
```

- Ebben a példában csak a 3. és a 4. ág zárja ki egymást kölcsönösen.
- Ez a kód nehezebben olvasható. Ilyenkor érdemes tagolást alkalmazni (lásd köv. oldal).

Feltételes utasítások

```
if (logikai-kif1)
{
    // 1. ág
}
```

```
if (logikai-kif2)
{
    // 2. ág
}
```

```
if (logikai-kif3)
{
    // 3. ág
}
else
{
    // 4. ág
}
```

Feltételes utasítások

```
10      int x = get_int();
11      switch (x)
12      {
13          case 1:
14              printf("elso\n");
15              break;
16          case 2:
17              printf("masodik\n");
18              break;
19          case 3:
20              printf("harmadik\n");
21              break;
22          default:
23              printf("nincs helyezes\n");
24              break;
25      }
```

- Nem minden nyelvben létezik.
  Pl. Python-ban nincs switch utasítás.

- A `switch` utasítás is a többirányú programelágaztatás egyik eszköze.
- Mindegyik `case` ágban logikai kifejezések helyett diszkrét értékeket lehet használni.
- A `case` ágak végén fontos a `break;` utasítás! Ennek hiányában a vezérlés „átfolyik” a köv. `case` ágra (hacsak nem ezt akarjuk).
- A `default` ág opcionális.

Feltételes utasítások

```
10      int x = get_int();
11      switch (x)
12      {
13          case 5:
14              printf("5\n");
15          case 4:
16              printf("4\n");
17          case 3:
18              printf("3\n");
19          case 2:
20              printf("2\n");
21          case 1:
22              printf("1\n");
23          default:
24              printf("Start!\n");
25      }
```

- Ebben a példában szándékosan elhagytuk a `break;` utasításokat.
- Inkább **NE** használjuk ezt a trükköt! A kód nehezebben értelmezhető, nagyobb a hiba esélye.

- Tipp: legyen minden `case` ág végén `break;` utasítás.
  A `default` ág végére is nyugodtan betehetjük a `break;` utasítást.

Feltételes utasítások

```
int x;
if (kif)
{
    x = 5;
}
else
{
    x = 6;
}
```

```
int x = (kif) ? 5 : 6;
```

- A fenti két kód ekvivalens.
- A (`?:`) neve: ternáris operátor (mivel *három* operandusa van).
- Nem muszáj használni, de nagyon rövidke if-else utasítások kiválthatók vele (mint itt a példában is).

Feltételes utasítások

`if, if-else, if-else if-…-else`

- Logikai kifejezéseket használunk a döntésekhez.

`switch`

- Diszkrét értékeket használunk a döntésekhez.

`?:`

- Nagyon egyszerű if-else helyett használható.

Feladat:

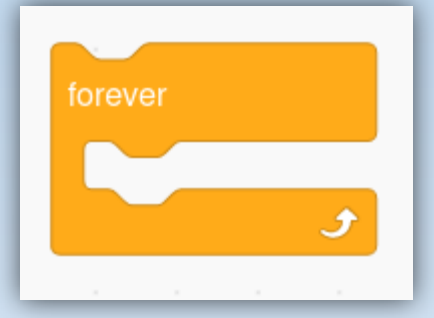
A 30. o.-on lévő kódot (`switch`) írjuk át `if-else if-…-else` konstrukcióra.

IV. Ciklusok

- A program bizonyos sorait többször is végre tudjuk hajtani.
- Többféle ciklust is használhatunk.

Ciklusok

```
while (1)
{

}
```

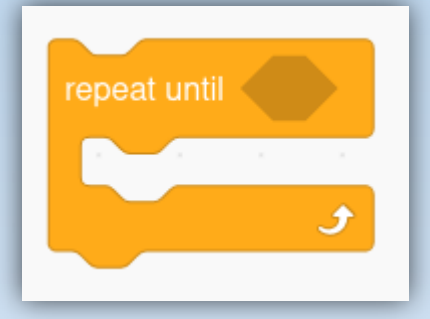


- Ez egy ún. *végtelen ciklus*. A kapcsos zárójelek közti programsorok végrehajtása a végtelenségig ismétlődik.

- Hogy lehet belőle kitörni?
  - A `break;` utasítás használatával.
  - A program leállításával (pl. Ctrl + C).

Ciklusok

```
while (logikai-kifejezés)
{

}
```



- Ha a `logikai-kifejezés` igaz, akkor végrehajtódnak a kapcsos zárójelek közti programsorok (*ciklusmag*), majd újra kiértékelődik a `logikai-kifejezés`. Ha igaz, ismét végrehajtódik a ciklusmag, stb. Ez addig ismétlődik, amíg a `logikai-kifejezés` igaz.
  Ha a `logikai-kifejezés` hamis, akkor befejeződik a ciklus. A vezérlés a ciklus utáni utasításra kerül.

- C-ben a `logikai-kifejezés`ben azt fogalmazzuk meg, hogy mikor ismétlődjön a ciklus. Scratch-ben azt kell megfogalmazni, hogy mikor lépjünk ki a ciklusból.

Ciklusok

```
do
{

}
while (logikai-kifejezés);
```

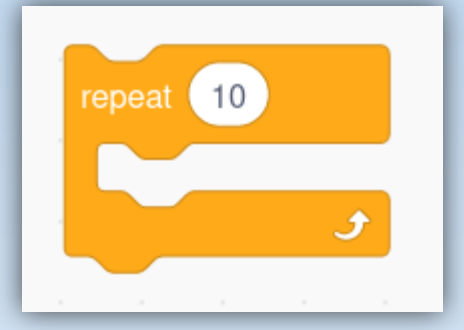
- A kapcsos zárójelek közti programsorok egyszer biztosan lefutnak! Ezután kiértékelődik a `logikai-kifejezés`. Ha igaz, akkor újra lefut a ciklusmag. Megint kiértékeli a `logikai-kifejezés`t, stb. Ha a `logikai-kifejezés` hamis, akkor kilép a ciklusból.

- Még egyszer: a `do-while` ciklus *legalább egyszer* lefut!

- Nincs jelen minden programozási nyelvben.
  Pl. Python-ban nincs `do-while` ciklus.

Ciklusok

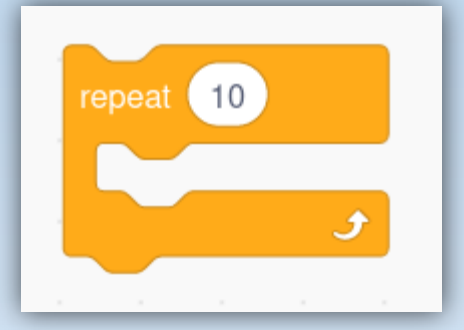
```
for (int i = 0; i < 10; ++i)
{

}
```



- Ez egy ún. *előírt lépésszámú ciklus*. Akkor használjuk, amikor előre tudjuk, hogy hányszor akarjuk a ciklust lefuttatni.
  A példában a ciklusmag pontosan 10-szer fog lefutni.
- Működése:
  - A ciklusváltozót (itt: `i`) inicializáljuk (deklaráljuk + kezdőértéket adunk neki).
  - Kiértékelődik a logikai kifejezés.
    - Ha igaz, akkor lefut a ciklusmag, megnöveljük a ciklusváltozó értékét, majd újra kiértékeljük a logikai kifejezést, stb.
    - Ha hamis, akkor kilépünk a ciklusból.

Ciklusok

```
for (start; kifejezés; növelés)
{

}
```



- Ez egy ún. *előírt lépésszámú ciklus*. Akkor használjuk, amikor előre tudjuk, hogy hányszor akarjuk a ciklust lefuttatni.
  A példában a ciklusmag pontosan 10-szer fog lefutni.
- Működése:
  - `start` végrehajtódik
  - `kifejezés` kiértékelődik
    - ha igaz, akkor lefut a ciklusmag, `növelés` lefut, majd `kifejezés` újra kiértékelődik, stb.
    - ha hamis, akkor kilépünk a ciklusból

Ciklusok

`while`

- Van egy feltételünk, és amíg ez igaz, addig végre akarunk hajtani bizonyos műveleteket. Nem tudjuk, hogy pontosan hányszor fog lefutni a ciklus. Lehet, hogy egyszer sem.
- A végtelen ciklus is gyakran előfordul. Sokszor a ciklusmag belsejében vizsgálunk meg egy feltételt, s ha igaz, akkor belülről kilépünk a ciklusból (`break`).

`do-while`

- Nem tudjuk, hogy pontosan hányszor fut le, de egyszer mindenképpen le akarjuk futtatni. Példa: felhasználói input bekérése.
- Ritkábban használatos.

`for`

- Bizonyos műveleteket meghatározott alkalommal akarunk végrehajtani. Elképzelhető, hogy fordítási időben még nem tudjuk, hogy pontosan hányszor fog lefutni a ciklus.
- Nagyon sokat használjuk.

A break és continue utasítások

- `break`
  - A `break` hatására kilépünk az adott ciklusból, azaz befejeződik a ciklus.
  - Használatos a `switch` utasítás esetén is. Hatására befejeződik a teljes `switch` utasítás.
- `continue`
  - A ciklusmagban található `continue` utasítás hatására azonnal (a ciklusmagból még hátralévő utasításokat figyelmen kívül hagyva) megkezdődik a következő iterációs lépés.
  - Csak ciklusokban alkalmazható.
  - A `break` utasításnál ritkábban szoktuk használni.

Házi feladat

- A K & R-féle „C Bibliában” nézzék át azokat a részeket, amikről szó volt az előadáson.
- Juhász István jegyzetéből nézzék át azokat a fogalmakat, amikről szó volt az előadáson (link).
- Scratch projekt feltöltése a GitHub-ra (link, feladat leírása az oldal alján).

Szorgalmi

- Haladjanak tovább a Linux operációs rendszerrel. Lejátszási lista: http://bit.ly/31pRf7A . Megtekintendő videók: 11, 12, 13, 14, 15.
- Olvassuk el: A copy-paste atyja, Larry Tesler