A C++ programozási nyelv középiskolásoknak

Szerkesztette:
Pánczél István

2015. augusztus

Tartalom

Előszó 4
1. Történeti áttekintés 5
2. Bevezetés 5
3. A C++ nyelv alapelemei 7
a) A C++ nyelv jelkészlete 7
b) A C++ nyelv azonosítói 7
c) Konstansok 8
d) Megjegyzések 9
e) Operátorok és írásjelek 10
4. C++ program szerkezete 10
a) Egyetlen modulból felépülő C++ program 10
b) Több modulból álló C++ program 11
5. A C++ alaptípusai, változók, konstansok 11
a) A C++ nyelv típusai 11
b) Egyszerű változók definiálása 12
c) Konstansok a C++ nyelvben 12
d) Értékek 12
6. Operátorok és kifejezések 12
7. A C++ nyelv utasításai 14
a) Szabványos be- és kimenet utasítások 15
b) Az if utasítás 18
c) Az if-else szerkezet 19
d) Egymásba ágyazott if utasítások 20
e) Az else-if szerkezet 20
f) A switch utasítás 21
g) A ciklus utasítások 23
h) A break, continue és a return utasítások 25
i) Definíciók bevitele az utasításokba 26
8. Származtatott adattípusok 26
a) Egydimenziós tömbök 26
b) Vektor 30
c) Többdimenziós tömb 37
d) Sztringek 39
e) A String osztály 40
f) A struktúra típus 47
9. Függvények 48
10. Fájlkezelés 53

Függelék 56
Programozási tételek 56
1) Az összegzés tétele 56
2) Az eldöntés tétele 57
3) A kiválasztás tétele 58
4) A megszámlálás tétele 59
5) A lineáris keresés tétele 60
6) A maximum és minimum érték keresés tétele 61
7) A kiválogatás tétele 62
8) A közvetlen kiválasztásos rendezés tétele 63

Előszó

Ez a dokumentum nem más, mint a középiskolás diákokat segítő jegyzet. Az itt szereplő információk különböző forrásokból származnak, a forráslista lejjebb látható. Megpróbáltam összeszedni, kigyűjteni azokat az alapvető ismereteket, melyek szükségesek a programozás alapjainak megismeréséhez, illetve az emelt szintű informatika érettségi teljesítéséhez.

Úgy gondoltam, hogy a C++ nyelv a legalkalmasabb erre a célra. Bár nem az egyik legkönnyebben tanulható programozási nyelv, azonban kellően magas szintű nyelv ahhoz, hogy nagy projektekben is használható legyen, ugyanakkor kellően alacsony szintű is, hogy lássuk egy kicsit az alapokat. A C++ nyelv továbbra is ott van a világon legjobban használt programozási nyelvei között, ez vitathatatlan. Az ebben a nyelvben szerzett tudással, sokkal könnyebben tanulható utána bármely más magasabb szintű nyelv. Az elmélet mellett konkrét, gyakorlati példákkal szemléltetem az aktuális ismeretanyagot.

A jegyzet végén, a függelék részben még több gyakorlati feladat látható, melyet ajánlatos önállóan is végigcsinálni. Az alapvető programozási tételeken kívül az eddigi emelt szintű informatika érettségi feladatainak megoldása is megtalálható C++ nyelven. Természetesen saját feladatot is kitalálhat bárki magának, és azt megoldhatja. Ezen kívül vannak informatika feladatgyűjtemények is, melyeket szintén érdemes áttanulmányozni és a bennük lévő feladatokat megoldani.

Még egyszer hangsúlyoznám, hogy ez a jegyzet nem az én önálló munkám, a C++ nyelvet nem én alkottam meg, és nem is fejlesztem. Egyszerűen csak a diákoknak próbálok segíteni azzal, hogy a különböző papír alapú, illetve elektronikus forrásokból összeszedtem és rendszereztem azokat az alapvető programozási ismereteket, melyek az emelt szintű informatika érettségihez kellenek. Tehát itt csak az alapokat ismerhetjük meg, ha valaki programozó szeretne lenni, akkor bizony mélyen bele kell merülni az általam is felhasznált szakirodalmakba, és a matematikával is barátságban kell lenni.

A programozó a 21. század kiemelt „mestersége”, a tudás elsajátítása nagy szorgalmat, kitartást igényel, de a legfontosabb a mindennapi gyakorlás, programozói munka. Persze először el kell végezni megfelelő iskolákat, attól függően, hogy ki milyen szintre szeretne eljutni ezen a területen (OKJ-s képzés, főiskola, egyetem, BSc képzés, MSc képzés). Ezzel persze még közel sem lett senki programozó, főleg jó programozó. Mi kell még? Hát sokévi programozói gyakorlat (minimum 8 – 10 év), állandó naprakész tudás, fogékonyság az új dolgok iránt, folyamatos szakmai fejlődés.

Sok sikert mindenkinek, aki ezt a jegyzetet elolvassa, és ezt az irányt választja.

A felhasznált irodalom:

Bjarne Stroustrup *– A C++ programozási nyelv*
Benkő Tiborné – Poppe András *– Együtt könnyebb a programozás Objektum-orientált C++*
Tóth Bertalan *– ANSI C++ összefoglaló*

1. Történeti áttekintés

A C++ nyelv kidolgozása az AT&T Bell Laboratóriumoknál dolgozó *Bjarne Stroustrop* nevéhez fűződik. Mint ahogy ismeretes a C nyelvet szintén itt fejlesztették ki a 70-es évek elején. Így nem kell csodálkozni azon, hogy a tíz évvel későbbi C++ fejlesztés a C nyelvre épült. A C nyelv ismerete ezért teljesen természetes kiindulópont a C++ nyelv megismeréséhez. *Bjarne Stroustrop* két fő szempontot tartott szem előtt a C++ kidolgozásánál:

- A C++ nyelv legyen felülről kompatibilis az eredeti C nyelvvel
- A C++ nyelv bővítse ki a C nyelvet a Simula 67 nyelvben használt osztályszerkezettel (class)

Az osztályszerkezet, amely a C nyelv **struct** adatszerkezetére épült, lehetővé tette az objektum-orientált programozás (OOP) megvalósítását.

A C++ nyelv több szakaszban nyerte el mai formáját. A *C++ Version 1.2* változata terjedt el először a világon (1985). A használata során felvetődött problémák és igények figyelembevételével *Bjarne Stroustrop* kidolgozta a *Version 2.0* nyelvdefiníciót (1988). A jelentős változtatások miatt a régi C++ (*1.2*) nyelven írt programok általában csak kisebb-nagyobb javítások után fordíthatók le a 2.0-ás verziót megvalósító fordítóprogrammal. Az évek folyamán a C++ nyelv újabb definíciói (*3.x*) jelentek meg, azonban a lényeges újdonságok két nagy csoportba sorolhatók:

- kivételek (exception) kezelése
- paraméterezett típusok, osztályok és függvények használata (templates, sablonok)

A C nyelvet több lépésben szabványosították. Az első (ANSI) szabvány 1983-ban jelent meg, így alapul szolgálhatott a C++ nyelv megformálásához. A C szabványt 1989-ben revízió alá vették (IOS/IEC), majd 1995-ben kibővítették a széles karakterek (wchar_t) használatának lehetőségével. A C++ szabvány kidolgozásában a ANSI X3J16 és az ISO WG21 bizottságok vettek részt a 90-es évek első felében. Munkájuk eredményeként 1997 novemberében megszületett az ANSI/ISO C++ szabvány, melyre a napjainkban használt legtöbb C++ fordítóprogram épül.

Mielőtt elkezdenénk a szabványos C++ nyelv elemeinek bemutatását, le kell szögeznünk, hogy a C++ nyelv a C nyelv szintakszisára épülő önálló programozási nyelv. Alapvető eltérés a két nyelv között, hogy amíg a C csak gyengén típusos nyelv, addig a C++ erősen típusos. Nem csak megengedi, de erősen támogatja is az objektum-orientált programozást.

2. Bevezetés

A C++ egy általános célú programozási nyelv, melynek fő alkalmazási területe a rendszerprogramozás. Támogatja az elvont adatábrázolást, az objektumorientált programozást valamint az általánosított programozást. Az eredeti programozási alapelv középpontjában az eljárás áll (a kívánt számításhoz szükséges `algoritmus`). A nyelvek ezt az alapelvet függvényparaméterek átadásával és a függvények által visszaadott értékekkel támogatják. Az algoritmus nem más, mint egy probléma megoldásának véges számú részlépésben történő egyértelmű, és teljes leírása.

Az évek során a programtervezés súlypontja az eljárások felől az adatszervezés irányába tolódott el. Ez leginkább a programok egyre nagyobb méretében tükröződött. Az egymással rokon eljárásokat az általuk kezelt adatokkal együtt *modul*-nak nevezzük. A moduláris programozási elv már nem elsősorban a függvényekre és azok paramétereinek átadására koncentrál. Az alapelve valahogy így fogalmazható meg: *„Döntsd el, mely modulokra van szükség és oszd fel a programot úgy, hogy az adatokat modulokba helyezd.”*. Ez tulajdonképpen az *adatrejtés* elve. A modularitás alapvető szempont minden sikeres nagy programnál.

A modularitás és az elvont adatábrázolás (saját típusokkal) a jó tervezéshez alapfontosságú. A felhasználói (általunk létrehozott típusok) azonban önmagukban nem elég rugalmasak ahhoz, hogy kiszolgálják igényeinket.

A C++ erősen támogatja az objektumorientált programozást is (OOP). Ez a programozási paradigma ma már szinte egyeduralkodó. Az OOP-t használva már elég jól lehet tervezni ahhoz, hogy nagyméretű szoftvereket is képesek legyünk jól megírni. Az objektumorientáltságról bővebben ebben a jegyzetben nem lesz szó.

A C++ ezek alapján egy multi-paradigma programozási nyelvnek mondható, mivel az előbb említett paradigmákat mind alkalmazhatjuk.

A magas szintű programozási nyelvek (így a C++ is) viszonylag gépfüggetlenek, sok utasítással rendelkeznek, és többségük összetett feladatokat képes megvalósítani. Az utasítások általában angol szavak, vagy azok rövidítéseiként adhatók meg. A programozás olyan folyamat, amely során egy feladat megoldását a számítógép számára „érthető" módon írjuk le (*0-ák és 1-ek sorozata*). A programozási nyelv a programozáskor használt leírás nyelve. A program a programozás eredményeként jön létre, utasítások sorozatából épül fel. A forrás program a megírt programszöveg, amelyet valamilyen szövegszerkesztővel hoztunk létre.

Fordító program (`compiler`) a programot a számítógép számára lefordítja, azaz a forrásprogramot egy úgynevezett tárgyprogrammá alakítja. A fordítási egység a nyelvnek az az egysége, ami a fordítóprogram egyszeri lefuttatásával, a program többi részétől elkülönülten lefordítható, így nem kell mindig a teljes programot újrafordítani. Egy fordítási egység több fájlból is állhat. A fordítás során a fordítás lépései sorrendben az előfordítás, fordítás és a linkelés. Ekkor a forráskódból a gép számára értelmezhető gépi kód keletkezik. Az egyes lépések nem különülnek el feltétlenül egymástól, tehát például az előfordító nem generál le egy különálló fájlt, hanem a kimenetét azonnal átadja a fordítónak.

Az előfordító (`preprocessor`) különböző szöveges változtatásokat hajt végre a forráskódon, előkészíti azt a tényleges fordításra. Feladatai a következők:

- header (fejléc) fájlok beszúrása
- a forrásfájlban fizikailag több sorban elhelyezkedő forráskód logikailag egy sorba történő csoportosítása (ha szükséges)
- kommentek helyettesítése whitespace karakterekkel
- az előfordítónak a programozó által megadott feladatok végrehajtása

Header file – a fejállományok típusokat, valamint a változók és függvények deklarációit tartalmazzák.
Whitespace karakter – a program szövegét ún. whitespace karakterek tördelik szét. Ilyennek számít maga a szóköz, a tabulátor, és a sorvége jel. A programozási nyelvek szintaktikája általában úgy fogalmaz, hogy ahol állhat egy whitespace karakter, ott állhat több is.

Az előfordító (`preprocessor`) a tényleges fordítóprogram futása előtt szövegesen átalakítja a forráskódot. A kettős kereszttel (#) kezdődő sorok az előfordítónak szóló utasítások. Például:

```
#include <iostream> - fájl beillesztés
```

A C++ a kényelmes és hatékony input/output kezelés érdekében úgynevezett adatfolyamokat (`streameket`) használ. Tehát egységes módon kezelhetjük a kiírást és beolvasást képernyőre, billentyűzetre, fájlokba vagy bármilyen más I/O eszközre. Alapvető adatfolyam objektumok:

iostream	I/O utasításokat tartalmaz a kiíratáshoz
fstream	megkönnyíti a fájlműveleteket

A C++ programozási nyelvben egy új szolgáltatás áll a programozók rendelkezésére nagy rendszerek készítéséhez, ez a `névtér`. A névtér egy hierarchikus programobjektum-azonosító tárolási struktúra. Azaz, a logikailag összetartozó deklarációk csoportosítása. A programobjektumok elhelyezése a névterekben egyszerű. A keresési útvonal programozható, megadása a `using` utasítással történik. A megkezdett névtér folytatható, a folytatások száma nincs korlátozva. Példa a standard névtér használatára:

```
using namespace std
```

A C++ nyelvben a szabványos I/O műveletek végzésére használt `cin` és `cout` adatfolyam (*stream*) objektumok nem részei a C++ nyelv definíciójának. A C++ szabvány minden könyvtári elemet egy közös `std` (*standard*) névterületen definiál. A szabványos outputként a `cout`, a szabványos input elvégzésére pedig a `cin` adatfolyam-objektumot használjuk. A << és >> jelek az adatfolyam irányát jelzik. A szöveg kiírása `""` között történik. Pl.

`cout << "Elso programunk"`	a szöveg a kimenetre (képernyőre) kerül
`cin >> n`	a konzolról (billentyűzet) az `n` változóba (lásd később)

A program írásakor megjegyzéseket is elhelyezhetünk, melyeket nem vesz figyelembe a fordító. Kétféle módon tehetjük meg ezeket:

/* */ között többsoros megjegyzés elhelyezése
// használatával pedig a sor végéig terjedő megjegyzést adhatunk meg

```
// egysoros megjegyzés
/* többsoros
megjegyzés*/
```

3. A C++ nyelv alapelemei

Az alapelemek nevek, számok és karakterek, melyekből a C++ program felépül. Ezeket az elemeket `tokennek` nevezzük. A C++ forrásprogram fordításakor a fordítóprogram a nyelv tokenjeit dolgozza fel (a tokeneket a fordítóprogram már nem bontja további részekre). A C++ nyelv alapelemeihez tartoznak a kulcsszavak, az azonosítók, a konstansok, a sztringliterálok, az operátorok és az írásjelek (lásd később).

a) A C++ nyelv jelkészlete

A szabványos C++ program készítésekor kétféle jelkészlettel dolgozunk. Az első jelkészlet azokat a karaktereket tartalmazza, amelyekkel a C++ programot megírjuk:

A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z				
a	b	c	d	e	f	g	h	i	j
k	l	m	n	o	p	q	r	s	t
u	v	w	x	y	z				
!	"	#	%	&	'	(	)	*	+
,	-	/	:	;	<	=	>	?	[
\	]	^	_	{	|	}	~		

A nem látható karakterek közül ide tartoznak még a szóköz, a vízszintes és függőleges tabulátor, a soremelés és a lapdobás karakterek is, melyek feladata a forrásszöveg tagolása. (ezeket a karaktereket összefoglaló néven white-space karaktereknek hívjuk.) Azok a karakterek (ANSI, UniCode) melyeket nem tartalmaz a C++ nyelv karakterkészlete, szintén szerepelhetnek a programban, de csak megjegyzések és sztringliterálok (szöveg konstansok) belsejében.

b) A C++ nyelv azonosítói

A C++ nyelvű program bizonyos összetevőire (pl. változókra, függvényekre, címkékre,...) névvel hivatkozunk. A nevek megfelelő megválasztása lényeges része a program írásának. Az azonosítók hossza megvalósítás függő, a legtöbb fordító legfeljebb 32 karakteres nevek használatát támogatja. Az azonosító első karaktere betű vagy „_” (aláhúzás jel) lehet, míg a második karaktertől kezdődően betűk, számok és aláhúzás jelek válthatják egymást. Az azonosítók elején az aláhúzás jel általában a rendszer által használt, illetve a C++ nyelv bővítését jelentő nevekben szerepel. A legtöbb programozási nyelvtől eltérően a C++ nyelv az azonosítókban megkülönbözteti a kis- és a nagybetűket. Ezért a következő nevek egymástól függetlenül használhatók a programban (nem azonosak): `alma, Alma, ALMA`

Elterjedt konvenció, hogy kisbetűvel írjuk a C++ azonosítókat és csupa nagybetűvel az előfordító által használt neveket. Például:

```
byte, TRUE, FALSE
```

Az értelmes szavakból összeállított azonosítókban az egyes szavakat általában nagybetűvel kezdjük:

```
FelsoSarok, XKoordinata
```

Bizonyos azonosítók speciális jelentést hordoznak. Ezeket a neveket foglalt szavaknak vagy kulcsszavaknak nevezzük. A foglalt szavakat a programban csak a hozzájuk rendelt értelmezésnek megfelelően lehet használni. A kulcsszavakat nem lehet átdefiniálni, új jelentéssel ellátni. Az alábbi táblázatban összefoglaltuk az ANSI C++ nyelv kulcsszavait:

asm	do	inline	return	try
auto	double	int	short	typedef
bool	dynamic_cast	long	signed	typeid
break	else	mutable	sizeof	typename
case	enum	namespace	static	union
catch	explicit	new	static_cast	unsigned
char	extern	operator	struct	using
class	false	private	switch	virtual
const	float	protected	template	void
const_cast	for	public	this	volatile
continue	friend	register	throw	wchar_t
default	goto	reinterpret_cast	true	while
delete	if			

A legtöbb fordítóprogram kibővíti a szabványos kulcsszavakat saját jelentéssel bíró szavakkal. Erre a C++ szabvány a két aláhúzás jel használatát javasolja, például:

```
__try, __property, __published
```

c) Konstansok

A C++ nyelv megkülönbözteti a numerikus és a szöveges konstansértékeket. A konstansok alatt mindig valamiféle számot értünk, míg a szöveges konstansokat sztringliterálnak hívjuk. A konstans értékek ilyen megkülönböztetését a tárolási és felhasználási módjuk indokolja. A C++ nyelvben karakteres, logikai, egész, felsorolt és lebegőpontos konstansokat használhatunk.

A C++ nyelv logikai konstansai az igaz értéket képviselő `true` és a hamis értékű `false`. A nyelv egyetlen mutató konstanssal rendelkezik a nullával (0), melyet gyakran a NULL szimbólummal jelölünk.

Egész konstansok – az egész konstansok számjegyek sorozatából állnak. A számjegyek decimális (10-es), oktális (8-as) vagy hexadecimális (16-os) számrendszerbeli jegyek lehetnek. Az egész konstansok, amennyiben nem előzi meg őket negatív (-) előjel, pozitív értékeket jelölnek.

Karakter konstansok – az ANSI (egybájtos) karakter konstansok egyszeres idézőjelek (' - aposztróf) közé zárt egy karaktert tartalmazó konstansok.Pl. 'a', '2', '@'

Bizonyos szabványos vezérlő- és speciális karakterek megadására az ún. *escape* szekvenciákat használhatjuk. Az *escape* szekvenciában a fordított osztásjel (*backslash* - \) karaktert speciális karakterek, illetve számok követik, mint ahogy az a következő táblázatból is látható.

Értelmezés	ASCII karakter	Escape szekvencia
csengő	BELL	'\a'
visszatörlés	BS	'\b'
lapdobás	FF	'\f'
újsor	NL (LF)	'\n'
kocsi-vissza	CR	'\r'
vízszintes tabulálás	HT	'\t'
függőleges tabulálás	VT	'\v'
aposztróf	'	'\''
idézőjel	"	'\"'
backslash	\	'\\'
kérdőjel	?	'\?'

Lebegőpontos konstansok – a lebegőpontos konstans olyan decimális szám, amely előjeles valós számot reprezentál. A valós szám általában egész részből, tizedes törtrészből és kitevőből tevődik össze. Az egész- és törtrészt tizedespont (.) kapcsolja össze, míg a kitevő (10 hatványkitevője) az **e**, vagy az **E** betűt követi.

Pl. `.1, -2., 100.45, 2e-3, 11E2, -3.1415`

A C++ nyelvben a lebegőpontos értékek a tárolásukhoz szükséges memóriaterület méretétől függően - ami a tárolt valós szám pontosságát és nagyságrendjét egyaránt meghatározza - lehetnek egyszeres (**float**), kétszeres (**double**) vagy nagy (**long double**) pontosságú számok. A lebegőpontos konstansok alaphelyzetben dupla pontosságú értékek. Vannak esetek, amikor megelégszünk egyszeres pontosságú műveletekkel is, ehhez azonban a konstansokat is egyszeres pontosságúként kell megadni a számot követő **f** vagy **F** betűk felhasználásával.

Pl. `3.1415F, 2.7182f`

Nagy pontosságú számítások elvégzéséhez nagy pontosságú lebegőpontos konstansokat kell definiálnunk az **l** (kis L) vagy az **L** betű segítségével.

Pl. `3.14159265358979323851, 2.718281828459045235411`

Sztringkonstansok (literálok) – az ANSI sztringliterál, amit sztringkonstansnak is szokás hívni, kettős idézőjelek közé zárt karaktersorozatot jelent.

Pl. `"Ez egy sztring konstans"`

A megadott karaktersorozatot a statikus memóriaterületen helyezi el a fordító, és ugyancsak eltárolja a sztringet záró '\0' karaktert (nullás byte-ot) is. A sztringkonstans tartalmazhat escape szekvenciákat is.

Pl. `"\nElso sor!\Masodik sor!\n"`

d) Megjegyzések

A megjegyzések olyan karaktersorozatok, melyek elhelyezésének célja, hogy a program forráskódja jól dokumentált, ezáltal egyszerűen értelmezhető, jól olvasható legyen. A C++ nyelvben a megjegyzések programban történő elhelyezésére /* ... */ jeleken kívül a // (két perjel) is használható. A // jel használata esetén a megjegyzést nem kell lezárni, hatása a sor végéig terjed.

Pl.
```
//egysoros megjegyzés
/* többsoros megjegyzés első sora
második sora
harmadik sora*/
```

e) Operátorok és írásjelek

Az operátorok olyan egy vagy több karakterből álló szimbólumok, melyek megmondják, hogyan kell feldolgozni az operandusokat. A következő táblázatban minden magyarázat nélkül felsoroljuk a C++ nyelv (szabványos) operátorait.

!	!=	%	%=	&	&&	&=	()	*	*=
+	++	+=	,	-	--	-=	->	.	/
/=	<	<=	<<	<<=	=	==	>	>=	>>
>>=	?:	[]	^	^=	sizeof	|	|=	||	~
::	.*	->*	new	delete					

Az írásjelek a C++ nyelvben olyan szimbólumokat jelölnek, amelyeknek csak szintaktikai szerepe van. Az írásjeleket általában azonosítók elkülönítésére, a programkód egyes részeinek kijelölésére használjuk, és semmilyen műveletet sem definiálnak. Néhány írásjel egyben operátor is.

Írásjel	**Az írásjel szerepe**
[]	Tömb kijelölése, méretének megadása
()	A paraméter-és az argumentum lista kijelölése
{}	Kódblokk vagy függvény behatárolása
*	A mutató típus jelölése a deklarációban
,	A függvény argumentumok elválasztása
:	Címke elválasztása
;	Az utasítás végének jelölése
…	Változó hosszúságú argumentumlista jelölése
#	Előfordító direktíva jelölése

4. C++ program szerkezete

A C++ nyelven megírt program egy vagy több forrásfájlban (fordítási egységben, modulban) helyezkedik el, melyek kiterjesztése általában `.cpp`. A programhoz általában ún. deklarációs (*include, header, fej-*) állományok is csatlakoznak, melyeket az `#include` előfordító utasítás segítségével építünk be a forrásállományokba.

a) Egyetlen modulból felépülő C++ program

A C++ program fordításhoz szükséges deklarációkat a `main()` függvénnyel azonos forrásfájlban, de tetszőleges számú más forrásállományban is elhelyezhetjük. A `main()` függvény kitüntetett szerepe abban áll, hogy kijelöli a program belépési pontját, vagyis a program futása a `main()` függvény indításával kezdődik. Ezért érthető az a megkötés, hogy minden C++ programnak tartalmaznia kell egy `main()` nevű függvényt, de csak egy példányban. A legegyszerűbb C++ program egyetlen üres main() függvény:

```
int main()
{
 return 0;
}
```

int	a függvény típusa
main	a függvény neve
()	a függvénynek nincs paramétere
return 0;	a függvény törzse, az utasításokat pontosvesszővel kell lezárni
0	visszatérési érték
{}	a függvény törzsét befoglaló zárójelek

b) Több modulból álló C++ program

A C++ nyelv tartalmaz eszközöket a moduláris programozás elvének megvalósításához. A moduláris programozás lényege, hogy minden modul önálló fordítási egységet képez, melyeknél érvényesül az adatrejtés elve. Mivel a modulok külön-külön lefordíthatók, nagy program fejlesztése, javítása esetén nem szükséges minden modult újrafordítani. Ez a megoldás jelentősen csökkenti a futtatható program előállításának idejét.

A futtatható fájl előállításakor minden modult külön le kell fordítanunk. A keletkező tárgymodulokat a szerkesztő (*linker*) építi össze a megfelelő könyvtári hivatkozások feloldásával. Általában elmondható, hogy a C++ forrásprogram előfordító utasítások, deklarációk és definíciók, utasításblokkok és függvények kollekciója.

5. A C++ alaptípusai, változók, konstansok

A C++ nyelvben minden felhasznált névről meg kell mondanunk, hogy mi az, és hogy mire szeretnénk használni. E nélkül a fordító általában nem tud mit kezdeni az adott névvel. A C++ nyelv szóhasználatával élve mindig deklarálnunk kell az általunk alkalmazni kívánt neveket. A deklaráció (leírás) során csak a név tulajdonságait (típus, tárolási osztály, láthatóság stb.) közöljük a fordítóval. Ha azonban azt szeretnénk, hogy az adott deklarációnak megfelelő memóriafoglalás is végbemenjen, definíciót kell használnunk. A definíció tehát olyan deklaráció, amely helyfoglalással jár. Ugyanazt a nevet többször is deklarálhatjuk, azonban az egymás követő deklarációknak egyezniük kell. Ezzel szemben valamely név definíciója csak egyetlenegyszer szerepelhet a programban. A fordító számára a deklaráció (definíció) során közölt egyik legfontosabb információ a típus.

a) A C++ nyelv típusai

A C++ nyelv típusait többféleképpen csoportosíthatjuk. Egyik szempont szerint beszélhetünk alaptípusokról és származtatott típusokról.

- *Alaptípusok* – karakter (*char*), egész (*int*) és lebegőpontos (*double*) típus
- *Származtatott típusok* – az alaptípusok felhasználásával felépített tömb, függvény, mutató, osztály, struktúra és unió típus
- *Void típus* – nem meghatározott típus

Az alaptípusokhoz tartozó *char, int* és *double* típuselőírásokhoz bizonyos más kulcsszavakat (típusmódosítókat) kapcsolva, újabb típuselőírásokhoz jutunk, amelyek értelmezése eltér a kiindulási előírástól. A típusmódosítók a hatásukat kétféle módon fejtik ki. A **short** és a **long** módosítók a tárolási hosszat, míg a **signed** és az **unsigned** az előjel értelmezését szabályozzák.

A típus egy név vagy kifejezés megfelelő használatát határozza meg. A C++ több alaptípussal rendelkezik, melyek közvetlen megfelelői bizonyos hardverszolgáltatásoknak.

Név	**Típus**	**Lehetséges értékek**
bool (logikai)	logikai	true/false (igaz/hamis)
char (karakter)	karakter	Pl.: 'a', 'b', …, 'Z', ...
int (egész)	egész	Pl.: 1, 2, 16, 314, ...
short int vagy short (rövid egész)	egész	Pl.: 1, 2, 16, 314, ...
long int vagy long (hosszú egész)	egész	Pl.: 1, 2, 16, 314, ...
float (valós)	valós	Pl.: 3,14, …
double (duplapontos valós)	valós	Pl.: 299793,0
long double (hosszú duplapontos valós)	valós	Pl.: 299793,0
wchar_t	szabad karakter	Pl.: s

b) Egyszerű változók definiálása

A C++ program memóriában létrehozott tárolóit névvel látjuk el, hogy tudjunk rájuk hivatkozni. A név segítségével a tárolóhoz értéket rendelhetünk, illetve lekérdezhetjük az eltárolt értéket. A névvel ellátott tárolókat a programozási nyelvekben változónak szokás nevezni.

Pl.
```
int konyv;
int n;
int alma, korte;
int tomb[5];
```

c) Konstansok a C++ nyelvben

A C++ nyelvben többféle módon használhatunk konstansokat. Az első lehetőség a **const** típusminősítő megadását jelenti a változódefinícióban. A változók értéke általában megváltoztatható.

Pl.
```
int a;
a=9;
```

Ha azonban a definícióban szerepel a **const** kulcsszó, a változó „csak olvasható” lesz, vagyis értékét nem lehet közvetlenül megváltoztatni. Ekkor a definícióban kötelező a kezdőérték megadása.

Pl.
```
const int a=15;
```

d) Értékek

A változók általában az értékadás során kapnak értéket, melynek általános alakja:

```
változó = érték;
```

A C++ nyelven az értékadás operátor és a fenti utasítás valójában egy kifejezés, amit a fordítóprogram értékel ki. Az értékadás operátorának bal- és jobb oldalán egyaránt szerepelhetnek kifejezések, melyek azonban lényegileg különböznek egymástól. A baloldalon szereplő kifejezés azt a változót jelöli ki (címzi meg) a memóriában, ahova a jobb oldalon megadott kifejezés értékét be kell tölteni.

A fenti alakból kiindulva a C++ nyelv külön nevet ad a kétfajta kifejezésnek. Annak a kifejezésnek az értéke, amely az egyenlőségjel bal oldalán áll, a balérték (*lvalue*), míg a jobboldalon szereplő kifejezés értéke a jobbérték (*rvalue*). Vegyünk példaként két egyszerű értékadást:

Pl.
```
int a;
a = 12;
a = a + 1;
```

Az első értékadás során az **a** változó, mint balérték szerepel, vagyis a változó címe jelöli ki azt a tárolót, ahova a jobboldalon megadott konstans értéket be kell másolni. A második értékadás során az **a** változó az értékadás mindkét oldalán szerepel. A baloldalon álló **a** ugyancsak a tárolót jelöli ki a memóriában (*lvalue*), míg a jobboldalon álló **a** egy jobbérték kifejezésben szerepel, melynek értékét (13) a fordító meghatározza az értékadás elvégzése előtt.

6. Operátorok és kifejezések

Az eddigiek során gyakran használtunk olyan utasításokat (mint például az értékadás), amelyek pontosvesszővel lezárt kifejezésből álltak. A kifejezések egyetlen operandusból, vagy az operandusok és a műveleti jelek (*operátorok*) kombinációjából épülnek fel. A kifejezés kiértékelése valamilyen érték kiszámításához vezethet, függvényhívást idézhet elő vagy mellékhatást (*side effect*) okozhat. Az esetek többségében a fenti három tevékenység valamilyen kombinációja megy végbe a kifejezések feldolgozása (kiértékelése) során.

Az operandusok a C++ nyelv azon elemei, amelyeken az operátorok fejtik ki hatásukat. Azokat az operandusokat, amelyek nem igényelnek további kiértékelést elsődleges (*primary*) kifejezéseknek nevezzük. Ilye-

nek az azonosítók, a konstans értékek, a sztringliterálok és a zárójelben megadott kifejezések. Hagyományosan az azonosítókhoz soroljuk a függvényhívásokat, valamint a tömbelem- és a struktúra taghivatkozásokat is.

A kifejezések kiértékelése során az operátorok lényeges szerepet játszanak. Az operátorokat több szempont alapján lehet csoportosítani. A csoportosítást elvégezhetjük az operandusok száma szerint. Az egyoperandusú (*unary*) operátorok esetén a kifejezés általános alakja:

```
operator operandus – prefixes alak pl. –a, sizeof(a)
operandus operator – postfixes alak pl. a++
```

Az operátorok többsége két operandussal rendelkezik, ezek a kétoperandusú (*binary*) operátorok:

```
operandus1 operator operandus2 – pl. a+b, a<<2, a+=b
```

A C++ nyelvben egyetlen háromoperandusú operátor, a feltételes operátor használható:

```
a < 0 ? –a : a
```

Az alábbi táblázatban a C++ nyelv műveleteit csoportosítottuk:

Precedencia	Operátor	Rövid leírás	Asszociativitás	Jelölés
1	`::`	Hatókör-feloldás	nincs	`a::b, ::b`

2	`()`
	`[]`
	`->`
	`.`
	`++`
	`--`
		Csoportosítás
		Tömb-elérés
		Mutatón keresztüli tag-elérés
		Objektumon keresztüli tag-elérés
		Posztfix növelés
		Posztfix csökkentés
			Bal	`(a)`
				`a[]`
				`ptr->b()`
				`a.b()`
				`a++`
				`a--`

3	`!`
	`~`
	`++`
	`--`
	`-`
	`+`
	`*`
	`&`
	`(típus)`
	`sizeof`
		Logikai tagadás
		Bitenkénti negálás
		Prefix növelés
		Prefix csökkentés
		Előjel -
		Előjel +
		Dereferálás
		Objektum címe
		Konverzió típusra
		Méret
			Jobb	`!a`
				`~a`
				`++a`
				`--a`
				`-a`
				`+a`
				`*ptr`
				`&a`
				`(típus)a`
				`sizeof(a)`

4	`->*` `.*`	Tagkiválasztás mutatón/objektumon	Bal	`a->*b()` `a.*b()`

5	`*`
	`/`
	`%`
		Szorzás
		Osztás
		Maradékszámítás
			Bal	`a*b`
				`a/b`
				`a%b`

6	`+` `-`	Összeadás Kivonás	Bal	`a+b` `a-b`
7	`<<` `>>`	Bitenkénti eltolás balra Bitenkénti eltolás jobbra	Bal	`a<<const` `a>>const`

8	`<`
	`<=`
	`>`
	`>=`
		Kisebb
		Kisebb-egyenlő
		Nagyobb
		Nagyobb-egyenlő
			Bal	`a<b`
				`a<=b`
				`a>b`
				`a>=b`

9	== !=	Egyenlő Nem egyenlő	Bal	a==b a!=b
10	&	Bitenkénti ÉS	Bal	a&b
11	^	Bitenkénti kizáró VAGY	Bal	a^b
12	|	Bitenkénti megengedő VAGY	Bal	a|b
13	&&	Logikai ÉS	Bal	a&&b
14	||	Logikai(megengedő) VAGY	Bal	a||b
15	?:	if-then-else operátor	Jobb	log. kif. ? kifejezés : kifejezés
16	= += -= *= /= %= &= ^= |= <<= >>=	Értékadás Összeadás és értékadás Kivonás és értékadás Szorzás és értékadás Osztás és értékadás Maradékképzés és értékadás Bitenkénti ÉS és értékadás Bitenkénti kizáró VAGY és értékadás Bitenkénti megengedő VAGY és értékadás Eltolás balra és értékadás Eltolás jobbra és értékadás	Jobb	a=b a+=b a_=b a*=b a/=b a%=b a&=b a^=b a|=b a<<=b a>>=b
17	,	Szekvencia operátor	Bal	a, b

Az operátorok precedenciája (*elsőbbsége*) akkor játszik szerepet a kifejezés kiértékelése során, ha a kifejezésben különböző precedenciájú műveletek szerepelnek. Ekkor mindig a magasabb precedenciával rendelkező operátort tartalmazó részkifejezés kerül először kiértékelésre, amit az alacsonyabb precedenciájú műveletek végrehajtása követ.

Pl.
```
5 + 3 * 4 = 17 – a szorzás erősebb művelet, mint az összeadás
(5 + 3) * 4 = 32 – a zárójelben lévő művelet először
```

Egy kifejezésben több azonos precedenciájú művelet is szerepelhet. Ebben az esetben az operátortáblázat csoportnevei mellett található útmutatást kell figyelembe venni a kiértékelés során, hisz a precedencia szabály nem alkalmazható. Az asszociativitás azt mondja meg, hogy az adott precedenciaszinten található műveleteket balról-jobbra vagy jobbról-balra haladva kell elvégezni.

Az értékadó utasítások csoportjában a kiértékelést jobbról-balra haladva kell elvégezni. Emiatt C++-ban adott a lehetőség több változó együttes inicializálására.

Pl.
```
int a,b,c;
a = b = c = 26;
```

7. A C++ nyelv utasításai

A C++ nyelven megírt program végrehajtható része elvégzendő tevékenységekből (*utasításokból*) épül fel. Az utasítások a strukturált programozás alapelveinek megfelelően **ciklusok**, **programelágazások** és **vezérlésátadások** szervezését teszik lehetővé. A C++ nyelv más nyelvekhez hasonlóan rendelkezik a vezérlésátadás **goto** utasításával, melynek használata nehezen követhetővé teszi a program szövegét. Ezen utasítás használata azonban az esetek többségében elkerülhető a **break** és a **continue** utasítások bevezetésével. A C++ nyelv utasításait hét csoportba sorolhatjuk:

Kifejezés utasítás	
Üres utasítás	;
Összetett utasítás	{}; try{}
Szelekciós utasítások	if, else, switch
Címkézett utasítások	case, default, ugrási címke
Vezérlésátadó utasítások	break, continue, goto, return, throw
Iterációs utasítások (ciklusok)	do, for, while

Tetszőleges kifejezés utasítás lesz, ha pontosvesszőt helyezünk mögé:

```
kifejezés;
```

A *kifejezés utasítás* végrehajtása a kifejezésnek az előzőekben ismertetett szabályok szerint történő kiértékelését jelenti. Mielőtt a következő utasításra kerülne a vezérlés, a teljes kiértékelés végbemegy.

Az *üres utasítás* egyetlen pontosvesszőből áll: `;`

Az üres utasítás használatára akkor van szükség, amikor logikailag nem kívánunk semmilyen tevékenységet végrehajtani, azonban a szintaktikai szabályok szerint a program adott pontján utasításnak kell szerepelnie. Az üres utasítást, melynek végrehajtásakor semmi sem történik, gyakran használjuk a **do**, **for**, **while** és **if** szerkezetekben.

A kapcsos zárójeleket használjuk arra, hogy a logikailag összefüggő deklarációkat és utasításokat egyetlen összetett utasításba vagy blokkba csoportosítsuk. Az összetett utasítás mindenütt felhasználható, ahol egyetlen utasítás megadását engedélyezi a C++ nyelv leírása. Összetett utasítást a következő három esetben használunk:

- Amikor több logikailag összefüggő utasítást egyetlen utasításként kell kezelni (ilyenkor általában csak utasításokat tartalmaz a blokk)
- Függvények törzseként
- Definíciók és deklarációk érvényességének lokalizálására

Az utasításblokkon belül az utasításokat és a definíciókat/deklarációkat tetszőleges sorrendben megadhatjuk. A blokkot nem kell pontosvesszővel lezárni. Az összetett utasítás általános formája:

```
{
lokális definíciók és deklarációk
utasítások
}
```

a) Szabványos be- és kimenet utasítások

Egy program során szükségünk lehet adatok kiírására vagy bekérésére a felhasználótól. A C++ Standard Library része az *iostream* könyvtár. Segítségével minden beépített típusra meghatároz kimenetet (pl. ki tudunk írni szövegeket, egész és lebegőpontos számokat is), de lehetőségünk van saját, általunk létrehozott típusokra is megadni a kimeneteiket. Alapértelmezettként a *cout* –ra kerülő kimenetek karaktersorozatokká alakítódnak át, ami a képernyőre íródik ki.

A `cout` szabványos kimeneti adatfolyam. A „`<<`” operátorral tudunk egyszerűen a képernyőre írni. Ezt felfoghatjuk akár „*tedd bele*” operátorként is.

Pl. `cout << kifejezés1 << kifejezés2;`

A különböző típusú kimenetek természetesen párosíthatók is, továbbá egy kimeneti kifejezés eredménye felhasználható további kimenetekhez is. Ez gyakorlatilag annyit jelent, hogy szépen, láncba köthetjük őket. (ahogy az előbbi példában is látható).

A karakter konstans egy karakter, egyszeres idézőjelek közé zárva. (pl: ’a’). Karakterlánc konstanst is használhatunk, melyeket idézőjelek közé kell zárni (pl.: „Kérem, adjon meg egy egész számot!”).

Az alábbi példában deklaráltunk egy egész, és kettő szöveges típusú változót, majd különböző kiíratásokat végeztünk karakterrel, számmal, szöveggel és a változókkal (előtte behívtuk a string osztályt - `#include <string>`). A 16. sorban láthatjuk, hogy szövegeket össze lehet fűzni a + operátorral. A 17. sorban pedig azt láthatjuk, hogy egy szöveghez hozzá lehet fűzni tetszőleges szöveget a += operátorral. Ezen kívül a `\n` *escape karakterrel* elérjük, hogy a kiírások új sorban kezdődjenek.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main()
6  {
7      int a = 7;
8      string szoveg1 = "Elmegyek a boltba ";
9      string szoveg2 = "es veszek kenyeret.";
10     cout << 'a' << '\n';
11     cout << 5 << '\n';
12     cout << a << '\n';
13     cout << "Alma\n";
14     cout << szoveg1 << '\n';
15     cout << szoveg2 << '\n';
16     cout << szoveg1 + szoveg2 << '\n';
17     szoveg1 += " delutan";
18     cout << szoveg1;
19
20     cout << endl << endl;
21     system("pause");
22     return 0;
23 }
```

Formázott kiírásokat az *escape karakterekkel* tehetünk. Szövegek esetén vízszintes tabulátorokkal, új sorba írással érhetünk el formázott elrendezéseket. Az alábbi példában néhány lehetőséget látunk:

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      cout << "\tAlma\t\t" << "Apple\n";
7      cout << "\tKorte\t\t" << "Pear\n";
8      cout << "\tOszibarack\t" << "Both peach" << endl;
9      cout << "\tSargabarack\t" << "Apricot";
10
11     cout << endl << endl;
12     system("pause");
13     return 0;
14 }
```

A 6. és 7. sorban két tabulátort helyeztünk el, mert a 8. sorban lévő `Both Peach` szöveg hosszabb az alapértelmezett tabulátornál. Azt is észrevehetjük, hogy a 8. sorban nem a `\n` *escape karakterrel* írtunk új sorba, hanem az `endl` (*end line – vége a sornak*) utasítással.

További megjelenítési lehetőségek vannak, ezekhez azonban be kell hívni az *iomanip* osztályt.

```
#include <iomanip>
```

Az alábbi példában néhány egyszerű kiírási megoldást szemléltetünk a teljesség igénye nélkül. A használt manipulátorok a következők:

```
left, right:      balra, jobbra igazítás
setw():           a kiírás szélességének megadása
setfill():        kitöltő karakter megadása
dec, oct, hex:    tízes, nyolcas, tizenhatos számrendszerbeli alak
setprecision:     tizedes jegyek számának megadása
fixed:            tizedes tört alak
scientific:       exponenciális alak (hatványalak)
```

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4
5  int main()
6  {
7      int a = 39;
8      double b = 56.805;
9      cout << left << setw(10) << setfill('*') << "Alma" << endl;
10     cout << right << setw(10) << setfill('x') << "Korte" << endl << endl;
11     cout << "Tizes szamrendszerben:\t\t" << dec << a << endl;
12     cout << "Nyolcas szamrendszerben:\t" << oct << a << endl;
13     cout << "Tizenhatos szamrendszerben:\t" << hex << a << endl << endl;
14     cout << "Tizedes tort alak:\t" << setprecision(6) << fixed << b << endl;
15     cout << "Exponencialis alak:\t" << setprecision(6) << scientific << b;
16     cout << endl << endl;
17     system("pause");
18     return 0;
19 }
```

Beolvasás szabványos bemenetről (*cin*) – a `cin` szabványos bemeneti adatfolyam. A >> („*olvasd be*”) műveleti jelet bemeneti operátorként használjuk. A >> jobb oldalán álló típus határozza meg, milyen bemenet fogadható el, és mi a beolvasó művelet célpontja.

Nézzünk egyszerű példát bevitelekre:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    int a;
    double b;
    char c;
    string szoveg1 = "Alma";
    string szoveg2;

    cout << "Adj meg egy egesz szamot: ";
    cin >> a;
    cout << "A megadott szam: " << a << endl << endl;

    cout << "Adj meg egy tizedes tort szamot: ";
    cin >> b;
    cout << "A megadott szam: " << b << endl << endl;

    cout << "Adj meg egy karaktert: ";
    cin >> c;
    cout << "A megadott karakter: " << c << endl << endl;

    cout << "Adj meg egy szoveget: ";
    cin >> szoveg2;
    cout << "A megadott szoveg: " << szoveg2 << endl << endl;

    cout << "A szoveg1 valtozo tartalma: " << szoveg1;

    cout << endl << endl;
    system("pause");
    return 0;
}
```

b) Az if utasítás

A C++ nyelv két lehetőséget biztosít a program kódjának feltételhez kötött végrehajtására - az **if** és a **switch** utasításokat. Az **if** utasítás segítségével valamely tevékenység (*utasítás*) végrehajtását egy kifejezés (*feltétel*) értékétől tehetjük függővé. Az **if** alábbi formájában az utasítás csak akkor hajtódik végre, ha a kifejezés értéke **nem nulla** *(igaz – true)*.

Fontos megjegyezni, hogy C++ esetén az *if*-ben az értéknek egy számnak kell lennie. 0 esetén hamis, egyéb esetben pedig igaznak értékeli ki. C++ -ban bár létezik logikai érték, a *bool* típus (*boolean*), azonban ez nem valódi logikai érték. A bool true-ja 1-esnek, a false-a pedig 1-nek felel meg. Azért van így a nyelvben, mert a C++ egy, a C-ből eredő alacsonyabbnak számító nyelv. Magasabb szintű nyelvekben (például Java, C#, stb.) létezik külön logikai, *boolean* típus. Azoknak az értékei nem feleltethetőek meg számoknak, és egy *if*-ben szereplő kifejezés csak és kizárólag logikai típus lehet, különben fordítási idejű hibát kapunk. C++ -ban ezzel ellentétben akármi lehet a kifejezés értéke az *if*-ben, hiszen mindent egy számként értékel ki. Így a kifejezés értéke lehet egy ’a’ karakter is. Ez esetben kikeresi az ’a’ ASCII

kódját, és azt vizsgálja, hogy nulla-e. Mint azt az előbb megbeszéltük, nulla esetén hamis, egyéb esetben pedig igaz.

Nézzünk egy egyszerű példát az if használatára:

```
#include <iostream>
using namespace std;

int main()
{
    int a;
    cout << "Adj meg egy egesz szamot: ";
    cin >> a;
    cout << endl;
    if (a > 30)
    {
        cout << "A beirt egesz szam nagyobb 30-nal.";
    }

    cout << endl << endl;
    system("pause");
    return 0;
}
```

c) Az if-else szerkezet

Az **if** utasítás teljes formájában, amely tartalmazza az **else** ágat, arra az esetre is megadhatunk egy tevékenységet (*utasítás2*), amikor a kifejezés (*feltétel*) értéke hamis *(false*). Ha az `utasítás1` és az `utasítás2` nem összetett utasítások, akkor pontosvesszővel kell őket lezárni.

```
if (kifejezés)
  utasítás1;
else
  utasítás2;
```

Nézzünk erre a szerkezetre is egy egyszerű példát.

```
#include <iostream>
using namespace std;

int main()
{
    int a;
    cout << "Adj meg egy egesz szamot: ";
    cin >> a;
    if (a > 30)
        cout << "A szam nagyobb 30-nal.";
    else
        cout << "A szam NEM nagyobb 30-nal.";

    cout << endl << endl;
    system("pause");
    return 0;
}
```

d) Egymásba ágyazott if utasítások

Az **if** utasítások egymásba is ágyazhatók. Ilyenkor azonban óvatosan kell eljárnunk, hisz a fordító nem mindig úgy értelmezi az utasítást, ahogy mi gondoljuk (a fordító minden **if** utasításhoz a hozzá legközelebb eső **else** utasítást rendeli). Egy járható út, ha a belső **if** utasítást kapcsos zárójelek közé, azaz utasítás blokkba helyezzük.

Nézzünk erre is egy egyszerű példát. Ha a megadott szám 30-nál nagyobb, akkor megnézzük még azt is, hogy páros szám, vagy nem. Ezt a maradékos osztással (%) tudjuk megállapítani (==, egyenlő), azaz, ha 2-vel osztva a maradék nulla, akkor páros számról van szó. Ha a szám nem nagyobb 30-nál, akkor további vizsgálat nincs, kiírjuk, hogy a szám nem nagyobb 30-nál.

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a;
7      cout << "Adj meg egy egesz szamot: ";
8      cin >> a;
9      if (a > 30)
10     {
11         if (a % 2 == 0)
12             cout << "\nA szam 30-nal nagyobb paros.\n";
13         else
14             cout << "\nA szam 30-nal nagyobb, nem paros.\n";
15     }
16     else
17         cout << "\nA szam 30-nal nem nagyobb\n";
18
19     cout << endl << endl;
20     system("pause");
21     return 0;
22 }
```

e) Az else-if szerkezet

Az egymásba ágyazott **if** utasítások gyakran használt formája, amikor az **else** ágakban szerepel az újabb **if** utasítás. Ezzel a szerkezettel a program **többirányú elágaztatását** valósíthatjuk meg. Ha bármelyik kifejezés igaz, akkor a hozzákapcsolt utasítás kerül végrehajtásra. Amennyiben egyik feltétel sem teljesült, a program végrehajtása az utolsó **else** utasítással folytatódik.

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a;
7      cout << "Adj meg egy szamot: ";
8      cin >> a;
9      if (a <= 30)
10         cout << "\nA szam 30-nal nem nagyobb\n";
11     else if (a % 2 == 0)
12         cout << "\nA szam 30-nal nagyobb paros.\n";
13     else
14         cout << "\nA szam 30-nal nagyobb NEM paros.\n";
15
16     cout << endl << endl;
17     system("pause");
18     return 0;
19 }
```

f) A switch utasítás

A switch utasítás többirányú programelágaztatást tesz lehetővé olyan esetekben, amikor egy egész kifejezés értékét több konstans értékkel kell összehasonlítanunk. Az utasítás általános alakja:

```
switch (kifejezés)
{
case konstans_kifejezés:
        utasítások
case konstans_kifejezés:
        utasítások
default:
        utasítások
}
```

A **switch** utasítás először kiértékeli a kifejezést, majd átadja a vezérlést arra a **case** címkére (esetre), amelyben a *konstans_ kifejezés* értéke megegyezik a kiértékelt kifejezés értékével és a program futása ettől a ponttól folytatódik. Amennyiben egyik **case** konstans sem egyezik meg a kifejezés értékével, a program futása a **default** címkével megjelölt utasítástól folytatódik. Ha nem használunk **default** címkét, akkor a vezérlés a **switch** utasítás blokkját záró kapcsos zárójel utáni utasításra adódik.

Amikor a **case** vagy a **default** címkével belépünk a **switch** utasítás törzsébe, akkor attól a ponttól kezdve a megadott utasítások sorban végrehajtódnak (a blokkot záró zárójel eléréséig). Általában azonban az adott esethez tartozó programrészlet végrehajtása után a **goto**, a **break** vagy a **return** utasítással kilépünk a **switch** utasításból. Amennyiben a **switch** utasítás után álló utasítással kívánjuk folytatni a program futását, akkor a **break** utasítást használjuk.

Nézzünk példát a **switch** utasításra. A programban két példát is láthatunk, az első esetben a beírt egész számtól függően 3 különböző felirat kerül a szabványos *outputra* (képernyőre). A második részben pedig azt láthatjuk, ahogy az általunk megadott valamelyik alapműveleti jel (*karakter*) szerint elvégzi a megfelelő műveletet, majd az eredményt kiírja a képernyőre. A **break** utasítással akadályozzuk meg, hogy a program tovább menjen a következő **case** ágra. Ha például a második részben nem lennének ott a **break** utasítások, a program egymás után végrehajtaná mind a négy ágat, azaz, egymás után kiszámolná az eredményeket, és képernyőre írná azokat (*ez is lehetne egy feladat!*).

```cpp
#include <iostream>
using namespace std;

int main()
{
    int a;
    cout << "Adj meg egy egesz szamot 1-tol 3-ig: ";
    cin >> a;
    switch (a)
    {
    case 1: cout << "\n1. feladat\n";
        break;
    case 2: cout << "\n2. feladat\n";
        break;
    default: cout << "\n3. feladat\n";
    }
    cout << "\nSzamologep!\n\n";
    char muv_jel;
    double x, y, e;
    cout << "Milyen muveletet szeretnel vegezni?\n";
    cout << "Irj +, - , * vagy / jelet: ";
    cin >> muv_jel;
    cout << "\nAdd meg az elso szamot: ";
    cin >> x;
    cout << "Add meg a masodik szamot: ";
    cin >> y;
    switch (muv_jel)
    {
    case '+':  e = x + y;
        cout << "\nAz osszeadas eredmenye: " << e << endl;
        break;
    case '-':  e = x - y;
        cout << "\nA kivonas eredmenye: " << e << endl;
        break;
    case '*':  e = x * y;
        cout << "\nA szorzas eredmenye: " << e << endl;
        break;
    case '/':  e = x / y;
        cout << "\nAz osztas eredmenye: " << e << endl;
        break;
    }

    cout << endl << endl;
    system("pause");
    return 0;
}
```

g) A ciklus utasítások

A programozási nyelveken bizonyos utasítások automatikus ismétlését biztosító programszerkezetet **iterációnak** vagy ciklusnak (*loop*) nevezzük. Ez az ismétlés mindaddig tart, amíg az ismétlési feltétel igaznak bizonyul. A C++ nyelv háromféle ciklusutasítást tartalmaz, melyek formája:

```
while (kifejezés) {utasítás}
for (kifejezés) {utasítás}
do {utasítás} while (kifejezés)
```

A ciklusokat csoportosíthatjuk a vezérlőfeltétel kiértékelésének helye alapján. Azokat a ciklusokat, amelyeknél az utasítás végrehajtása előtt kerül feldolgozásra a vezérlőfeltétel, **elől tesztelő** ciklusnak nevezzük. Ezeknél a ciklus soron következő iterációja csak akkor hajtódik végre, ha a feltétel igaz (nem nulla). A **while** és a **for** elől tesztelő ciklusok.

Ezzel szemben a **do** ciklus legalább egyszer mindig lefut, hisz a vezérlőfeltétel ellenőrzése az utasítás végrehajtása után történik, azaz hátul tesztelő ciklus.

Mindhárom ciklusfajta esetén a helyesen szervezett ciklus befejezi a működését, amikor a vezérlőfeltétel hamissá (*nulla*) válik. Vannak esetek azonban, amikor szándékosan vagy véletlenül olyan ciklust hozunk létre, melynek vezérlőfeltétele soha sem lesz hamis. Ezeket a ciklusokat **végtelen** ciklusnak nevezzük.

A ciklusokból a vezérlőfeltétel hamis értékének bekövetkezte előtt is ki lehet ugrani (a végtelen ciklusból is). Erre a célra további utasításokat biztosít a C++ nyelv, mint a **break**, a **return**, illetve a ciklus törzsén kívülre irányuló **goto**. A ciklus törzsének bizonyos utasításait átugorhatjuk a **continue** utasítás felhasználásával. A **continue** hatására a ciklus következő iterációjával folytatódik a program futása.

A while ciklus – a while ciklus mindaddig ismétli a hozzá tartozó utasítást (a ciklus törzsét), amíg a vizsgált kifejezés (vezérlőfeltétel) értéke igaz (nem nulla). A vizsgálat mindig megelőzi az utasítás végrehajtását. A while jól olvasható alakja:

```
while (kifejezés)
      utasítás
```

Nézzünk egy egyszerű példát:

```
#include <iostream>
using namespace std;

int main()
{
    int n, osszeg;
    cout << "A program osszeadja az egesz szamokat a megadott szamig.\n";
    cout << "Meddig adjam ossze a szamokat? Adj meg egy egesz szamot: ";
    cin >> n;
    osszeg = 0;
    while (n > 0)
    {
        osszeg = osszeg + n;
        n = n - 1;
    }
    cout << "\nA szamok osszege: " << osszeg;
    cout << endl;

    cout << endl << endl;
    system("pause");
    return 0;
}
```

A for ciklus – a for utasítást általában akkor használjuk, ha a ciklusmagban megadott utasítást adott számszor kívánjuk végrehajtani. A for utasítás általános alakjában külön megjelöltük az egyes kifejezések szerepét:

```
for (init_kif ; feltétel_kif ; léptető_kif)
```

Tekintsük példaként az előző feladatot, de most for ciklussal.

```
#include <iostream>
using namespace std;

int main()
{
    int n, osszeg;
    cout << "A program osszeadja a szamokat egy megadott egesz szamig.\n";
    cout << "Meddig adjam ossze a szamokat? Add meg az egesz szamot: ";
    cin >> n;
    cout << endl;
    osszeg = 0;
    for (int i = 0; i <= n; i++)
        osszeg = osszeg + i;
    cout << "A szamok osszege: " << osszeg << endl;

    cout << endl << endl;
    system("pause");
    return 0;
}
```

A do-while ciklus – mint ahogy a ciklusok bevezető részében említettük, a do-while utasításban a ciklus törzsét képező utasítás végrehajtása után kerül sor a tesztelésre. Így a ciklus törzse legalább egyszer mindig végrehajtódik. A do-while utasítás használatának jól olvasható formája:

```
do
  utasítás
while (kifejezés);
```

A do-while ciklus futása során mindig az utasítás végrehajtását követi a kifejezés kiértékelése. Amennyiben a kifejezés értéke igaz (*nem 0, true*), akkor új iteráció kezdődik, míg hamis (*0, false*) érték esetén a ciklus befejezi működését.

Példaként tekintsük ismét az előzőekben használt összeadó programot, most do-while ciklussal:

```
#include <iostream>
using namespace std;

int main()
{
    int n, osszeg;
    cout << "A program osszeadja a szamokat egy megadott egesz szamig.\n";
    cout << "Meddig adjam ossze a szamokat? Add meg az egesz szamot: ";
    cin >> n;
    cout << endl;
    osszeg = 0;
    do
    {
        osszeg = osszeg + n;
        n = n - 1;
    } while (n > 0);
    cout << "A szamok osszege: " << osszeg << endl;

    cout << endl << endl;
    system("pause");
    return 0;
}
```

h) A break, continue és a return utasítások

Vannak esetek, amikor egy ciklus szokásos működésébe közvetlenül be kell avatkoznunk. Ilyen feladat például, amikor adott feltétel teljesülése esetén ki kell ugrani a ciklusból, vagy amikor a ciklus végrehajtását a következő iterációval kívánjuk folytatni. Ezen feladatok elvégzésére a legtöbb programozási nyelv a goto utasítás használatára hagyatkozik, azonban a C++ nyelven külön utasítások, a **break** és a **continue** állnak a programozó rendelkezésére.

A break utasítás – a break hatására az utasítást tartalmazó legközelebbi switch, while, for és do-while utasítások működése megszakad és a vezérlés a megszakított művelet utáni első utasításra kerül. Az előzőekben már láttuk a switch utasítás példa programjában.

A continue utasítás – a continue utasítás a while, a for és a do-while ciklusok soron következő iterációját indítja el, a ciklustörzsben a continue után elhelyezkedő utasítások átlépésével. A while és a do-while utasítások esetén a következő iteráció a vezérlőfeltétel ismételt kiértékelésével kezdődik. A for ciklus esetében azonban, a feltétel kifejezés kiértékelését megelőzi a léptető kifejezés feldolgozása.

Nézzünk erre is egy egyszerű példát:

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      cout << "A program 0-tol 99-ig kiirja a 7-el vagy 11-el oszthato szamokat.\n\n";
7      int i;
8      for (i = 0; i < 100; i++)
9      {
10         if (i % 7 && i % 11)
11             continue;
12         cout << i << endl;
13     }
14
15     cout << endl << endl;
16     system("pause");
17     return 0;
18 }
```

Ez a kis program részlet elsőre nem biztos, hogy teljesen érthető. Nézzük meg a működését, valamint vizsgáljuk meg néhány értékre.

A *for* ciklusban csak annyi szerepel, hogy az *i* értékei 0-tól 99-ig mennek (i < 100).

Az *if* értéke két érték „logikai és”-éből áll össze. Így csak akkor lesz az *if* értéke igaz, ha mind a kettő rész értéke igaz. (logikai és: csak akkor igaz, ha mind a kettő igaz). A két rész bal oldala: „i % 7”, jobb oldala pedig: „i % 11”. Őket a „logikai és” köti össze, ami C++-ban a dupla &, azaz: „&&”.

A kifejezésben szereplő % (százalék jel) a maradékos osztás maradékát jelenti. Ezt modulo-nak nevezik. Egy példa: 13 % 5 eredménye: 3, hiszen 13-ban az 5 meg van 2-szer, és maradék a 3.

Az *if* minden egyes *i*-re kiértékeli, hogy igaz, vagy hamis. Nézzük meg 4 esetben: i=6, i=7, i=11, i=13.

Ha az *i* értéke 6, akkor mi történik? Az *if* bal oldala: i % 7, azaz 6 % 7, melynek eredménye: 6, mivel 6-ban a 7 meg van 0-szor, és maradék a 6. Ez egy nullától különböző szám, így értéke igaz. A „logikai és” csak akkor igaz, ha mind a kettő igaz, ezért a másodikat is ki kell értékelnie. Ez az i % 11, azaz 6 % 11, melynek eredménye: 6, azaz igaz. Így a teljes *if* igaz lesz. Ennek következménye az, hogy beleugrik az *if* igaz ágába, azaz a *continue* utasítást végrehajtja. Ennek következménye pedig az, hogy i=6 esetén nem hajtja végre a „`cout << i << endl;`” utasítást, azaz a 6-ot nem írja ki.

Következő eset legyen az, amikor az *i* értéke 7 (azaz a soron következő). Nézzük a kifejezés bal oldalát: i % 7, azaz 7 % 7, melynek eredménye a 0, mivel 7-ben a 7 meg van egyszer, és maradék a nulla. Így az if-ben szereplő kifejezés bal oldala hamis (mivel értéke 0). A „logikai és" értéke csak akkor igaz, ha mindkettő igaz, de mivel az első hamis, ez már nem is teljesülhet, így a második részt már meg sem nézi. Így a teljes *if* értéke hamis, ennek eredménye az, hogy átugorja az *if* igaz ágát, a *continue* utasítást. Mivel azt a sort átugrotta, ezért a „`cout << i << endl;`" utasítás viszont most végrehajtódik, tehát kiírja a képernyőre a 7-es számot.

Nézzük, hogy az i=11 esetben mi történik. 11 % 7 eredménye 4, tehát igaz. Így megnézi a második felét is, ami 11 % 11, aminek eredménye 0, tehát hamis. „Logikai és" esetében az „igaz és hamis" hamisként értékelődik ki, így a teljes if is hamisként értékelődik ki. Ennek következménye az, hogy a *continue*-t átugorja, majd kiírja a 11-et a képernyőre.

i=13 esetben tulajdonképpen ugyanaz történik, mint az i=6 esetben (gondold végig).

Így a program ugyanilyen módon találja meg a 7 vagy a 11 többszöröseit.

A return utasítás – a return utasítás befejezi az éppen futó függvény működését és a vezérlés visszakerül a hívó függvényhez. Ha a **main()** függvényben használjuk a return utasítást, akkor a programunk befejezi a futását, hisz a main() függvény az („*őt hívó*") operációs rendszernek adja vissza a vezérlést.

A return utasítást

```
return ;
```

alakban használva olyan függvényből léphetünk ki, amely a nevével nem ad vissza semmilyen értéket (***void** típusú függvény, illetve **eljárás***). A függvények többsége azonban valamilyen értékkel (a függvényértékkel) tér vissza a hívás helyére, mely értéket a return utasításban definiálhatunk:

```
return kifejezés ;
```

A return utasítással részletesen a függvényeket tárgyaló részben foglalkozunk.

i) Definíciók bevitele az utasításokba

A C++ nyelv megengedi, hogy a változók deklarációját bárhova helyezzük a program kódján belül. Egyetlen feltétel, hogy a változót mindenképpen deklarálnunk (*definiálnunk*) kell a felhasználása előtt. Élve ezzel a lehetőséggel a változó deklarációja (*definíciója*) és a felhasználása közel helyezhető, elkerülve ezzel bizonyos gyakori programozási hibákat.

Az utasításokban definiált változók csak az utasításon belül használhatók. Így ha a ciklusváltozó értékére cikluson kívül is szükségünk van, nem szabad ezt a megoldást használnunk.

8. Származtatott adattípusok

Az eddigi példákban olyan változókat használtunk, amelyek csak egyetlen érték (*skalár*) tárolására alkalmasak. A programozás során azonban gyakran van arra szükség, hogy azonos vagy különböző típusú elemekből álló adathalmazt a memóriában tároljunk, és az adatokon valamilyen műveletet hajtsunk végre. C++ nyelven a **tömbök**, illetve a felhasználói típusok (**struct**, **class**, **union**) segítségével elegánsan megoldhatjuk a fenti problémát.

a) Egydimenziós tömbök

A tömb (*array*) típus olyan adatok halmaza, amelyek azonos típusúak, a memóriában folytonosan helyezkednek el és előre meghatározott számú eleme van. Az elemek elérése a tömb nevét követő idexelés operátorban megadott elemsorszám (*index*) segítségével történik. A tömb tehát a változók olyan készlete, melyekre közös névvel és egy indexszel hivatkozunk.

A leggyakrabban használt tömbtípus egyetlen kiterjedéssel (*dimenzióval*) rendelkezik. Az egydimenziós tömböket **vektornak** is szokás nevezni. Azonban ún. *többdimenziós* tömbök használatára is adott a lehetőség. Kétdimenziós tömbök esetén az elemek tárolása soronként (sorfolytonosan) történik.

Az egydimenziós tömböket definiálnunk kell, melynek általános alakja:

```
típus tömbnév [méret];
```

A definícióban szereplő típus, amely az elemek típusát definiálja, a void és a függvénytípus kivételével tetszőleges típus lehet. A szögletes zárójelek között a tömb méretét adjuk meg. Ez a méret, amelynek a fordító által kiszámítható konstans kifejezésnek kell lennie, a tömbben tárolható elemek számát definiálja.

Pl. `int a[5]`

A fenti tömb egész típusú, és 5 darab elem (egész számok) tárolására alkalmas.

A tömb elemeinek egymás után történő elérésére általában a for ciklust használjuk, melynek változója a tömb indexe:

Pl.
```
for (int i = 0 ; i < 5 ; i++)
        a[i] = i * i;
```

A fenti példában feltöltöttük a tömb elemeit az indexek négyzetével.

A tömb számára a memóriában lefoglalt memóriaterület mérete a **sizeof(a)** kifejezéssel pontosan lekérdezhető, míg a **sizeof(a[0])** kifejezés egyetlen elem méretét adja meg. Így a két kifejezés hányadosából (*egészosztás*) mindig megtudható a tömb elemeinek száma.

Nézzük ezt a gyakorlatban:

```
#include <iostream>
using namespace std;

int main()
{
    int tomb[8];
    for (int i = 0; i < 8; i++)
        tomb[i] = i * i;
    cout << "A tomb szamara lefoglalt memoriaterulet:\t\t" << sizeof(tomb)
        << " byte";
    cout << endl;
    cout << "A tomb egyes elemeinek lefoglalt memoriaterulet:\t"
        << sizeof(tomb[0]) << " byte";
    cout << endl;
    int n = sizeof(tomb) / sizeof(tomb[0]);
    cout << "\nA tomb elemszama: " << n;
    cout << endl;

    system("pause");
    return 0;
}
```

Tekintsünk egy másik példát tömb feltöltésére és az elemeinek kiíratására. A tömb maximum 500 darab egész számból állhat.

```
#include <iostream>
using namespace std;

int main()
{
    int n;
    int a[500];
    cout << "A program megadott meretu tombot feltolt egesz szamokkal.";
    cout << "\nAz egesz szamokat is te adod meg.";
    cout << "\n\nHany elemu legyen a tomb: ";
    cin >> n;
    cout << "\nKerem az egesz szamokat:\n\n";
    for (int i = 0; i < n; i++)
    {
        cout << "A(z) " << i + 1 << ". szam: ";
        cin >> a[i];
    }
    cout << endl;
    cout << "A megadott tomb elemei:\n\n";
    for (int i = 0; i < n; i++)
    {
        cout << a[i] << " ";
    }

    cout << endl;
    system("pause");
    return 0;
}
```

Az előbbi feladat természetesen karakterekkel is működik, íme:

```
#include <iostream>
using namespace std;

int main()
{
    int n;
    char a[500];
    cout << "A program megadott meretu tombot feltolt karakterekkel.";
    cout << "\nA karaktereket is te adod meg.";
    cout << "\n\nHany elemu legyen a tomb: ";
    cin >> n;
    cout << "\nKerem a karaktereket:\n\n";
    for (int i = 0; i < n; i++)
    {
        cout << "A(z) " << i + 1 << ". karakter: ";
        cin >> a[i];
    }
    cout << endl;
    cout << "A megadott tomb elemei:\n\n";
    for (int i = 0; i < n; i++)
    {
        cout << a[i] << " ";
    }
    cout << endl << endl;
    system("pause");
    return 0;
}
```

A C++ nyelv lehetővé teszi, hogy a tömböket a definiálásuk során inicializáljuk. Ez a **kezdőérték adás** eltér az egyszerű változók esetén használt megoldástól.

```
típus tömbnév[méret] = {vesszővel tagolt inicilizációs lista};
```

Pl. `int a[8] = {1, 2, 3, 4, 5, 6, 7, 8};`

`char b[7] = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};`

Tekintsünk erre is egy példát, két tömböt inicializáltunk, majd kiírattuk az elemeiket. Az első tömb elemei egész számok, a második tömb elemei pedig karakterek, mindkét esetben a tömb méretét nyolceleműre állítottuk be.

```
#include <iostream>
using namespace std;

int main()
{
    int a[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
    char b[8] = { 'a', '*', '&', 'c', 'K', 'e', '@', 'P' };
    cout << "Az elso tomb elemei:\n\n";
    for (int i = 0; i < 8; i++)
    {
        cout << a[i] << " ";
    }
    cout << endl;
    cout << "\nA masodik tomb elemei:\n\n";
    for (int i = 0; i < 8; i++)
    {
        cout << b[i] << " ";
    }
    cout << endl << endl;
    system("pause");
    return 0;
}
```

Gyakran szükség van arra, hogy egy tömb elemeit sorrendbe tegyük. Ilyenkor jól jön a `sort` függvény, nézzük ennek használatát:

```
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
    int tomb[10] = { 34, 23, 12, 56, 5, 47, 39, 73, 124, 11 };
    int tomb_elemszama = sizeof(tomb) / sizeof(tomb[0]);
    sort(tomb, tomb + tomb_elemszama);

    for (int i = 0; i < tomb_elemszama; i++)
    {
        cout << tomb[i] << " ";
    }

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Vegyük észre, hogy a `sort` függvény használatához be kell hoznunk az `algorithm` könyvtárat (`#include <algorithm>`). A függvénynek két paramétert kell megadni, hányadik elemtől hányadik elemig rendezze a tömböt. Alapesetben minden elemet rendezni szeretnénk, ekkor első paraméterként a tömbünk nevét adjuk meg, második paraméterként pedig a tömb nevét, majd egy pluszjelet követően a tömbünk hosszát (ahogyan az látszik is az előbbi példában). Azonban megadhatunk ennél kevesebbet is, mondjuk, az első három elemet rendezze. Ilyenkor második paraméterként a következőt kell megadnunk: tömb neve + 3. Azt is megtehetjük, hogy ne az első elemtől kezdődően rendezze az elemeket, hanem például csak az ötödiktől. Ehhez az első paraméter esetén a tömb neve után, szintén egy pluszjelet követően megadjuk az ötöt: tömb neve + 5. A kettőt kombinálni is lehet, tegyük fel, hogy a harmadik elemtől akarunk rendezni a nyolcadik elemig:

```
sort(tömb neve + 3, tömb neve + 8).
```

Vigyázzunk arra, hogy a második paraméternél lévő szám nagyobb legyen az első paraméternél lévő számnál!

b) Vektor

A `vector` típussal egydimenziós tömböket helyettesíthetünk. A lényege, hogy sok azonos típusú elemet egyesítünk a vektor fogalmában egy adategységgé. A vektor a memóriában sorfolytonosan helyezkedik el. Vektorok használatához be kell hozni a `vector` osztályt:

```
#include <vector>
```

Ezután létrehozhatunk vektorokat a következőképpen:

```
vector <típus> név;
```

Az előbbi deklarációval egy üres vektor jött létre megadott névvel. Megadhatjuk deklaráláskor, hogy hány elemű legyen a vektorunk. Ilyenkor az elemeknek nulla értéket ad.

```
vector <típus> név(elemszám);
```

Nézzünk erre egy egyszerű példát:

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main()
6  {
7      vector <int> szamok(10);
8
9      for (int i = 0; i < szamok.size(); i++)
10     {
11         cout << szamok[i] << " ";
12     }
13
14     cout << endl << endl;
15     system("pause");
16     return 0;
17 }
```

Azt is megtehetjük, hogy a vektor létrehozásakor az elemszámon kívül megadjuk az elemeket is. Ebben az esetben a vektor tartalma annyi darab megadott érték lesz, amennyi az elemszám. A következő példában 7 darab 8-as, illetve 12 darab G karakter lesz a vektorok tartalma.

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector <int> szamok(7, 8);

    for (int i = 0; i < szamok.size(); i++)
    {
        cout << szamok[i] << " ";
    }

    cout << endl << endl;

    vector <char> betuk(12, 'G');

    for (int i = 0; i < betuk.size(); i++)
    {
        cout << betuk[i] << " ";
    }

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Ha egy vektort fel szeretnénk gyorsan tölteni véletlen számokkal, akkor használjuk a `rand()` függvényt. Ha a generált számok tartományát is meg akarjuk határozni, akkor a `rand()` függvényünket a következőképpen használjuk (1 és 100 közé eső egész számot generál):

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector <int> szamok(10);
    for (int i = 0; i < 10; i++)
    {
        szamok.at(i) = rand() % 100 + 1;
    }
    for (int i = 0; i < 10; i++)
    {
        cout << szamok.at(i) << " ";
    }
    cout << endl << endl;
    system("pause");
    return 0;
}
```

Ha vektorokkal dolgozunk, használhatjuk a tagfüggvényeit (tömböknél nincsenek ilyenek). A tagfüggvények használatával könnyebben és gyorsabban tudunk dolgozni, ezért igen hasznosak. A tagfüggvények úgy hívhatók elő, hogy a vektor neve után egy pontot írunk, majd a felbukkanó listából kiválasztjuk a számunkra megfelelő függvényt. Tekintsünk most néhányat a teljesség igénye nélkül. Bővebb információt és további tagfüggvényeket a következő címen találhatunk:

http://www.cplusplus.com/reference/vector/vector/?kw=vector

Üres-e a vektor (0 – HAMIS, 1 – IGAZ)

```
vektor_neve.empty();
```

Példa:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector <int> szamok(7, 8);

    cout << szamok.empty();

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Vektor elemeinek száma

```
vektor_neve.size();
```

Példa:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector <int> szamok(7, 8);

    cout << szamok.size();

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Hozzáférés a vektor i. eleméhez

```
vektor_neve.at(i);
```

Példa:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector <int> szamok(10);
    for (int i = 0; i < 10; i++)
    {
        szamok.at(i) = rand() % 100 + 1;
    }

    for (int i = 0; i < szamok.size(); i++)
    {
        cout << szamok[i] << " ";
    }

    cout << endl << endl << szamok.at(3);

    cout << endl << endl;
    system("pause");
    return 0;
}
```

A vektor végéhez hozzáfűz egy megadott elemet

```
vektor_neve.push_back(elem);
```

Példa:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector <int> szamok(10);
    for (int i = 0; i < 10; i++)
    {
        szamok.at(i) = rand() % 100 + 1;
    }

    szamok.push_back(12);

    for (int i = 0; i < szamok.size(); i++)
    {
        cout << szamok[i] << " ";
    }

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Elhagyja a vektor utolsó elemét

```
vektor_neve.pop_back();
```

Példa:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector <int> szamok(10);
    for (int i = 0; i < 10; i++)
    {
        szamok.at(i) = rand() % 100 + 1;
    }

    for (int i = 0; i < szamok.size(); i++)
    {
        cout << szamok[i] << " ";
    }

    szamok.pop_back();

    cout << endl << endl;

    for (int i = 0; i < szamok.size(); i++)
    {
        cout << szamok[i] << " ";
    }

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Megadja a vektor első elemét

```
vektor_neve.front();
```

Példa:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector <int> szamok(10);
    for (int i = 0; i < 10; i++)
    {
        szamok.at(i) = rand() % 100 + 1;
    }

    for (int i = 0; i < szamok.size(); i++)
    {
        cout << szamok[i] << " ";
    }

    cout << endl << endl << szamok.front();

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Megadja a vektor utolsó elemét

```
vektor_neve.back();
```

Példa:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector <int> szamok(10);
    for (int i = 0; i < 10; i++)
    {
        szamok.at(i) = rand() % 100 + 1;
    }

    for (int i = 0; i < szamok.size(); i++)
    {
        cout << szamok[i] << " ";
    }

    cout << endl << endl << szamok.back();

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Törli a vektor minden elemét

```
vektor_neve.clear();
```

Példa:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector <int> szamok(10);
    for (int i = 0; i < 10; i++)
    {
        szamok.at(i) = rand() % 100 + 1;
    }

    for (int i = 0; i < szamok.size(); i++)
    {
        cout << szamok[i] << " ";
    }

    szamok.clear();

    cout << endl << endl;

    cout << szamok.size();

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Természetesen a vektorok elemeit is sorba rendezhetjük, amint azt már a tömböknél láttuk. A szintaktika itt egy kicsit más. Az `algorithm` könyvtárat most is be kell hívni, majd használhatjuk a `sort()` függvényt. Alapesetben két paramétere van. Az első paraméter megadja, hogy melyik elemtől, a második paraméter pedig, hogy melyik elemig hajtsa végre a sorba rendezést. Az alábbi példa 20. sorában látható, hogy az összes elemet rendezni szeretnénk (begin – kezdet, end – vég).

Példa:

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  using namespace std;
5
6  int main()
7  {
8      vector <int> szamok(10);
9
10     for (int i = 0; i < 10; i++)
11     {
12         szamok.at(i) = rand() % 100 + 1;
13     }
14
15     for (int i = 0; i < szamok.size(); i++)
16     {
17         cout << szamok[i] << " ";
18     }
19
20     sort(szamok.begin(), szamok.end());
21
22     cout << endl << endl;
23
24     for (int i = 0; i < szamok.size(); i++)
25     {
26         cout << szamok[i] << " ";
27     }
28
29     cout << endl << endl;
30     system("pause");
31     return 0;
32 }
```

Most is lehetőségünk van arra, hogy megváltoztassuk a rendezni kívánt elemek tartományát. Az első paraméternél egy pluszjel után beírhatunk egy számot, ez jelzi, hogy ettől a helytől rendezi az elemeket (vigyázat, 0-tól számolunk). A második paraméternél egy mínuszjel után tehetjük ugyanezt, visszafelé haladva ezen a helyen lévő elemig rendez (itt is 0-tól számolunk). A következő példában azt láthatjuk, hogy a 10 elemű vektorunk 3. helyén álló szám az első rendezendő elem, és visszafelé számolva a 4. helyen álló szám az utolsó rendezendő elem. Mindkét esetben figyelembe vettem a nullától való számolást: + 2 például azt jelenti valójában, hogy 0. 1. 2. azaz, a harmadik helyről van szó.

Példa:

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    vector <int> szamok(10);

    for (int i = 0; i < 10; i++)
    {
        szamok.at(i) = rand() % 100 + 1;
    }

    for (int i = 0; i < szamok.size(); i++)
    {
        cout << szamok[i] << " ";
    }

    sort(szamok.begin() + 2, szamok.end() - 3);

    cout << endl << endl;

    for (int i = 0; i < szamok.size(); i++)
    {
        cout << szamok[i] << " ";
    }

    cout << endl << endl;
    system("pause");
    return 0;
}
```

c) Többdimenziós tömb

C++-ban tulajdonképpen olyan, hogy többdimenziós tömb nem létezik. Helyette a nyelv a tömbökben lévő tömböket támogatja. Ez alatt mit is értünk?

Az egydimenziós tömbökkel az előbb foglalkoztunk. Egy egyszerű, 10 egész számot tartalmazó tömböt nagyon könnyen fel tudunk venni:

```
int tomb[10];
```

Megjegyzendő, hogy ennek mérete fix, azaz soha nem változhat, valamint fordítási időben foglalódik le a memóriaterület.

Nézzünk egy olyan példát, ahol szükségünk lenne egy többdimenziós tömbre, például, ha mátrixokkal szeretnénk számolni. Egy mátrix igazából nem más, mint „táblázatba rendezett számok”. Lehet őket összeadni és szorozni is. Ennek módja és a mögötte lévő matematika most nem fontos. A lényeg, hogy ezeket valahogyan el szeretnénk tárolni, hogy aztán tudjunk velük dolgozni.

Bár nem létezik többdimenziós tömb, a C++ mégis megengedi, hogy egyszerűen, az egydimenziós tömbökhöz hasonlóan is tudjuk kezelni őket. Ha például egy 10 x 10-es méretű egész számokat tartalmazó „kétdimenziós” tömböt szeretnénk, azt felvehetjük a következő, egyszerű módon:

```
int tomb[10] [10];
```

A tömb elemeihez ugyanúgy hozzáférhetünk, mint ahogyan azt az egydimenziós tömbök esetében tettük, azaz például ha az ötödik sor 4-edik eleméhez szeretnénk hozzáférni, és értékül a 3-as számot szeretnénk adni, az a következő képpen néz ki:

```
tomb[4][5] = 3;
```

Igazából, hogy melyiket vesszük sornak és melyiket oszlopnak teljesen rajtunk múlik. Azonban a balról jobbra történő írás / olvasás a megszokott nálunk, valamint a „következő sor” alatt mi az alatta lévő sort értjük. Így a

```
tomb[4][5]
```

esetében a „4” a sorban negyedik lévő elemre utal, tehát ez az oszlopszám. Az „5” pedig azt jelenti, hogy az ötödik sorban, vagyis ez a sorszámot jelenti.

Vajában a többdimenziós tömböket nem így szoktuk használni, hanem dinamikus memóriakezeléssel, amire egyelőre bővebben nem térnék ki.

Több dimenziós tömb természetesen nem csak 2 dimenziós lehet, hanem akármekkora.

A következő példában két darab 5*5-ös tömböt deklaráltunk, melyek elemei egész számok. A tömböket inicializáltuk, azaz értékeket adtunk az elemeknek, hogy ne kelljen nekünk beírni. Ezután kiírattuk a tömböket, majd egyszerű szorzással kiszámoltattuk a két tömb megfelelő elemeinek szorzatait, majd ezt a szorzat tömböt is képernyőre írattuk.

```
#include <iostream>
using namespace std;

int main()
{
    int a[5][5] = { { 1, 7, 3, 4, 5 }, { 6, 3, 7, 2, 1 }, { 7, 4, 3, 6, 2 },
    { 4, 3, 7, 3, 6 }, { 7, 3, 5, 8, 3 } };
    int b[5][5] = { { 6, 4, 5, 7, 2 }, { 3, 9, 1, 3, 7 }, { 8, 3, 6, 8, 5 },
    { 5, 4, 6, 1, 8 }, { 6, 2, 0, 9, 4 } };
    cout << "A program ket darab 5*5-os egesz szamokbol allo tombot";
    cout << " deklaral.\n";
    cout << "Ezeket a tomboket kiirja a kepernyore, majd osszeszorozza";
    cout << " az elemeiket.\n";
    cout << endl;
    cout << "Az elso tomb elemei:\n\n";
    for (int i = 0; i < 5; i++)
    {
        for (int j = 0; j < 5; j++)
        {
            cout << a[i][j] << "\t";
        }
        cout << endl;
    }
    cout << endl;
    cout << "A masodik tomb elemei:\n\n";
    for (int i = 0; i < 5; i++)
    {
        for (int j = 0; j < 5; j++)
        {
            cout << b[i][j] << "\t";
        }
        cout << endl;
    }
    int c[5][5];
    for (int i = 0; i < 5; i++)
    {
        for (int j = 0; j < 5; j++)
        {
            c[i][j] = a[i][j] * b[i][j];
        }
```

```
41
42      }
43      cout << "\nA szorzat tomb elemei:\n\n";
44      for (int i = 0; i < 5; i++)
45      {
46          for (int j = 0; j < 5; j++)
47          {
48              cout << c[i][j] << "\t";
49          }
50          cout << endl << endl;
51      }
52      system("pause");
53      return 0;
54  }
```

d) Sztringek

Az egydimenziós tömböket leggyakrabban karakter sztringek tárolására használjuk. A C++ nyelv nem rendelkezik önálló sztring típussal, ezért a karaktertömböket használja a sztringek tárolására. A sztring tehát olyan karaktertömb (`char[]`), melyben a karaktersorozat végét nulla értékű bájt (0) jelzi.

Amikor helyet foglalunk valamely sztring számára, akkor a sztring végét jelző bájtot is figyelembe kell venni. Ha az **str** tömbben maximálisan 80 karakteres szöveget szeretnénk tárolni, akkor a tömb méretét 80+1=81-nek kell megadnunk.

```
char sor[81];
```

Fontos, hogy mindig lezárjuk bináris nullával, amit karakterként így írunk: `'\0'`. Ezt azért tesszük, mert megállapodás, hogy ezzel jelezzük egy karakterlánc, azaz egy sztring végét, és minden sztring kezelő függvényt is úgy írunk meg, hogy ezt a lezáró jelet figyeljük. Természetesen a **Standard könyvtárban** (*std*) lévő sztring kezelő metódusok (eljárások, függvények) is ezt figyelik, így ha eltérünk ettől, vagy csak elfelejtjük, akkor máris mindenféle nehezen kideríthető hibához juthatunk.

Egy ilyen sztring, ami egy C típusú sztring (hiszen a C nyelvből ered, mint ahogy szinte minden a C++-ban) nem más, mint a memóriában egymás mellett lévő karakterek sorozata. Az „egymás mellett lévőség” persze nem mindig igaz, csak abban az esetben lehetséges, ha fordítási időben foglalódik le a memória területe. Eddig csak olyan példákat láttunk, ahol ez így történt. Ilyenkor persze, ha túl nagy helyet szeretnénk kérni, azt az operációs rendszer nem biztos, hogy teljesíteni tudja. Közel sem biztos, hogy talál akkor összefüggő szabad helyet a memóriában, mint amekkorára mi írni szeretnénk.

Amennyiben a sztringünket elfelejtjük lezárni egy nullával (`'\0'`), akkor, ha ki szeretnénk írni az ott lévő tartalmat, mondjuk azt, hogy „alma”, akkor egészen addig olvassa azt a memóriaterületet, amíg egy nullát (`'\0'`) nem talál. Így könnyen lehet, hogy a várt eredmény az „alma”, azonban amit a képernyőn látunk, az inkább hasonlít erre: „alma+&@[tz~^”. Az így kiírt memória szemét tartalmazhat nem megjelenítendő karaktereket. Egyébként azért jelennek meg más karakterek, mert azon a memóriaterületen már van valami. Ezekre, ha alacsony szinten tekintünk nem mások, mint számok. De mivel sztring kiíratást kértünk, ezért a program karakterekként szeretné megjeleníteni, azaz minden ott lévő 1 bájtnyi hosszúságú számnak megkeresi az ASCII-beli karakter megfelelőjét, és azt írja ki. Nem megjeleníthető karakterek például a tabulátor, a sortörés, stb.

Ha egy sztringbe írni szeretnénk, a helyzet még rosszabb, de legalább könnyebben észrevehető. Ebben az esetben egy olyan helyre szeretnénk írni, amit nem foglaltunk le a programunk számára. Abban a pillanatban, hogy ilyet szeretnénk tenni, az operációs rendszer „rácsap egy nagyot a kezünkre”, amitől elszáll a program. Ha debug módban indítottuk az éppen használt IDE-nkből (Integrated Developement Enviorement ~ Integrált fejlesztői környezet), akkor valószínűleg nem csak elszáll a program, de mindenféle hasznos hibaüzenetet is kiír a számunkra.

Így tehát mindig, minden esetben figyeljünk a sztringek lezárására, és ne próbáljunk meg eltérni a szabványtól.

Az elmondottakra tekintsünk egy példát:

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      char tomb[5];
7      cout << "A program feltolt egy 5 elemu tombot a betuvel.\n\n";
8      cout << "A tomb merete: " << sizeof(tomb);
9      cout << endl;
10     for (int i = 0; i < 5; i++)
11     {
12         tomb[i] = 'a';
13     }
14     tomb[4] = '\0';
15     cout << "\nA tomb tartalma: " << tomb;
16
17     cout << endl << endl;
18
19     system("pause");
20     return 0;
21 }
```

A 14. sorban lezártuk az utolsó elemet egy bináris nullával, az előzőekben elmondott okok miatt. Az is szépen látszik (ha lefuttatjuk a programot), hogy a képernyőre 4 darab „a” betű íródik, hiszen a valódi tömbméret ennyi.

e) A *String* osztály

Az alacsony szintű, C típusú sztringek kezelése igencsak nehézkes. Ezért létezik egy, a Standard Library-ban létező String osztály, amely nagyban megkönnyítheti a dolgunkat.

Része egy, az előbb írt C típusú sztring. Ha nagyon mélyre megyünk, ez megtalálható benne. Azonban a felelősség nagy részét leveszi a vállunkról. A saját memóriáját dinamikusan kezeli. Kérhetünk ellenőrzött hozzáférést is, így nem az operációs rendszer „csap rá a kezünkre”, és nem száll el a program, hanem egy kivétel dobódik. A kivétel egy olyan hiba, amire előre fel tudunk készülni. A kivételkezelés egy elég komoly és nehéz téma, ezért erről itt bővebben nem írok.

A String osztály rengeteg tagmetódussal rendelkezik, melyek használatával egy mondatban egy „sz” megkeresése, bizonyos részek kicserélése, vagy az „alma” felülírása „kókuszdió”-ra (ami lényegesebben hosszabb szó) már pofon egyszerű feladat.

A string típus (*osztály*) használatához a string *include* file-t kell „betölteni”.

```
#include <string>
```

Ha egy string adattal akarunk valamilyen műveletet végezni akkor a következő formát kell használni:

```
stringváltozó.művelet (argumentumok)
```

A szövegben előforduló karakter számozása *nullával* kezdődik és a „*hossz – 1*”-ig tart. Az utolsó karakternek nem kell zérusnak lennie, mint a C nyelvben, az osztály tudja magáról hogy milyen hosszú. Az értékadás szövegeknél is működik, bár az értékadásnál a jobb oldali szöveg átmásolódik a bal oldali szövegbe!

```
string szoveg1 = "alma"
string szoveg2 = "paradicsom"
szoveg1 = szoveg2
```

A `szoveg1` tartalma "paradicsom" lesz.

A `string` osztály néhány egyszerűen használható tagfüggvénye:

Szöveg (string) hossza:

```
szöveg.lenght();
```

Példa:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string szoveg = "Alma";
    cout << szoveg.length();

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Hozzáférés a szöveg i. karakteréhez

```
szöveg.at(i);
```

Példa:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string szoveg = "Alma";
    cout << szoveg.at(2);

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Üres-e a szöveg (0 – HAMIS, 1 – IGAZ)

```
szöveg.empty();
```

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string szoveg = "Alma";
    cout << szoveg.empty();

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Szöveg kiürítése

```
szöveg.clear();
```

Példa:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string szoveg = "Alma";
    szoveg.clear();
    cout << szoveg.empty();

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Szövegrész törlése

```
szöveg.erase();
```

Két paramétere van, hányadik pozíciótól töröljünk, és mennyi karaktert. Az alábbi példában két karakter marad: Aa.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string szoveg = "Alma";
    szoveg.erase(1, 2);
    cout << szoveg;

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Szövegrész kivágása

`szöveg.substr();`

Két paramétere van, hányadik pozíciótól, és hány karakter vágjunk ki. Az alábbi példában két karaktert vágunk ki az 1. pozíciótól kezdve, vagyis az eredmény: lm.

Példa:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string szoveg = "Alma";
    cout << szoveg;
    cout << endl << endl;
    cout << szoveg.substr(1, 2);

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Szöveg hozzáfűzés szöveghez

`szöveg.append();`

Az alábbi példa eredménye: Almaecet.

Példa:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string szoveg = "Alma";
    cout << szoveg.append("ecet");

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Szöveg beillesztése szövegbe

`szöveg.insert();`

Két paraméter van, az elsőben megadjuk, hogy hányadik pozíciótól, a másodikban pedig a beillesztendő szöveg található. Az alábbi példa eredménye: Alma.

Példa:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string szoveg = "Aa";
    cout << szoveg.insert(1, "lm");

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Szövegrész lecserélése

`szöveg.replace();`

Három paraméter van, hányadik pozíciótól, hány karaktert és mire cseréljen le. Az alábbi példa eredménye: Barna folyas.

Példa:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string szoveg = "Barna tojas";
    cout << szoveg.replace(6, 3, "foly");

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Szövegrész keresése

`szöveg.find();`

Két paramétere van, mit keresünk (szöveg) és hányadik pozíciótól keressünk (ha nem adjuk meg, akkor elölről keres). Ha több egyforma szövegrész van, akkor is csak az első megtalált szövegrész pozícióját adja meg (a szövegrész első karakterének pozícióját). Ha nem találja meg a szövegrészt, akkor egy úgynevezett `extremális` értéket (szélsőértéket) ad vissza. Ez tulajdonképpen egy nem létező index, ennek értéke a gép hardverétől függ, pl. 32 bites gép esetén 2^{32} = 4 294 967 295.

Ezt a nem létező index értéket megnézhetjük az `npos` tagállandóval: `cout << szoveg.npos;`

Az alábbi példában az ugrani szövegrészt keressük elölről kezdve, első előfordulásának első karaktere a 12. pozíción helyezkedik el.

Példa:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string szoveg = "Az oroszlan ugrani keszul. A tigris is ugrani akar.";
    cout << szoveg.find("ugrani");

    cout << endl << endl;
    system("pause");
    return 0;
}
```

A következő példában megadtuk azt is, hogy hányadik pozíciótól kezdjen keresni. Mivel 13-at adtunk meg, így onnan végzi a keresést, és a 39. pozíción meg is találja a következő „ugrani” szövegrész első karakterét.

Példa:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string szoveg = "Az oroszlan ugrani keszul. A tigris is ugrani akar.";
    cout << szoveg.find("ugrani", 13);

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Szövegrész keresése

`szöveg.rfind();`

Majdnem ugyanaz, mint az előző keresés tagfüggvény. Lényeges különbség azonban, hogy az utolsó előfordulást keresi!

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string szoveg = "Az oroszlan ugrani keszul. A tigris is ugrani akar.";
    cout << szoveg.rfind("ugrani");

    cout << endl << endl;
    system("pause");
    return 0;
}
```

A fenti példában a második előfordulás első karakterének pozícióját kapjuk meg, azaz, a 39. pozíciót. Ha megadunk egy pozíciót, akkor onnan keres, de visszafelé nézi a szöveget! Tehát, ha például a 38. pozíciótól akarunk keresni, akkor az eredmény 12 lesz. Ez azért van, mert arról a helyről visszafelé csak egyszer fordul elő az „ugrani” szó, és ennek első karaktere a 12. pozíción van a szövegben. Nézzük ezt is meg egy példával:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string szoveg = "Az oroszlan ugrani keszul. A tigris is ugrani akar.";
    cout << szoveg.rfind("ugrani", 38);

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Ha a 11. pozíciótól vagy annál kisebb pozíciótól keresünk, akkor nem találja meg az „ugrani” szövegrészt. Ekkor azt a bizonyos `extremális` értéket kapjuk vissza. Lássuk:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string szoveg = "Az oroszlan ugrani keszul. A tigris is ugrani akar.";
    cout << szoveg.rfind("ugrani", 11);

    cout << endl << endl;
    system("pause");
    return 0;
}
```

Szövegek összehasonlítása

```
szöveg.compare();
```

Egy paramétere van, az a szövegrész, melyet összehasonlítunk az eredeti szövegünkkel. Ha teljesen megegyezik (kis-nagy betű eltéréseket is figyel), akkor 0 (IGAZ), egyébként más szám (HAMIS).

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main()
6  {
7      string szoveg = "Alma";
8      cout << szoveg.compare("Alma");
9
10     cout << endl << endl;
11     system("pause");
12     return 0;
13 }
```

f) A struktúra típus

A programozás során azonban gyakran találkozhatunk olyan problémákkal, amelyek megoldásához különböző típusú adatokat önálló programozási egységben kell feldolgoznunk. Tipikus területe az ilyen jellegű feladatoknak az adatbázis-kezelés, ahol a fájl tárolási egysége a rekord tetszőleges mezőkből épülhet fel.

C++ nyelven a struktúra (struct) típus több tetszőleges típusú (kivéve a void és a függvény típust) adatok együttese. Ezek az adatelemek önálló, a struktúrán belül érvényes nevekkel rendelkeznek. Az objektumok szokásos elnevezése struktúraelem vagy adattag (*member*).

A struktúra típusú változó létrehozása logikailag két részre osztható. Először deklarálnunk kell magát a struktúratípust, melyet felhasználva változókat definiálhatunk. A struktúra szerkezetét meghatározó deklaráció általános formája:

```
struct struktúratípus
{
típus1 tag1;
típus2 tag2;
…
típusN tagN;
};
```

Felhívjuk a figyelmet arra, hogy a struktúra deklarációja azon kevés esetek egyike, ahol a pontosvesszőt kötelező kitenni. Az adattagok deklarációjára a C++ nyelv szokásos deklarációs szabályai érvényesek. A fenti típussal változót a már megismert módon készíthetünk.

A struktúra (*struct*) és a tömbök között a legnagyobb különbség, hogy míg egy tömbben csak egy féle adattípust tárolhatunk, a struktúra több típusú adat tárolására is alkalmas.

Erre egy egyszerű példa lehet, ha emberek adatait szeretnénk tárolni (nevüket, lakcímüket, születési dátumukat illetve egy azonosítót). Többféle adatról beszélünk, ezért külön tömbök kellenének a nevek (szövegek), az azonosítók (egész számok), stb. tárolására. Az adatokat így nehézkes kezelni, nehéz velük dolgozni. Azonban, ha struktúrát használunk, akkor egy struktúrában össze tudjuk foglalni mindezt, és jóval egyszerűbben tudjuk az adatainkat kezelni. Megj.: A feladat megoldásánál használjunk egy struktúrát a dátumhoz is.

Nézzünk egy egyszerű példát:

```
#include <iostream>
#include <string>

using namespace std;

struct Datum
{
    int ev;
    int honap;
    int nap;
};

struct Ember
{
    string nev;
    string lakcim;
    Datum szuletesiEv;
    int azon;
};

int main()
{
    Ember a;
    a.azon = 13;
    a.nev = "Kis Peter";
    a.lakcim = "Mimoza u. 28.";
    a.szuletesiEv.ev = 1993;
    a.szuletesiEv.honap = 3;
    a.szuletesiEv.nap = 28;
    cout << a.azon << endl;
    cout << a.nev << endl;
    cout << a.szuletesiEv.ev << ".";
    cout << a.szuletesiEv.honap << ".";
    cout << a.szuletesiEv.nap << "." << endl;
    cout << a.lakcim << endl << endl;

    cout << endl << endl;
    system("pause");
    return 0;
}
```

A tagokat a .(pont)-tal tudjuk elérni. Fontos megemlíteni, hogy C++-ban a struktúra az tulajdonképpen egy osztály (*class*), melynek minden egyes adattagja publikus (*public*). Hogy ez mit jelent, arról most még ne beszéljünk. A struktúrákat gyakorlatilag minden emelt szintű informatika érettségi programozás részénél érdemes használni.

9. Függvények

A függvény a C++ program olyan névvel ellátott egysége (alprogram), amely a program más részeiből annyiszor meghívható, ahányszor csak szükség van a függvényben definiált tevékenységsorozatra. A hagyományos (funkció-orientált) C++ program általában sok kisméretű, jól kézben tartható függvényből

épül fel. A gyakran használt függvények lefordított kódját könyvtárakba rendezhetjük, amelyekből a szerkesztőprogram a hivatkozott függvényeket beépíti a programunkba.

A függvények hatékony felhasználása érdekében a C++ nyelv lehetőséget biztosít arra, hogy a függvény bizonyos belső tárolóinak a függvényhívás során adjunk értéket. Hogy melyek ezek a tárolók és milyen típussal rendelkeznek, azt a függvény definíciójában a függvény neve után zárójelben kell megadnunk. A hívásnál pedig hasonló formában kell felsorolnunk az átadni kívánt értékeket. A szakirodalom ezekre a tárolókra és értékekre különböző nevekkel hivatkozik.

A függvény-definícióban szereplő tárolók	**A függvényhívás során megadott értékek**
formális paraméterek	*aktuális paraméterek*
formális argumentumok	*aktuális argumentumok*
paraméterek	*argumentumok*

A függvényhívás során a vezérlés a hívó függvénytől átkerül az aktivizált függvényhez. Az argumentumok (amennyiben vannak) szintén átadódnak a meghívott függvénynek. A már bemutatott **return** utasítás végrehajtásakor, illetve a függvény fizikai végének elérésekor a hívott függvény visszatér a hívás helyére, és a return utasításban szereplő kifejezés, mint függvényérték (visszatérési érték) jelenik meg. A visszatérési érték nem más, mint a függvényhívás kifejezés értéke.

A **saját** késztésű függvényeinket mindig **definiálni** kell. A definíció, amelyet csak egyszer lehet megadni, a C++ programon belül bárhol elhelyezkedhet. Amennyiben a függvény definíciója megelőzi a felhasználás (hívás) helyét akkor, ez egyben a függvény prototípusa is.

A függvény deklarációja (prototípusa) tartalmazza a függvény nevét, visszatérési értékének típusát valamint információt szolgáltat a paraméterek számáról és típusáról. A prototípust a függvényhívás előtt kell elhelyeznünk a programban. A C++ fordító csak a prototípus ismeretében fordítja le a függvényhívást. (A függvény definíciója helyettesíti a prototípust.)

Az elmondottakban megtaláljuk annak magyarázatát, hogy a szabványos könyvtári függvények deklarációját tartalmazó fejállományokat miért a forrásfájl elején építjük be *(#include*) a programunkba. Valamely prototípus többször is szerepelhet, azonban mindegyik előfordulásnak azonosnak kell lennie.

Nézzük meg a függvénydefiníció általános formáját! A függvény fejsorában a „paraméter-deklarációs lista” az egyes paramétereket vessző elválasztva tartalmazza. Minden egyes paraméter előtt szerepel annak típusa.

```
<visszatérési típus> függvénynév (<paraméter-deklarációs lista>)
{
függvény törzse
<lokális definíciók és deklarációk>
<utasítások>
}
```

A <> jelek között megadott részek hiányozhatnak a definícióból. Példaként készítsük el két egész szám összeadását végző függvényt:

```
1  #include <iostream>
2  using namespace std;
3
4  int osszeado(int, int);
5
6  int main()
7  {
8      int n, m;
9      cout << "Osszeadok ket egesz szamot. Kerem a szamokat:";
10     cout << endl << endl;
11     cout << "Az elso szam:\t\t";
12     cin >> n;
13     cout << "A masodik szam:\t\t";
14     cin >> m;
15     cout << endl << endl;
16     cout << "A szamok osszege:\t" << osszeado(n, m);
17     cout << endl << endl;
18     system("pause");
19     return 0;
20 }
21
22 int osszeado(int a, int b)
23 {
24     return a + b;
25 }
```

A fenti példa 4. sorában látható a függvény prototípusa, a 22. sortól pedig maga a függvény. A prototípusban megadtuk a függvény visszatérési értékének típusát (*int*), illetve a paraméterek számát, típusát (*2 darab egész típusú paraméter*). A 16. sorban hívtuk meg a függvényt az általunk megadott két egész számmal, és ugyanitt ki is írattuk az eredményt. Fontos megjegyezni, hogy a függvény hívásakor az `n` és az `m` paraméterek értékei átmásolódnak a 22. sortól kezdődő függvény paramétereibe (*n paraméter az a paraméterbe, m paraméter pedig a b paraméterbe*).

Nézzünk most egy kicsit összetettebb példát! A jól ismert másodfokú egyenlet megoldó képletét felhasználva számoljuk ki a gyököket függvény segítségével. Ehhez először be kell tölteni a `cmath` osztályt (`#include <cmath>`), mert használni szeretnénk a négyzetgyök függvényt (**sqrt**). Ebben az osztályban még sok, mások által megírt matematikai függvények és egyéb definíciók találhatók. Az 5. sorban készítettük el a függvényünk prototípusát, melynek visszatérési értéke `void` típusú (*nem ad vissza értéket, tulajdonképpen egy eljárás*), és három paramétere pedig `double` típusú. A függvényt a 26. sortól kezdődően alkottuk meg, és a 19. sorban hívtuk meg az általunk megadott három tizedes törttel (természetesen egész számok is megadhatók). A programunkat egy kicsit formáztuk is a megfelelő *escape* karakterekkel (tabulátor, sortörés).

```cpp
#include <iostream>
#include <cmath>
using namespace std;

void megoldo_keplet(double, double, double);

int main()
{
    double a, b, c;
    cout << "Masodfoku egyenletet oldok meg.\n";
    cout << "Add meg az a, b es c ertekeket!\n\n";
    cout << "Az 'a' erteke:\t";
    cin >> a;
    cout << "A 'b' erteke:\t";
    cin >> b;
    cout << "A 'c' erteke:\t";
    cin >> c;
    cout << endl << endl;
    megoldo_keplet(a, b, c);

    cout << endl << endl;
    system("pause");
    return 0;
}

void megoldo_keplet(double a, double b, double c)
{
    double d = b*b - 4 * a*c;
    if (d < 0)
    {
        cout << "Nincs megoldas!\n";
    }
    else if (d == 0)
    {
        cout << "A megoldas: " << (-b) / (2 * a) << endl;
    }
    else
    {
        cout << "Az elso megoldas:\t" << ((-b) + sqrt(d)) / (2 * a);
        cout << endl;
        cout << "A masodik megoldas:\t" << ((-b) - sqrt(d)) / (2 * a);
        cout << endl;
    }
}
```

Téglalap kerületét kiszámoló program.

```cpp
#include <iostream>
using namespace std;

int teglalap_kerulet(int, int);

int main()
{
    int x, y;
    cout << "Egy teglalap keruletet szamolom ki.\n";
    cout << "Az oldalak hossza egesz szam lehet.\n\n";
    cout << "Az a oldal hossza:\t";
    cin >> x;
    cout << "A b oldal hossza:\t";
    cin >> y;
    cout << "\nA teglalap kerulete:\t" << teglalap_kerulet(x, y);

    cout << endl << endl;
    system("pause");
    return 0;
}
int teglalap_kerulet(int a, int b)
{
    return 2 * (a + b);
}
```

Csonka gúla térfogatát kiszámoló program.

```cpp
#include <iostream>
#include <cmath>
using namespace std;

double csonkagula_terfogat(double, double, double);

int main()
{
    double a, b, c;
    cout << "Egy negyzet alapu csonkagula terfogatat szamolom ki.\n";
    cout << "\nMennyi legyen az alaplap ele:\t\t";
    cin >> a;
    cout << "Mennyi legyen a fedolap ele:\t\t";
    cin >> b;
    cout << "Mennyi legyen a csonkagula magassaga:\t";
    cin >> c;
    cout << "\nA csonkagula terfogata:\t\t\t" << csonkagula_terfogat(a, b, c);
    cout << endl << endl;
    system("pause");
    return 0;
}

double csonkagula_terfogat(double x, double y, double m)
{
    double T = x * x;
    double t = y * y;
    return ((T + t + sqrt(T*t))*m) / 3;
}
```

10. Fájlkezelés

Egy programban nem csak a képernyőre történő írásra vagy az onnan való beolvasásra lehet szükségünk. Sok esetben fájlokat is kell kezelnünk. Eddig az *iostream* –et használtuk a C++ standard library-ből. Segítségével tudtunk kiírni és beolvasni a képernyőről. A fájlműveletekhez az *fstream* –et használhatjuk.

A *cout*-hoz hasonló adatfolyamot használhatunk fájlkezeléshez is. az *fstream*-et include-olva létrehozhatunk *ofstream, ifstream* és *fstream* objektumokat. Az *ofstream* az output file stream. Segítségével létrehozhatunk fájlokat és írhatunk beléjük. Az *ifstream* az input file stream. Segítségével fájlokból tudunk beolvasni adatokat. Az *fstream* tulajdonképpen a kettő együtt (file stream). Vele tudunk fájlokat létrehozni, megnyitni, bele tudunk írni, olvasni tudunk beléjük és le is tudjuk őket zárni. Fájlok megnyitására és lezárására természetesen a csak írni és a csak olvasni tudó *ofstream* és *ifstream* is képes.

Kövessünk végig egy rövid példát, amiben megnyitunk egy létező „a.txt” fájlt, beolvassuk a tartalmát soronként, majd minden egyes sort kiírunk egy „b.txt” fájlba. Használjunk a feladathoz külön egy ifstream és egy ofstream objektumot.

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4
5  using namespace std;
6
7  int main()
8  {
9      string sor;
10
11     ifstream file_bemenet;
12     file_bemenet.open("a.txt");
13
14     ofstream file_kimenet;
15     file_kimenet.open("b.txt");
16
17     while (!file_bemenet.eof())
18     {
19         getline(file_bemenet, sor);
20         file_kimenet << sor << endl;
21     }
22
23     file_bemenet.close();
24     file_kimenet.close();
25
26     system("pause");
27     return 0;
28 }
```

A példában, mint látható, létrehoztunk egy *ifstream*-et „file_bemenet” néven, és egy *ofstream*-et „file_kimenet” néven. Mind a kettőnél az *open()* tagfüggvénnyel nyitottuk meg a fájlokat. Az *ofstream* esetében, mivel nem talált ilyen fájlt, automatikusan létrehozza azt üresen, majd megnyitja.

Utána egy *while* ciklusban olvassuk be a fájl tartalmát soronként. A *while* feltételeként azt vizsgáljuk, hogy a megnyitott fájlunknál a végén járunk-e. Ezt az *eof()* tagfüggvénnyel tesszük meg. EOF azaz End Of File. Hogy ezt honnan tudja? Minden operációs rendszer használ valamilyen karaktert, amivel a fájl végeit jelzi.

Egy sor beolvasásához a *getline()* függvényt használjuk. A függvény első paramétere egy input stream. Ez lehet a már korábban használt *cin* is, amennyiben a képernyőről szeretnénk beolvasni egy sort, vagy akár lehet egy

ifstream objektum is, mint a jelen esetben. A metódus második paraméterként egy stringet vár, ahova beolvashatja a sort. Erre a célra létrehoztunk egy stringet „sor” néven, amiben mindig ideiglenesen eltároljuk az adott sort.

Miután ebbe beolvastuk, a „file_kimenet”-ünkbe beleirányítjuk a már megszokott „<<” operátorral a „sor” nevű stringünk tartalmát.

Miután az egész fájlt átmásoltuk, a végén lezárjuk a „file_bemenet” és a „file_kimenet” *ifstream* illetve *ofstream* objektumainkat.

Fontos megjegyezni, hogy a *getline()* függvény mindent beolvas egészen a sortörésig (windows-ban ez a ’\n’ karakter), de a sortörés nem kerül bele a stringbe. Ezért is tettünk külön, minden egyes sorunk után egy sortörést a fájlba való írásnál.

Egy emelt szintű informatika érettségi során a legtöbbször nem arra van szükség, hogy egy fájlt soronként kezeljünk, hanem a forrásfájlban megadott értékeket be tudjuk olvasni és el tudjuk tárolni valami féle, általunk létrehozott adatszerkezetben (ami az esetek 99%-ban egy struktúra).

Nézzünk egy olyan egyszerű példát, ahol egy háziállat adatait olvassuk be. A fájl legyen egy olvasható txt fájl, melyben egy sor egy állat adatait tartalmazza rendre: az állat azonosítója, neve, születési éve, súlya (kg).

A fájl tartalmára példa:

3255532123	Bodri	2004	11.3
3255534417	Füles	1999	7.4
…	…	…	…

Egy érettségi feladatnál mindig megmondják, hogy maximum hány sor lehet a forrásban (hogy ne kelljen dinamikus memóriakezeléssel foglalkozni), továbbá az egyes adatok típusára is tesznek valamilyen utalásokat, például, hogy a születési évnél csak az évet rögzítik, vagy például azt, hogy a súlya nem biztos, hogy egy egész szám. A feladat legyen annyi, hogy az adatokat beolvassuk és eltároljuk, majd utána kiírjuk őket a képernyőre.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
struct Allat
{
    string azon, nev;
    int ev;
    float suly;
};
Allat kisallatokat_tartalmazo_tomb[20];
int allatok_szama = 0;
void egy_kisallat_adatainak_kiirasa(int);
int main()
{
    ifstream file_bemenet;
    file_bemenet.open("allatok.txt");
    while (!file_bemenet.eof())
    {
        file_bemenet >> kisallatokat_tartalmazo_tomb[allatok_szama].azon;
        file_bemenet >> kisallatokat_tartalmazo_tomb[allatok_szama].nev;
        file_bemenet >> kisallatokat_tartalmazo_tomb[allatok_szama].ev;
        file_bemenet >> kisallatokat_tartalmazo_tomb[allatok_szama].suly;
        allatok_szama++;
    }
    file_bemenet.close();
```

```
27      for (int i = 0; i < allatok_szama; i++)
28      {
29          egy_kisallat_adatainak_kiirasa(i);
30          cout << endl << endl;
31      }
32      system("pause");
33      return 0;
34  }
35  void egy_kisallat_adatainak_kiirasa(int index)
36  {
37      cout << kisallatokat_tartalmazo_tomb[index].azon << endl;
38      cout << kisallatokat_tartalmazo_tomb[index].nev << endl;
39      cout << kisallatokat_tartalmazo_tomb[index].ev << endl;
40      cout << kisallatokat_tartalmazo_tomb[index].suly << endl;
41  }
```

A feladat megoldása során létrehoztunk egy, az adatoknak megfelelő struktúrát, és egy ilyen struktúrákból álló tömböt, amiben eltárolhatjuk az adatokat. Továbbá létrehoztunk még egy egész számot, amiben azt tároljuk el, hogy hány állat adatait olvastuk be a fájlból.

Minden ilyen adatot globálisan vettünk fel, mert így sokkal egyszerűbben tudjuk kezelni és elérjük őket mindenhol. Ez persze nem egy szép megoldás, de egy érettségi feladatnál ez egyáltalán nem számít. Ha minden szükséges adatot így, globálisként veszünk fel, akkor azokat minden függvényünkben elérünk, így nem kell az adatokat átadni paraméterként.

A példában az előzőhöz képest az a nagy különbség, hogy nem soronként olvasunk be a fájlból, hanem adatonként. Ezt a *cin*-hez hasonlóként használhatjuk a „>>” operátorral. Így könnyen be tudjuk olvasni az adatokat.

Ha így teszünk, a kurzor pozíciójától minden white space-t átugrik (szóköz, tabulátor, sortörés, stb.) majd beolvas egy „adatnak vélt adatot”. Ez alatt azt értjük, hogy például az állat súlyának beolvasásakor azt nem egy egész számként egy karakterként és egy másik egész számként észleli (11.4), hanem egy lebegőpontos számként.

A már az előbb látott példában is szereplő while ciklusban olvasunk be mindent, egészen a fájl végéig, majd azt mindig eltároljuk a tömbünkben. A beolvasás után lezárjuk a tömböt, majd ismét bejárjuk és kiírjuk a már látott módon.

`Függelék`

Programozási tételek

1) Az összegzés tétele

Adott egy N elemű számsorozat. Számítsuk ki az elemek összegét! A sorozat elemeit már ismertnek tételezzük fel és ezeket az N elemű A(N) vektorban tároljuk.

```
#include <iostream>
using namespace std;

const int MAX_DB = 300;

int main()
{
    int darab, osszeg;
    int a[MAX_DB];
    cout << "Az osszegzes tetele.\n\n";
    cout << "Hany egesz szamot adjak ossze (max. " << MAX_DB << " db.): ";
    cin >> darab;
    cout << "\nKerem a szamokat:\n\n";
    for (int i = 0; i < darab; i++)
    {
        cout << "A(z) " << i + 1 << ". szam: ";
        cin >> a[i];
    }
    osszeg = 0;
    for (int i = 0; i < darab; i++)
    {
        osszeg = osszeg + a[i];
    }
    cout << "\nA szamok osszege: " << osszeg;
    cout << endl << endl;
    system("pause");
    return 0;
}
```

2) Az eldöntés tétele

Adott egy N elemű sorozat, és egy, a sorozaton értelmezett T tulajdonság. Döntsük el, hogy van-e a sorozatban legalább egy T tulajdonságú elem (30-nál nagyobb szám)!

```
#include <iostream>
using namespace std;

const int MAX_DB = 300;

int main()
{
    int darab;
    int a[MAX_DB];
    cout << "Az eldontes tetele.\n\n";
    cout << "Hany egesz szamot vizsgaljak (max. " << MAX_DB << " db.): ";
    cin >> darab;
    cout << "\nKerem a szamokat:\n\n";
    for (int i = 0; i < darab; i++)
    {
        cout << "A(z) " << i + 1 << ". szam: ";
        cin >> a[i];
    }

    int i = 0;
    while (a[i] <= 30 && i < darab)
    {
        i++;
    }
    if (i == darab)
    {
        cout << "\nNincs ";
    }
    else
    {
        cout << "\nVan ";
    }
    cout << "ilyen elem.";
    cout << endl << endl;
    system("pause");
    return 0;
}
```

3) A kiválasztás tétele

Adott egy N elemű sorozat és egy, a sorozat elemein értelmezett T tulajdonság. Tudjuk, hogy a sorozatban van legalább egy T tulajdonságú elem (30-nál nagyobb szám). Adjuk meg az első ilyen elem sorszámát!

```
#include <iostream>
using namespace std;

const int MAX_DB = 300;

int main()
{
    int darab;
    int a[MAX_DB];
    cout << "A kivalasztas tetele.";
    cout << "\nFeltetelezzuk, hogy van legalabb 1 db";
    cout << "30-nal nagyobb egesz szam!\n\n";
    cout << "Hany egesz szamot vizsgaljak (max. " << MAX_DB << " db.): ";
    cin >> darab;
    cout << "\nKerem a szamokat:\n\n";
    for (int i = 0; i < darab; i++)
    {
        cout << "A(z) " << i + 1 << ". szam: ";
        cin >> a[i];
    }

    int i = 0;
    while (a[i] <= 30 && i < darab)
    {
        i++;
    }
    cout << "\n\nAz elso ilyen elem sorszama: " << i + 1 << ".";
    cout << endl << endl;
    system("pause");
    return 0;
}
```

4) A megszámlálás tétele

Adott egy N elemű sorozat és egy, a sorozat elemein értelmezett T tulajdonság (30-nál nagyobb szám). Számoljuk meg, hogy hány T tulajdonságú eleme van a sorozatnak.

```
#include <iostream>
using namespace std;

const int MAX_DB = 300;

int main()
{
    int darab;
    int a[MAX_DB];
    cout << "A megszamlalas tetele. A 30-nal nagyobbakat szamolom meg.";
    cout << endl;
    cout << "Hany egesz szamot vizsgaljak (max. " << MAX_DB << " db.): ";
    cin >> darab;
    cout << "\nKerem a szamokat:\n\n";
    for (int i = 0; i < darab; i++)
    {
        cout << "A(z) " << i + 1 << ". szam: ";
        cin >> a[i];
    }
    int elemek_szama = 0;
    for (int i = 0; i < darab; i++)
    {
        if (a[i]>30)
        {
            elemek_szama++;
        }
    }
    cout << "\n\nA 30-nal nagyobb elemek szama: " << elemek_szama;
    cout << endl << endl;
    system("pause");
    return 0;
}
```

5) A lineáris keresés tétele

Adott egy N elemű sorozat és egy, a sorozat elemein értelmezett T tulajdonság (30-nál nagyobb). Döntsük el, hogy a sorozatban van-e T tulajdonságú elem, és ha igen, akkor adjuk meg az első ilyen elem sorszámát!

```
#include <iostream>
using namespace std;

const int MAX_DB = 300;

int main()
{
    int darab;
    int a[MAX_DB];
    cout << "A linearis kereses tetele.";
    cout << endl;
    cout << "Hany egesz szamot vizsgaljak (max. " << MAX_DB << " db.): ";
    cin >> darab;
    cout << "\nKerem a szamokat:\n\n";
    for (int i = 0; i < darab; i++)
    {
        cout << "A(z) " << i + 1 << ". szam: ";
        cin >> a[i];
    }
    int i = 0;
    while (a[i] <= 30 && i < darab)
    {
        i++;
    }
    if (i != darab)
    {
        cout << "\nAz elso ilyen elem sorszama: " << i + 1 << ".";
    }
    else
    {
        cout << "\nNem volt ilyen elem.";
    }
    cout << endl << endl;
    system("pause");
    return 0;
}
```

6) A maximum és minimum érték keresés tétele

Adott egy N elemű A(N) sorozat. Határozzuk meg a sorozat legnagyobb és legkisebb elemének értékét! Megjegyzés: Nem vizsgáljuk azokat az eseteket, amikor több maximális vagy minimális érték van, esetleg a sorozat minden eleme egyforma.

```
#include <iostream>
using namespace std;

const int MAX_DB = 300;

int main()
{
    int darab;
    int a[MAX_DB];
    cout << "A maximum es minimum ertek kereses tetele.";
    cout << endl;
    cout << "Hany egesz szamot vizsgaljak (max. " << MAX_DB << " db.): ";
    cin >> darab;
    cout << "\nKerem a szamokat:\n\n";
    for (int i = 0; i < darab; i++)
    {
        cout << "A(z) " << i + 1 << ". szam: ";
        cin >> a[i];
    }
    int max_ertek = a[0];
    int min_ertek = a[0];
    for (int i = 1; i<darab; i++)
    {
        if (a[i] > max_ertek)
        {
            max_ertek = a[i];
        }
        if (a[i] < min_ertek)
        {
            min_ertek = a[i];
        }
    }
    cout << "\n\nA legnagyobb szam:\t" << max_ertek;
    cout << "\n\nA legkisebb szam:\t" << min_ertek;
    cout << endl << endl;
    system("pause");
    return 0;
}
```

7) A kiválogatás tétele

Adott egy N elemű sorozat A(N) és egy, a sorozat elemein értelmezett T tulajdonság (30-nál nagyobb). Határozzuk meg a sorozat összes T tulajdonságú elemét, és gyűjtsük ki ezen elemek sorszámait a B() vektorba! (A művelet után írjuk ki az eredményt!)

```
#include <iostream>
using namespace std;

const int MAX_DB = 300;

int main()
{
int darab;
int a[MAX_DB];
cout << "A kivalogatas tetele.";
cout << endl;
cout << "Hany egesz szamot vizsgaljak (max. " << MAX_DB << " db.): ";
cin >> darab;
cout << "\nKerem a szamokat:\n\n";
for (int i = 0; i<darab; i++)
{
cout << "A(z) " << i + 1 << ". szam: ";
cin >> a[i];
}
int eredmeny[MAX_DB];
int j = 0;
for (int i = 0; i<darab; i++)
{
if (a[i] > 30)
{
eredmeny[j] = i;
j++;
}
}
cout << "\n\nA talalt elemek sorszamai:\n\n";
for (int i = 0; i<j; i++)
{
cout << eredmeny[i] + 1 << ".\n";
}
cout << endl << endl;
system("pause");
return 0;
}
```

8) A közvetlen kiválasztásos rendezés tétele

Adott egy N elemű A(N) sorozat. Rendezzük növekvő sorrendbe a sorozat tagjait! Megjegyzés: Ez a legegyszerűbb, de leglassúbb rendezési forma. A lényege az, hogy két ciklussal dolgozva először A(1)-től indulva kiválasztjuk a vektor legkisebb elemét úgy, hogy az elemeket összehasonlítjuk és felcseréljük, ha rossz sorrendben vannak. Az első helyen ezután biztos a legkisebb elem fog állni! Ezután A(2)-től kezdve megismételjük az eljárást, egészen A(N-1)-ig.

```
#include <iostream>
using namespace std;

const int MAX_DB = 300;

int main()
{
    int darab;
    int a[MAX_DB];
    cout << "A kozvetlen kivalasztasos rendezes tetele.";
    cout << endl;
    cout << "Hany egesz szamot vizsgaljak (max. " << MAX_DB << " db.): ";
    cin >> darab;
    cout << "\nKerem a szamokat:\n\n";
    for (int i = 0; i < darab; i++)
    {
        cout << "A(z) " << i + 1 << ". szam: ";
        cin >> a[i];
    }
    int temp;
    for (int i = 0; i < darab - 1; i++)
    {
        for (int j = i + 1; j < darab; j++)
        {
            if (a[j] < a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    cout << "\n\nA szamok rendezetten:\n\n";
    for (int i = 0; i < darab; i++)
    {
        cout << a[i] << " ";
    }
    cout << endl << endl;
    system("pause");
    return 0;
}
```