COMPUTERBOOKS

Programozzunk nyelven!

C++

AZ ANSI C++ TANKÖNYVE

TÓTH BERTALAN

Programozzunk C++ nyelven!

Az ANSI C++ tankönyve

LEKTOR
DR. TAMÁS PÉTER

COMPUTERBOOKS
BUDAPEST, 2004

A könyv készítése során a Szerző és a Kiadó a legnagyobb gondossággal jártak el. Ennek ellenére hibák előfordulása nem kizárható.
Az ismeretanyag felhasználásának következményeiért sem a Kiadó, sem a Szerző felelősséget nem vállalnak.

A könyv az Oktatási Minisztérium támogatásával, a Felsőoktatási Pályázatok Irodája által lebonyolitott Felsőoktatási Tankönyv- és Szakkönyvtámogatási Pályázat keretében jelent meg

ISBN: 963 618 301 5

© Kiadó. ComputerBooks Kiadó Kft
1126 Budapest Tartsay Vilmos u. 12
Telefon/fax. 3753-591, 3751-564
E-mail: info@computerbooks.hu
http:www.computerbooks.hu
borítóterv: Székely Edith
Nyomtatta és kötötte a MAROSI-PRINT Kft.

Tartalomjegyzék

Bevezetés

Egy új könyvvel való ismerkedés során két kérdésre kereshetjük a választ. Milyen a könyv felépítése, mit ad nekem ez a mű? A másik, nem kevésbé fontos kérdés, hogy mi is az a könyvben tárgyalt C++ nyelv?

Gondolatok a könyvről

A „*Programozzunk C++ nyelven!*" című könyv elsősorban azok számára íródott, akik most kezdenek ismerkedni napjaink programozási nyelveinek alapját képező C++ nyelvvel. Nem szükséges a teljes művet elolvasni ahhoz, hogy a feldolgozott témakört programok írásával mélyítse el az Olvasó. Amennyiben a feldolgozást sikeresnek ítéli meg, továbbléphet a következő fejezetre. Az önellenőrzéshez a CD-mellékleten található C++ fejlesztőeszköz, tesztprogram és feladatok nyújtanak segítséget.

A könyv azok számára is értékes szakirodalom lehet, akik ugyan már programokat írnak C++ nyelven, azonban a C++ nyelvnek még nem minden területén mozognak otthonosan. A fejezetek a címükben szereplő témakört teljes részletességgel és példák sokaságával tárják az Olvasó elé.

A könyv felépítése

A könyv felépítése olyan, hogy folyamatosan olvasva meg lehet ismerkedni a C++ nyelvvel, valamint a legfontosabb könyvtári függvényekkel és osztályokkal. A C++ nyelv múltja, jelene és lehetséges jövőjének bemutatása után a könyv további fejezeteit három részre osztjuk.

Az első „*A C++ mint egy jobb C nyelv*" rész a C++ nyelv hagyományos programozást támogató lehetőségeit tárja az Olvasó elé. Ezek az ismeretek elsősorban a nem objektum-orientált programkészítés során hasznosíthatók.

A második „*Az objektum-orientált C++ nyelv*" című rész hat fejezeten keresztül az objektum-orientált programkészítés alapjaival ismerteti meg az Olvasót. A tárgyalás során a részletekben való elmélyülés helyett egy világos áttekintést adunk a témakörrel kapcsolatos fogalmakról és megoldásokról.

A harmadik rész valójában a könyv függeléke, azonban ennek ellenére egy sor hasznos információt tartalmaz a C++ nyelv könyvtári függvényeiről és osztályairól. Többek között bemutatjuk az adatbevitel és adatkivitel hagyományos és objektumos megoldásait, a karaktersorozatok kezelésének eszközeit, valamint a szabványos sablonkönyvtár (*Standard Template Library*) összetevőit.

Mit található a CD-mellékleten?

A könyvhöz csatolt CD-melléklet azon túl, hogy megkíméli az Olvasót a példák unalmas begépelésétől, a kitűzött feladatok megoldását is tartalmazza. Ugyancsak megtalálható itt egy tesztprogram és a ***Dev-C++ 5.0 beta 7 (4.9.7.0)*** ingyenes C++ fejlesztőeszköz. A ***Dev-C++*** szabad szoftver, amelyet a *GNU General Public License* alapján lehet terjeszteni. (Ez azt jelenti, hogy szabadon továbbadható és módosítható.)

Mitől más ez a könyv?

Könyvünk megírását évtizedes C/C++ oktatási gyakorlat előzte meg a Budapesti Műszaki Egyetem Gépészmérnöki Karán és Mérnöktovábbképző Intézetében. Ennek során szerzett tapasztalatok nagyban hozzájárultak könyvünk szerkezetének és tartalmának kialakításához. Ezúton is köszönetet szeretnénk mondani annak a több száz hallgatónak, akik közvetve ugyan, de hozzájárultak könyvünk megjelenéséhez.

Gondolatok a C++ nyelvről

A C++ hatékony, általános célú programozási nyelv. Segítségével kis alkalmazások és nagy programok egyaránt előállíthatók. A C++ hagyományos fejlesztőeszközként és objektum-orientált programozási nyelvként egyaránt használható.

A múlt

A C nyelv alapelemeinek többsége a *Martin Richards* által kifejlesztett ***BCPL*** *(Basic Combined Programming Language*, 1963) nyelvből származik. Ez a származtatás közvetett módon - a ***B*** nyelven keresztül - ment végbe. A B nyelvet *Ken Thompson* dolgozta ki 1970-ben az *AT&T Bell Laboratóriumok* cégnél, és ezen a nyelven készült el az első UNIX operációs rendszer DEC PDP-7 számítógépre.

A B nyelv azonban nem bizonyult elég hatékonynak az új PDP-11 számítógép UNIX operációs rendszerének megírására. Mivel azonban a magas szintű nyelven történő implementációról nem akartak lemondani, *Dennis Ritchie* 1971-ben hozzálátott a ***B*** nyelv új változatának kidolgozásához, amit ***C***-nek neveztek el. A felmerült problémák kiküszöbölésén kívül, Ritchie igyekezett a C nyelvbe visszahozni azokat az általános programozási elemeket, amelyek az ***Algol-60*** nyelv „karcsúsítása" során kikerültek a BCPL és B nyelvekből.

Több éven át csak az AT&T UNIX operációs rendszerrel szállított C-fordító képviselte a C nyelv definícióját. Az 1978-ban megjelent, *B.W. Kernigham* és D. M. Ritchie által írt „*The C Programming Language*" című könyv szolgált a nem UNIX-alapú C-implementációk kiindulópontjául.

A C nyelv világméretű elterjedése összefügg a mikroszámítógépek megjelenésével. Bár ebben az időben még nem volt szabványa a C nyelvnek, azonban a K&R könyvben leírt függvények használatával lehetett olyan programot írni, amely a különböző mikro- és miniszámítógépeken lefordítva ugyanúgy futott. Megjelent az igény a szabványos C nyelv megalkotására, amely a gombamód szaporodó fordítók és gépek között biztosíthatja a C nyelvű forrásprogramok hordozhatóságát (portabilitását).

A C nyelvet több lépésben szabványosították. Az első (ANSI) szabvány 1983-ban jelent meg, így alapul szolgálhatott a C++ nyelv megformálásához. A ANSI C szabvány végleges változata (ANSI X3.159-1989) hat évi munka után, 1989-ben készült el. A C szabványt 1989-ben revízió alá vették (IOS/IEC), majd 1995-ben kibővítették a széles karakterek (*wchar_t*) használatának lehetőségével.

A C nyelv történetét itt le is zárhatnánk, napjainkban újabb C-fordítók már nem születnek.

A jelen

A C nyelv helyét, szerepét világszerte a C++ nyelv veszi át, melynek kifejlesztése 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 csoda, 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:

1. A C++ nyelv legyen felülről kompatibilis az eredeti C nyelvvel.
2. 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++ szabvány kidolgozásában az 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 nem típusos nyelv, addig a C++ erősen típusos, objektum-orientált nyelv.

A C programok többsége minden további nélkül átvihető C++ rendszerbe. Mivel a C++ nyelv „erősen típusos nyelv", a szükséges módosításokat általában a konverziók és a deklarációk területén kell elvégezni.

(Egy lehetséges) jövő

Napjainkban egyre több olyan programozási nyelv születik, melyek gyökere a C/C++ nyelvekből ered. Ilyen szemszögből tekintve elmondhatjuk, hogy a C++ nyelv átvette az Algol nyelvtől azt a szerepet, hogy más nyelvek is merítenek belőle.

A jövőben a C++ elsősorban a helyi (kliens) fejlesztések eszköze lesz, míg más területeken az újabb nyelvek hódítanak. Külön is érdemes kiemelni a hálózatos (Internetes) megoldásokhoz használt eszközök közül a *Sun* cég által támogatott ***Java*** nyelvet, és *Microsoft* hasonló célú fejlesztőeszközét, a ***C#*** (*C sharp*) nyelvet.

Érdekességképpen nézzünk néhány példát a rokonság bizonyítására:

```
// kozos.cpp - C++
#include <iostream>
using namespace std;

void main() {
  for (int i=1; i<23; i++)
        if (i%3==0)
              cout<<i<<endl;
}

// Class1.cs - C#
using System;
public class Class1 {
  public static void Main(string[] args) {
        for (int i=1; i<23; i++)
              if (i%3==0)
                    System.Console.WriteLine(i);
  }
}
```

```java
// Class1.java - Java
import java.lang.*;
public class Class1 {
  public static void main (String[] args) {
        for (int i=1; i<23; i++)
              if (i%3==0)
                    System.out.println(i);
  }
}
```

```cpp
// kozos.cpp - C++
#include <iostream>
using namespace std;
class Class1 {
  public:
        static void Main() {
              for (int i=1; i<23; i++)
                    if (i%3==0)
                          cout<<i<<endl;
        }
};

void main(){
    Class1::Main();
}
```

Egyre nagyobb jelentőséggel bírnak a kiszolgáló oldali fejlesztések programozási nyelvei, mint a PHP, Perl és a Python, amelyek szintén emlékeztetnek a C++-ra.

Az áttekintő jellegű gondolatok után kezdjük el az C++ nyelvvel való ismerkedést!

I.

A C++ mint egy jobb C nyelv

1. Első találkozás C++ nyelven írt programmal

Tekintsük az alábbi egyszerű, C nyelven megírt programot, amely bekér egy szöveget és két számot, majd kiírja a szöveget és a számok szorzatát!

```
#include <stdio.h>
#include <conio.h>
void main() {
  char nev[20];
  int  a;
  double b;

  printf("Kerem a szoveget: ");
  scanf("%s",nev);
  printf("A=");
  scanf("%d",&a);
  printf("B=");
  scanf("%lf",&b);
  printf("%s : A*B=%lf\n",nev,a*b);
  getch();
}
```

A példában a I/O műveletek elvégzéséhez a szabványos C-könyvtárban található ***scanf***() és ***printf***() függvényeket használtuk. Ezen függvények nem tekinthetők a C nyelv részének, hiszen csak könyvtári függvények. A C++ nyelv a szabványos I/O műveletek kezelésére szintén tartalmaz kiegészítést, a **cin** és a **cout** adatfolyam (*stream*) objektumok definiálásával, amelyek szintén nem képezik részét a C++ nyelv definíciójának. Ezen osztályok felhasználásával a fenti program szabványos C++ nyelven elkészített változata:

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

void main()
{
  char nev[20];
  int  a;
  double b;

  cout << "Kerem a szoveget: ";
  cin  >> nev;
  cout << "A=";
  cin  >> a;
  cout << "B=";
  cin  >> b;
  cout << nev <<" : A*B=" << a*b;
  cin.get();
  cin.get();
}
```

Szembeötlő eltérés a C nyelvű programhoz képest, hogy az I/O műveletek használata egyszerűbbé vált. Nem kell figyelni a megfelelő formátum megadására, illetve a helyes paraméterezésre, mindezt elvégzi helyettünk a fordítóprogram. A C++ szabvány minden könyvtári elemet a közös *std* névterületen definiál. Emiatt a hagyományos C++ megoldás

```
#include <iostream.h>
```

helyett, a szabványos formát alkalmazzuk a példaprogramokban:

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

A szabványos input elvégzésére a ***cin***, míg a szabványos outputként a ***cout*** adatfolyam-objektumot használjuk. Létezik még egy szabványos hiba *stream* is, a ***cerr***. Mindhárom objektum definícióját az *IOSTREAM* fejállomány tartalmazza. (A 16-bites karaktereket tartalmazó szövegek kezelését a fenti objektumok *w* betűvel kezdődő párjai támogatják: ***wcout***, ***wcin***, ***wcerr***.) Az adatfolyam-osztályok lehetőségeit a későbbiekben részletesen tárgyaljuk. Az itt bemutatott szabványos adatbeviteli és adatkiviteli (I/O) műveleteket a példaprogramokban kívánjuk felhasználni.

2. A C++ nyelv alapelemei

A C++ nyelvvel való ismerkedés legelején áttekintjük a C++ programozási nyelv azon alapelemeit - a neveket, a számokat és a karaktereket - amelyekből a C++ program felépül. Az ANSI C++ szabvány nyelvhasználatával élve, ezeket az elemeket *token*nek nevezzük. A C++ forrásprogram fordításakor a fordítóprogram a nyelv tokenjeit dolgozza fel. (A tokeneket a fordító már nem bontja további részekre.). A C++ nyelv alapelemeihez tartoznak a kulcsszavak, az azonosítók, a konstansok, a sztringliterálok, az operátorok és az írásjelek.

2.1. A 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	z	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 (szövegkonstansok) belsejében.

2.2. A C++ nyelv azonosítói

A C++ nyelvű program bizonyos összetevőire (például. változókra, függvényekre, címkékre) névvel hivatkozunk. A nevek (azonosítók, szimbólumok) megfelelő megválasztása lényeges része a program írásának. Az azonosítók hossza általában implementációfüggő - a legtöbb fordító legfeljebb 32 karakteres nevek használatát támogatja. Az azonosító első karaktere betű vagy _(aláhúzásjel) lehet, míg a második karaktertől kezdődően betűk, számok és aláhúzásjelek válthatják egymást. Az azonosítók elején az aláhúzásjel 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 az alábbi 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 (makrókat).

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

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

```
__try, __property, __published
```

2.3. 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 felsorolt (**enum**) konstansok definiálásával a típusokat ismertető fejezetben részletesen foglalkozunk.

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.

2.3.1. 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.

Decimális (10-es alapú) egész számokat jelölnek azok a konstansok, amelyeknek első számjegye nem 0, például:

```
2003,   -1989,        23,          -1,          0
```

Oktális (8-as alapú) egész konstansok első jegye 0, amelyet oktális számjegyek követnek:

```
03712,  -03706,       040,         -01,         0
```

Hexadecimális (16-os alapú) egész konstansokat a *0x*, illetve a *0X* előtag különbözteti meg az előző két konstansfajtától. Az előtagot hexadecimális jegyek követik:

```
0x7cA,  -0X7c6,       0x20,        -0x1,        0
```

Mint látni fogjuk, a C++ nyelvben egész számokat különböző típusok reprezentálnak. Az egyes típusok közötti eltérés az előjel értelmezésében és a tárolási méretben jelentkezik. A konstansok megadásakor a konstans után elhelyezett betűvel írhatjuk elő a konstans értelmezését.

A fenti példákban közönséges egész számokat adtunk meg. Ha azonban előjel nélküli (**unsigned**) egészet kívánunk használni, akkor az **u** vagy az **U** betűt kell a szám után írnunk:

```
65535u,          0177777U,          0xFFFFu
```

Nagyobb előjeles egészek tárolására az ún. hosszú (**long**) egészet használjuk, amelyet a szám után helyezett **l** (kis L) vagy **L** betűvel jelölünk:

```
19871207L,       0x12f35e71
```

Utolsó lehetőségként az **U** és **L** betűket együtt használva előjel nélküli hosszú (**unsigned long**) egész konstansokat is megadhatunk:

```
3087007744UL,    0xB8000000LU
```

2.3.2. Karakterkonstansok

Az ANSI (egybájtos) karakterkonstansok egyszeres idézőjelek (' - aposztróf) közé zárt egyetlen karaktert tartalmazó konstansok:

```
'a', '1', '@', 'é', 'ab', '01'
```

Az egyetlen karaktert tartalmazó karakterkonstansok által képviselt számérték a karakter 8-bites ANSI kódja. A két karaktert tartalmazó *unicode* karakterkonstansok számértéke 16-bites:

```
L'A',  L'ab',
```

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	*BEL*	`'\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	*?*	`'\?'`
ANSI karakter oktális kóddal	*ooo*	`'\ooo'`
ANSI karakter hexa. kóddal	*hh*	`'\xhh'`

2.3.3. Lebegőpontos konstansok

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

```
1,  -2.,  100.45,  2e-3,  11E2,  -3.1415925,  31415925E-7
```

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:

```
3.1415F,                    2.7182f
```

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

```
3.141592653589793238 5L,          2.718281828459045235 41
```

2.4. Sztringkonstansok (literálok)

Az ANSI sztringliterál, amit sztringkonstansnak is szokás hívni, kettős idézőjelek közé zárt karaktersorozatot jelent:

```
"Ez egy ANSI sztringkonstans!"
```

A megadott karaktersorozatot a statikus memóriaterületen helyezi el a fordító, és ugyancsak eltárolja a sztringet záró '\0' karaktert (nullás bájtot) is. A sztringkonstans tartalmazhat *escape* szekvenciákat is, melyek esetén csak a nekik megfelelő karakter (egy bájt) kerül tárolásra:

```
"\nEz az első sor!\nA második sor!\n"
```

Egymás után elhelyezkedő sztringkonstansokat egyetlen sztringliterálként tárolja a fordító:

```
"Hosszú szövegeget két vagy" " több darabra tördelhetünk."
```

A széles karaktereket tartalmazó unicode sztringkonstansok előtt az **L** betűt kell használnunk:

```
L"Ez egy unicode sztring konstans!"
```

2.5. Megjegyzések a programban

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 a /* ... */ jeleken kívül a // (két perjel) is használható. A // jel használata esetén a megjegyzést nem kell lezárni, hatása a sor végéig terjed.

```
/* Az alábbi részben megadjuk
   a változók definícióját   */
int i=0;      /* segédváltozó */

// Az alábbi részben megadjuk
// a változók definícióját
int i=0;      // segédváltozó
```

2.6. Operátorok és írásjelek

Az operátorok olyan (egy vagy több karakterből álló) szimbólumok, amelyek megmondják, hogyan kell feldolgozni az operandusokat. Az operátorok részletes ismertetésére a további fejezetekben kerül sor. Itt csak azért teszünk róluk említést, mivel szintén a C++ nyelv alapegységei (tokenjei). A következő táblázatban minden magyarázat nélkül felsoroltuk a C++ nyelv (szabványos) operátorait:

static_cast	!	!=	%	%=	&	&&	&=	()	*	*=
const_cast	+	++	+=	,	-	--	-=	->	.	/
dynamic_cast	/=	<	<=	<<	<<=	=	==	>	>=	>>
reinterpret_cast	>>=	?:	[]	^	^=	|	|=	||	~	::
Sizeof	.*	->*	**new**	**delete**						

Az írásjelek a C++ nyelvben olyan szimbólumokat jelölnek, amelyeknek csak szintaktikai szerepük 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 argumentumlista kijelölése,
{}	Kódblokk vagy függvény behatárolása,
*	A mutatótípus jelölése a deklarációkban,
,	A függvény-argumentumok elválasztása,
:	Címke elválasztása
;	Az utasítás végének jelölése
...	Változó hosszúságú argumentumlista jelölése,
#	Előfordító direktíva jelölése.

3. A 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.

3.1. Egyetlen modulból felépülő C++ program

A C++ program fordításához 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. Az alábbi példában egyetlen forrásfájlból álló C++ program szerkezete követhető nyomon:

```
#include <iostream>          // Előfordító utasítások
#define  EGY   1
#define  KETTO 2

using namespace std;         // Globális definíciók és
int sum(int, int);           // deklarációk
int e=8;

//A main függvény definíciója
void main()
{
  int a;                     // Lokális definíciók és
  a= EGY;                    // deklarációk, utasítások
  int b = KETTO;
  e=sum(a,b);
  cout<<"Az osszeg: "<<e<<endl;
  cin.get();
}

//A sum függvény definíciója
int sum(int x, int y) {
  int z;                     // Lokális definíciók és
  z=x+y;                     // deklarációk, utasítások
  return z;
}
```

A fenti példaprogram két függvény tartalmaz. A *sum()* függvény a paraméterként kapott két számot összeadja, és függvényértékként ezt az összeget szolgáltatja.

Tároljuk a példaprogramot a *CppProg.cpp* állományban! A futtatható fájl előállítása (*translation*) két fő lépésben megy végbe:

- Az első lépés a *CppProg.cpp* forrásfájl fordítása (*compiling*), melynek során olyan közbenső állomány jön létre, amely a program adatait és gépi szintű utasításait tartalmazza. (IBM PC számítógépen ezt a fájlt tárgymodulnak (*object modul*) hívjuk, és .OBJ kiterjesztésű állományban helyezkedik el. A fordítás eredménye még nem futtatható, hiszen tartalmazhat olyan (pl. könyvtári) hivatkozásokat, amelyek még nem kerültek feloldásra.
- A második lépés feladata a futtatható állomány összeállítása (*linking*). Itt fontos szerepet játszanak a C++ szabványos függvények kódját tartalmazó könyvtárak, melyek általában .LIB kiterjesztésű állományokban helyezkednek el. (IBM PC számítógépen a keletkező futtatható programot .EXE fájl tartalmazza.)

3.2. 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. Az adatrejtés elvét a későbbiekben tárgyalásra kerülő fájlszintű érvényességi tartomány (láthatóság, *scope*), névterületek (*namespace*) és a tárolási osztályok biztosítják. Ezek megfelelő használatával a modul bizonyos nevei kívülről (*extern*) is láthatók lesznek, míg a többi név elérhetősége a modulra korlátozódik.

A több modulból álló C++ program fordításának bemutatásához az előző alfejezet példáját vágjuk ketté! A *CppProg1.cpp* fájl csak a ***main()*** függvényt és a *CppProg2.cpp* állományban elhelyezkedő *sum()* függvény leírását (prototípusát) tartalmazza. Az **extern** kulcsszó jelzi, hogy a *sum()* függvényt más modulban kell a szerkesztőnek keresnie.

```
#include <iostream>          // Előfordító utasítások
#define  EGY   1
#define  KETTO 2

using namespace std;         // Globális definíciók és
extern int sum(int, int);    // deklarációk
int e=8;

//A main függvény definíciója
void main()
{
  int a;                     // Lokális definíciók és
  a= EGY;                    // deklarációk, utasítások
  int b = KETTO;
```

```
    e=sum(a,b);
    cout<<"Az osszeg: "<<e<<endl;
    cin.get();
}
```

A *CppProg2.cpp* fájlban csak a *sum()* függvény található:

```
//A sum függvény definíciója
  int sum(int x, int y) {
  int z;                    // Lokális definíciók és
  z=x+y;                    // deklarációk, utasítások
  return z;
}
```

A futtatható fájl előállításakor minden modult külön le kell fordítanunk, (*CppProg1.obj*, *CppProg1.obj*). A keletkező tárgymodulokat a szerkesztő (*linker*) építi össze a megfelelő könyvtári hivatkozások feloldásával. A több modult tartalmazó C++ programok készítését a különböző fejlesztőeszközök a projekt (*project*) koncepció bevezetésével támogatják. A projektbe foglalt forrásállományok fordítása és szerkesztése automatikusan végbemegy. A keletkező futtatható fájl általában a projekt nevét viseli.

Általában is elmondható, hogy a C++ forrásprogram előfordító utasítások, deklarációk és definíciók, utasításblokkok és függvények kollekciója. Az egyes részek további tagolását és ismertetését a következő alfejezetekben végezzük.

4. Alaptípusok, változók és 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 azt a nevet többször is deklarálhatjuk, azonban az egymást 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.

4.1. A C++ nyelv típusai

A C++ nyelv típusait többféleképpen csoportosíthatjuk. Az első csoportosítást a tárolt adatok jellege alapján végezzük:

Csoport	*Mely típusokat tartalmazza*
Integrál (egész jellegű) típusok, (melyek felhasználhatók a **switch** utasításban.)	**bool**, **char**, **wchar_t**, **int**, **enum**
Aritmetikai típusok (elvégezhető rajtuk a négy alapművelet).	Az integrál típusok, kiegészítve a **float**, **double** és a **long double** típusokkal.
Skalár típusok (**for** ciklusváltozókhoz használhatók).	Az aritmetikai adattípusok, a referenciák és a mutatók
Aggregate (több érték tárolására alkalmas típusok).	Tömb és a felhasználói típusok (**class**, **struct**, **union**).

Az előzőeknél sokkal egyszerűbb, azonban bizonyos szempontból kibővített értelmezését jelenti a típusoknak az a csoportosítás, amely az alaptípusokat (*base types*) és a származtatott (*derived*) típusokat különbözteti meg. Az alaptípusokhoz a **char**, az előjeles és előjel nélküli egészek és a lebegőpontos típusok tartoznak (ez nem más, mint az aritmetikai típusok csoportja az **enum** típus nélkül). Az elemi adattípusok jellemzőit táblázatban foglaltuk össze:

Adattípus	Értékkészlet	Méret (bájt)	Pontosság (jegy)
bool	**false, true**	1	
char	-128..127	1	
signed char	-128..127	1	
unsigned char	0..255	1	
wchar_t	0..65535	2	
int	-2147483648..2147483647	4	
unsigned int	0..4294967295	4	
short int	-32768..32767	2	
unsigned short	0..65535	2	
long int	-2147483648..2147483647	4	
unsigned long	0..4294967295	4	
float	3.4E-38..3.8E+38	4	6
double	1.7E-308..1.7E+308	8	15
long double	3.4E-4932..3.4E+4932	10	19

A származtatott típusok csoportja az alaptípusok felhasználásával felépített tömb, függvény, mutató, osztály, struktúra és unió típusokat tartalmazza.

A csoportosítást ki kell egészítenünk, egy olyan típusnévvel, amely éppen a típus hiányát jelzi (üres típus) - ez a név a **void**. A **void** típuselőírás használatára a későbbiek során, a mutatókat és a függvényeket tárgyaló fejezetekben, visszatérünk.

4.1.1. Típuselőírások, típusmódosítók

Az alaptípusokhoz tartozó **char**, **int** és **double** típuselőírásokhoz bizonyos más kulcsszavakat (típusmódosítókat) kapcsolva, újabb típuselőírásokhoz jutunk, amelyek értelmezése eltér a kiindulási előírástól. A típusmódosítók 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 **short int** típus egy rövidebb (2 bájton tárolt), míg a **long int** típus egy hosszabb (4-bájtos) egész típust definiál. A **long double** típus az adott számítógépen értelmezett legnagyobb pontosságú lebegőpontos típust jelöli.

Az egész típusok lehetnek előjelesek (*signed*) és előjel nélküliek (*unsigned*). Az **int** típusok (**int**, **short int**, **long int**) alapértelmezés szerint pozitív és negatív egészek tárolására egyaránt alkalmasak, míg a **char** típus előjelének értelmezése

implementációfüggő. A fenti négy típus előjeles vagy előjel nélküli volta egyértelművé tehető a **signed**, illetve az **unsigned** típusmódosítók megadásával.

A típusmódosítók önmagukban típuselőírásként is használhatók. Az alábbiakban (ábécé-sorrendben) összefoglaltuk a lehetséges típuselőírásokat. Az előírások soronként azonos típusokat jelölnek.

bool
char
wchar_t
double
enum *típusnév*
float
int, signed, signed int
long double
long int, long, signed long, signed long int
signed char
short int, short, signed short, signed short int
struct *típusnév*
class *típusnév*
union *típusnév*
unsigned char
unsigned int, unsigned
unsigned long, unsigned long int
unsigned short, unsigned short int
void

4.1.2. Típusminősítők

A típuselőírásokat típusminősítővel együtt használva a deklarált azonosítóhoz az alábbi két tulajdonság egyikét rendelhetjük. A **const** kulcsszóval olyan nevet definiálhatunk, melynek értéke nem változtatható meg (csak olvasható).

A **volatile** típusminősítővel olyan név hozható létre, melynek értékét a programunktól független kód (például egy másik futó folyamat vagy szál) is megváltoztathatja. A **volatile** közli a fordítóval, hogy nem tud mindent, ami az adott változóval történhet. (Ezért például a fordító minden egyes, ilyen tulajdonságú változóra történő hivatkozáskor, a memóriából veszi fel az változóhoz tartozó értéket.)

int const
const int
volatile char
long int volatile

4.1.3. A felsorolt típus (*enum*)

Az **enum** olyan adattípust jelöl, melynek lehetséges értékei egy konstanshalmazból kerülnek ki. Az **enum** típust konstansnevek felhasználásával származtatjuk:

```
enum azonosító { felsorolás }
```

A felsorolásban szereplő konstansok az **int** típus értéktartományából vehetnek fel értéket. Nézzünk néhány példát a felsorolt típus létrehozására!

```
enum valasz { igen, nem, talan};
```

A fenti programsor feldolgozása után létrejön a felsorolt típus (**enum** *valasz* vagy *valasz*), és három egész konstans *igen*, *nem* és *talan*. A fordító a felsorolt konstansoknak balról jobbra haladva, nullával kezdve egész értékeket feleltet meg. A példában az *igen*, *nem* és *talan* konstansok értéke *0*, *1* és *2*.

Ha a felsorolásban a konstans nevét egyelőség jel és egy egész érték követi, akkor a konstanshoz a fordító a megadott számot rendeli, és a tőle jobbra eső konstansok ezen kiindulási értéktől kezdődően kapnak értéket:

```
enum valasz { igen, nem=10, talan};
```

Ekkor az *igen*, *nem* és *talan* konstansok értéke rendre *0*, *10* és *11* lesz. A felsorolásban azonos értékek többször is szerepelhetnek:

```
enum valasz { igen, nem=10, lehet, talan=10};
```

A **enum** típust elsősorban csoportos konstansdefiniálásra használjuk - ekkor a típusnevet el is hagyhatjuk:

```
enum { also=-2, felso, kiraly=1, asz };
```

ahol a konstansok értékei sorban *-2*, *-1*, *1* és *2*.

Az alábbi deklarációt használva, a programrészlet C++-ban figyelmeztetéshez vezet:

```
enum szin {fekete, kek, zold};

enum szin col;
int  kod;

col = zold;          // rendben
kod = col;           // rendben, a kod értéke 2 lesz
col = 26;            // figyelmeztetés!
col = szin(26);      // ok!
```

Az **enum** deklarációban megadott név típusnévként is használható a kulcsszó megadása nélkül:

```
enum Boolean {False, True};

Boolean x = False;
enum Boolean y = x;
```

4.2. 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. Nézzük először általános formában, hogyan néz ki a változók definíciója (deklarációja):

⟨*tárolási osztály*⟩ *típus* ⟨*típus ...* ⟩ *változónév*⟨=*kezdőérték*⟩ ⟨, .. ⟩;

Az általánosított formában a ⟨ ⟩ jelek az opcionálisan megadható részeket jelölik, míg a három pont az előző definíciós elem ismételhetőségére utal. Külön felhívjuk a figyelmet arra, hogy a deklarációs sort pontosvesszővel kell lezárni.

```
int alfa;
int beta=4;
int gamma, delta=9;
```

A deklarációban a típust megelőzheti néhány alapszó (**auto**, **register**, **static** vagy **extern**), amelyek a változó tárolásával kapcsolatban tartalmaznak előírásokat. Az előírások, amiket tárolási osztálynak nevezünk, meghatározzák a változó elhelyezkedését, láthatóságát és élettartamát. Minden változóhoz tartozik tárolási osztály, még akkor is, ha azt külön nem adtuk meg. Ha a változót a függvényeken kívül definiáljuk, akkor az alapértelmezés szerint globális (más modulból elérhető - **extern**) tárolási osztállyal rendelkezik, míg a függvényeken belül definiált változók alaphelyzetben automatikus (**auto**) változók.

Az **extern** tárolási osztályú változók akkor is kapnak kezdőértéket (nullát), ha nem adunk meg semmit. A függvényen belül definiált automatikus változókat tetszőleges kifejezéssel inicializálhatjuk.

```
int av = sin(log(rand()));
```

C++-ban (ellentétben a C nyelvvel) a globális elérésű változók kezdőértékeként szintén tetszőleges kifejezést megadhatunk:

```
long ev = (1 << 4)+exp(-1);
```

4.3. Saját típusok előállítása

A C++ nyelv típusnevei, a típusmódosítók és a típusminősítők megadásával, általában több alapszóból tevődnek össze, például:

```
volatile unsigned long int
```

Definiáljunk a fenti típus felhasználásával egy kezdőérték nélküli változót!

```
volatile unsigned long int idozites;
```

A C++ nyelv tartalmaz egy speciális tárolási osztályt (**typedef**), amely lehetővé teszi, hogy érvényes típusokhoz szinonim neveket rendeljünk:

```
typedef volatile unsigned long int tido;
```

Az új típussal már sokkal egyszerűbb az *idozites* változót definiálni:

```
tido idozites;
```

Különösen hasznos a **typedef** használata összetett típusok esetén, ahol a típusdefiníció felírása nem mindig triviális. A típusok készítése azonban mindig eredményes lesz, ha a következő tapasztalati szabály betartjuk:

- Írjunk fel egy kezdőérték nélküli változódefiníciót, ahol az a típus szerepel, amelyhez szinonim nevet kívánunk kapcsolni!
- Írjuk a definíció elé a **typedef** kulcsszót, ami által a megadott név nem változót, hanem típust fog jelölni!

4.5. 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:

```
int a;
a=7;
```

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.)

```
const int a=30;
a=7;                          // Hibajelzést kapunk a fordítótól.
```

A másik, szintén gyakran használt megoldás, amikor az előfordító ***#define*** utasításával létrehozott makrók hordoznak konstans értékeket. Az előfordító által használt neveket csupa nagybetűvel szokás írni:

```
#define FEKETE 0
#define KEK 1
#define ZOLD 2
#define PIROS 4
#define PI 3.14159265

void main()
{
int a=KEK;
double f;
  a += PIROS;
  f = 90*PI/180;
}
```

Ezek a szimbolikus nevek valójában konstans értékeket képviselnek. Az előző programrészlet az előfordítás után:

```
void main()
{
  int a=1;
  double f;
  a+=4;
  f=90*3.14159265/180;
}
```

A harmadik lehetőség, az **enum** típus használatát jelenti, ami azonban csak egész (**int**) típusú konstansok esetén alkalmazható. Az előző példában szereplő színkonstansokat az alábbi alakban is előállíthatjuk:

```
enum szinek {fekete, kek, zold, piros=4};

void main()
{
        int a = kek;
        a += piros;
}
```

Az **enum** és a **const** konstansok igazi konstansok, hisz nem tárolja őket a memóriában a fordító. Míg a ***#define*** konstansok a definiálás helyétől a fájl végéig fejtik hatásukat, addig az **enum** és a **const** konstansokra a szokásos C++ láthatósági és élettartam szabályok érvényesek. Általában is elmondható, hogy a C++ nyelvben javasolt elkerülni a ***#define*** konstansok használatát, mivel bizonyos esetekben hibát vihetnek a forrásprogramba.

4.6. Értékek, címek, mutatók és referenciák

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

```
változó = érték;
```

A C++ nyelven az értékadó operátort tartalmazó utasítás valójában egy kifejezés, amit a fordítóprogram kiértékel. 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.

4.6.1. Balérték és jobbérték

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!

```
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 jobb oldalon megadott konstans értéket be kell másolni. A második értékadás során az *a* változó az értékadás mindkét oldalán szerepel. A bal oldalon álló *a* ugyancsak a tárolót jelöli ki a memóriában (*lvalue*), míg a jobb oldalon álló *a* egy jobbérték kifejezésben szerepel, melynek értékét (13) a futó program határozza meg az értékadás elvégzése előtt.

4.6.2. Ismerkedés a mutatóval és a referenciával

A ***mutatók*** használata a C++ nyelvben alapvető követelmény. Ebben a részben csak a mutatók fogalmát vezetjük be, míg alkalmazásukkal a könyvünk további fejezeteiben részletesen foglalkozunk.

Definiáljunk egy egész típusú változót!

```
int x = 23;
```

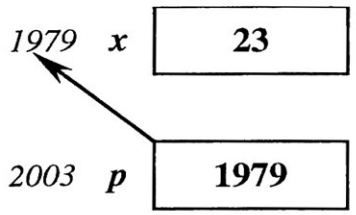
A definíció hatására a memóriában létrejön egy (**int** típusú) tároló, amelybe bemásolódik a kezdőérték. Az

```
int * p;
```

definíció hatására szintén létrejön egy tároló, melynek típusa **int***. Ez a változó **int** típusú változók címének tárolására használható, melyet a „címe” művelet során szerezhetünk meg, például:

```
p = &x;
```

A művelet után az *x* név és a **p* érték ugyanarra a memóriaterületre hivatkoznak. (A **p* kifejezés a „*p* által mutatott" tárolót jelöli.)



Ennek következtében a

```
*p = x +12;
```

kifejezés feldolgozása során az *x* változó értéke *35*-re módosul.

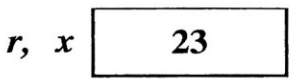
A ***hivatkozási*** (referencia) típus felhasználásával már létező változókra hivatkozhatunk, alternatív nevet definiálva. A definíció általános formája:

```
típus & azonosító = változó;
```

A referencia definiálásakor kötelező kezdőértéket adnunk. A fentiekben definiált *x* változóhoz referenciát is készíthetünk:

```
int x = 23;
int & r = x;
```

Ellentétben a mutatókkal, a referencia tárolására általában nem jön létre külön változó. A fordító egyszerűen második névként egy új nevet ad az *x* változónak (*r*).



Ennek következtében az alábbi kifejezés kiértékelése után szintén *35* lesz az *x* változó értéke:

```
r = x +12;
```

Ellentétben a *p* mutatóval, melynek értéke (ezáltal a mutatott tároló) bármikor megváltoztatható, az *r* referencia a változóhoz kötött.

Ha a referenciát *konstans* értékkel, vagy *eltérő típusú* változóval inicializáljuk, a fordító először létrehozza a hivatkozás típusával megegyező tárolót, majd ide másolja a kezdőértékként megadott kifejezés értékét.

```
int & n = '\n';
```

```
unsigned b = 2003;
int & r = b;
r++;  // b nem változik!
```

Saját mutató-, illetve referenciatípus szintén létrehozható a **typedef** tárolási osztály felhasználásával:

```
int x = 23;
typedef int & tri;
typedef int * tpi;
tri r = x;
tpi p = &x;
```

*4.6.2.1. A void * típusú általános mutatók*

Az előző példákban a pointerekkel **int** típusú változókra mutattunk. A változó eléréséhez azonban nem elegendő annak címét tárolnunk (ez van a mutatóban), hanem definiálnunk kell a tároló méretét is, amit a mutató típusa közvetít a fordító felé. A C++ nyelv típus nélküli, ún. általános mutatók használatát is lehetővé teszi:

```
int x;
void * ptr=&x;
```

amely azonban sohasem jelöl ki tárolót. Ha ilyen mutatóval szeretnénk a hivatkozott változónak értéket adni, akkor felhasználói típuskonverzióval (*cast*) típust kell rendelnünk a cím mellé, például:

```
*(int *)ptr = 23;
```

4.6.2.2. Többszörös indirektségű mutatók

A mutatókat többszörös indirektségű kapcsolatok esetén is használhatunk. Ekkor a mutatók definíciójában több csillag (*) szerepel. Tekintsünk néhány definíciót, és mondjuk meg, hogy mi a létrehozott változó!

`int x;`	*x* egy egész típusú változó.
`int * p;`	*p* egy **int** típusú mutató (amely **int** változóra mutathat).
`int **q;`	*q* egy **int*** típusú mutató (amely **int*** változóra, vagyis egészre mutató pointerre mutathat).

Már említettük, hogy a C++ nyelven megadott típusok eléggé összetettek is lehetnek. A bonyolultság feloldásában, a deklarációk értelmezésében a mutatók esetén is ajánlott a **typedef** használata:

```
typedef int *  iptr;     // egészre mutató pointer típusa
iptr p, *q;
```

vagy

```
// egészre mutató pointer típusa
typedef int *  iptr;

// iptr típusú objektumra mutató pointer típusa
typedef iptr * ipptr;

iptr p;
ipptr q;
```

A definíciók megadása után kijelenthetjük, hogy az

```
x = 23;
p = &x;
q = &p;
x = x + *p + **q;
```

utasítások lefutását követően az *x* változó értéke *69* lesz.

5. 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.

A operandusok a C++ nyelv azon elemei, amelyeken az operátorok fejtik ki hatásukat. Azokat az operandusokat, amelyek nem igényelnek további kiértékelést elsődleges (*primary*) kifejezéseknek nevezzük. Ilyenek az azonosítók, a konstans értékek, a sztringliterálok és a zárójelben megadott kifejezések. Hagyományosan az azonosítókhoz soroljuk a függvényhívásokat, valamint a tömbelem- és a struktúratag-hivatkozásokat 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:

```
op operandus          vagy          operandus op
```

Az első esetben, amikor az operátor (*op*) megelőzi az operandust előrevetett (*prefixes*), míg a második esetben hátravetett (*postfixes*) alakról beszélünk:

```
-a          a++       sizeof(a)    float(a)        &a
```

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

```
operandus1 op operandus2
```

Ebben a csoportban a hagyományos aritmetikai műveletek mellett megtalálhatók a bitműveletek elvégzésére szolgáló operátorok is:

```
a+b         a!=b         a<<2         a+=b       a & 0xff00
```

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

```
a < 0 ? -a : a
```

Az alábbi táblázatban a C++ nyelv műveleteit precedencia szerint csoportosítottuk. Az első csoportban a legmagasabb precedenciájú műveletek találhatók. (A *kif* rövidítéssel a kifejezésket jelöltük.)

1.	***Asszociativitás: balról-jobbra***
osztálynév::*tag*	hatókör-feloldás
névterület-név::*tag*	hatókör-feloldás
::*név*	globális hatókör
::*minősített_név*	globális hatókör
2.	***asszociativitás: balról-jobbra***
kif(*kif-lista*)	függvényhívás
mutató[*kif*]	tömbindexelés
mutató->tag	közvetett tagkiválasztás
objektum.tag	közvetlen tagkiválasztás
balérték++	(postfix) léptetés előre
balérték--	(postfix) léptetés vissza
típus(*kif-lista*)	érték létrehozása
dynamic_cast<*típus*>(*kif*)	futásidejű ellenőrzött típus-átalakítás
static_cast<*típus*>(*kif*)	fordításidejű ellenőrzött típus-átalakítás
reinterpret_cast<*típus*>(*kif*)	ellenőrizetlen típus-átalakítás
const_cast<*típus*>(*kif*)	konstans típus-átalakítás
typeid(*kif*)	futásidejű típusazonosítás
typeid(*típus*)	típusazonosítás
3.	***asszociativitás: jobbról-balra***
! *kif*	logikai tagadás (NEM)
~ *kif*	bitenkénti negálás
+ *kif*	+ előjel
- *kif*	- előjel
++ *balérték*	(prefix) léptetés előre
-- *balérték*	(prefix) léptetés vissza
&*balérték*	a címe operátor
**kif*	az indirektség operátor
(*típus*) *kif*	típus-átalakítás
sizeof *kif*	objektum bájtban kifejezett mérete
sizeof (*típus*)	típus bájtban kifejezett mérete
new *típus*	tárterület foglalása
new *típus*[*kif*]	tárterület foglalása tömb számára
new *típus*(*kif_lista*)	tárterület foglalása és inicializálása
new (*kif_lista*) *típus*	tárterület elhelyezése
new (*kif_lista*) *típus* (*kif_lista*)	tárterület elhelyezése és inicializálása
delete *mutató*	tárterület felszabadítása
delete [ ] *mutató*	tárterület felszabadítása

4.		***asszociativitás: balról-jobbra***
	objektum. tagmutató*	osztálytagra történő indirekt hivatkozás
	mutató-> tagmutató*	mutatóval megadott objektum tagjára való indirekt hivatkozás
5.		***asszociativitás: balról-jobbra***
	*kif * kif*	szorzás
	kif / kif	osztás
	kif % kif	maradék
6.		***asszociativitás: balról-jobbra***
	kif + kif	összeadás
	kif – kif	kivonás
7.		***asszociativitás: balról-jobbra***
	kif << kif	biteltolás balra
	kif >> kif	biteltolás jobbra
8.		***asszociativitás: balról-jobbra***
	kif < kif	kisebb
	kif <= kif	kisebb vagy egyenlő
	kif > kif	nagyobb
	kif >= kif	nagyobb vagy egyenlő
9.		***asszociativitás: balról-jobbra***
	kif == kif	egyenlő
	kif!= kif	nem egyenlő
10.-14.		***asszociativitás: balról-jobbra***
10.	*kif & kif*	bitenkénti ÉS
11.	*kif ^ kif*	bitenkénti VAGY
12.	*kif | kif*	bitenkénti kizáró VAGY
13.	*kif && kif*	logikai ÉS
14.	*kif || kif*	logikai VAGY
15.		***asszociativitás: jobbról-balra***
	kif ? kif : kif	feltételes kifejezés
16.		***asszociativitás: jobbról-balra***
	balérték = kif	egyszerű értékadás
	*balérték *= kif*	szorzat megfeleltetése
	balérték /= kif	hányados megfeleltetése
	balérték %= kif	maradék megfeleltetése
	balérték += kif	összeg megfeleltetése
	balérték -= kif	különbség megfeleltetése
	balérték <<= kif	balra eltolt bitek megfeleltetése
	balérték >>= kif	jobbra eltolt bitek megfeleltetése
	balérték &= kif	bitenkénti ÉS megfeleltetése
	balérték ^= kif	bitenkénti kizáró VAGY megfeleltetése
	balérték |= kif	bitenkénti VAGY megfeleltetése

17.		***asszociativitás: balról-jobbra***
	throw *kif*	kivétel kiváltása
18.		***asszociativitás: balról-jobbra***
	kif , *kif*	műveletsor
	Előfordító műveleti jelek	
	#	illesztés sztringbe
	##	illesztés szövegbe
	defined(*azonosító*)	makró létezésének vizsgálata

A felhasználói típushoz kapcsolódóan a C++ lehetőséget kínál az operátorok többségének átdefiniálására, túlterhelésére (*operator overloading*) az alábbi operátorok kivételével:

```
. .*      ::      ?:
```

Az átdefiniálás során az egyes operátorok új értelmezést kaphatnak, azonban a fenti táblázat szerinti precedencia és asszociativitás nem változtatható meg.

5.1. Precedencia és asszociativitás

Annak érdekében, hogy bonyolultabb kifejezéseket is helyesen tudjunk használni, meg kell ismerkednünk az elsőbbségi (precedencia) szabályokkal, amelyek meghatározzák a kifejezésekben szereplő műveletek kiértékelési sorrendjét. Az egyes operátorok közötti elsőbbségi kapcsolatot a műveletek táblázatban foglaltuk össze. A táblázat csoportjai az azonos precedenciával rendelkező operátorokat tartalmazzák. A csoportok mellett külön jeleztük az azonos precedenciájú operátorokat tartalmazó kifejezésben a kiértékelés irányát, amit asszociativitásnak (csoportosításnak) hívunk. A táblázat első csoportja a legnagyobb precedenciával rendelkező műveleteket tartalmazza.

5.1.1. Az elsőbbségi (precedencia) szabály

Az operátorok precedenciája akkor játszik szerepet a kifejezés kiértékelése során, ha a kifejezésben különböző precedenciájú műveletek találhatók. Ekkor mindig a magasabb precedenciával rendelkező operátort tartalmazó részkifejezés értékelődik ki először, amit az alacsonyabb precedenciájú műveletek végrehajtása követ. Ennek megfelelően például az

```
3 + 4 * 5
```

kifejezés értéke *23*. Ha az összeget zárójelbe tesszük, megváltoztatva ezzel a kiértékelés sorrendjét

```
(3 + 4) * 5
```

35 lesz az eredmény. Ha azonban a kiindulási kifejezésben a zárójelbe a szorzatot helyezzük

```
3 + (4 * 5)
```

akkor ismét *23*-at kapunk eredményként, hisz ebben az esetben a zárójelezés csak megerősítette, de nem változtatta meg a számítás menetét.

Az egyes operátorok működésének mélyebb ismerete nélkül nézzünk néhány példát a precedencia érvényesülésére! A precedencia-táblázat segítségével gyorsan felírhatjuk azt az ekvivalens kifejezést, ahol zárójelek felhasználásával kihangsúlyozzuk a kiértékelés menetét.

Kifejezés:	**Ekvivalens kifejezés:**
`a & b || c`	`(a & b) || c`
`a = b || c`	`a = (b || c )`
`++*p++`	`++(*(p++))`

5.1.2. A csoportosítási (asszociativitás) szabály

Egy kifejezésben több azonos precedenciájú művelet is szerepelhet. Ebben az esetben az operátortáblázat csoportszámai 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.

Például az értékadó utasítások csoportjában a kiértékelést jobbról-balra haladva kell elvégezni, így adott a lehetőség több változó együttes inicializálására:

```
int a,b,c;
a = b = c = 23;
```

Zárójelek használatával most is felírhatunk egy olyan kifejezést, amely ekvivalens a fentivel;

```
a = (b =(c = 23));
```

5.2. Mellékhatások és a rövidzár kiértékelés

Bizonyos műveletek - a függvényhívás, a többszörös értékadás és a léptetés (növelés, csökkentés) - feldolgozása során a kifejezés értékének megjelenése mellett bizonyos változók is megváltozhatnak. Ezt a jelenséget mellékhatásnak (*side effect*) hívjuk. A mellékhatások kiértékelésének sorrendjét nem határozza meg a C++ szabvány, ezért javasolt minden olyan megoldás elkerülése, ahol a kifejezés eredménye függ a mellékhatások kiértékelésének sorrendjétől, például:

```
a[i] = i++;
std::cout<<++n<<pow(2,n)<<endl;
```

A művelettáblázatból látható, hogy a logikai kifejezések kiértékelése szintén balról-jobbra haladva történik. Bizonyos műveleteknél nem szükséges a teljes kifejezést kiértékelni ahhoz, hogy egyértelmű legyen a kifejezés értéke. Példaként vegyük a logikai ÉS (&&) operátort, amely használata esetén a bal oldali operandus 0 értéke esetén a jobb oldali operandus kiértékelése feleslegessé válik! Ezt a kiértékelési módot rövidzár (*short-circuit*) kiértékelésnek nevezzük.

Ha a rövidzár kiértékelése során a logikai operátor jobb oldalán valamilyen mellékhatás kifejezés áll,

```
x || y++
```

az eredmény nem mindig lesz az, amit várunk. A fenti példában x nem nulla értéke esetén az y léptetésére már nem kerül sor.

5.3. Az elsődleges kifejezés operátorai

Az ANSI C nyelvben az elsődleges kifejezések a literálok (konstans értékek), az azonosítók és a zárójeles kifejezések voltak. A C++ nyelv ezt a listát bővíti a **this** kulcsszóval, a hatókör feloldó operátorral **::**, a nevekkel és az osztály-destruktorral ~.

literálok	egész-, karakter-, valós- és karaktersorozat-konstansok	
(kifejezés)		
this		
::	::azonosító, ::operátorfüggvény-név, ::minősített név.	
Név	azonosító, operátorfüggvény-név, konverziós függvény neve, ~osztálynév, minősített név	minősített osztálynév::név

5.4. Aritmetikai operátorok

Az aritmetikai operátorok csoportja a szokásos négy alapműveleten túlmenően a maradékképzés operátorát (%) is tartalmazza. Az összeadás (+), a kivonás (-), a szorzás (*) és az osztás (/) művelete egész és lebegőpontos számok esetén egyaránt használható. Az osztás egész típusú operandusok esetén egész osztást jelöl:

`23 / 4`	a kifejezés értéke (a hányados)	5
`23 % 4`	az kifejezés értéke (a maradék)	3

Egész *a* és *b* (nem 0) esetén mindig igaz az alábbi kifejezés:

```
(a / b) * b + (a % b) = a
```

A csoportba tartozik az egyoperandusú mínusz (-) operátor is, melynek segítségével a mögötte álló operandus értéke ellentétes előjelűre változtatható (negálható).

Aritmetikai operátorokat leggyakrabban matematikai feladatok programozása során használunk. Példaként írjuk át az

$$a + \frac{b - c}{-d \cdot e}$$

kifejezést C++ kifejezéssé! A helyes átíráshoz a műveleti jelek cseréjén túlmenően mérlegelnünk kell a precedencia-viszonyokat. Ezzel általában nincs is gond, hiszen már az általános iskolában megtanultuk, hogy a szorzás és osztás magasabb precedenciával rendelkezik az összeadásnál, illetve a kivonásnál. Tört átírásánál biztosan jól járunk el, ha a számlálót és nevezőt is zárójelek közé helyezzük.

```
a+(b-c)/(-d*e)
```

Az átírt kifejezés nevezőjében a zárójel használata most kiküszöbölhető, hisz a szorzás helyett ismételt osztást is használhatunk. Ezt figyelembe véve a matematikai alaktól eltérünk ugyan, de jól olvasható C++ kifejezéshez jutunk:

```
a-(b-c)/d/e
```

Megjegyezzük, hogy a + és a - operátorok egyik vagy mindkét operandusa mutató is lehet, ekkor pointer-aritmetikáról beszélünk. A megengedett pointer-aritmetikai műveletek, ahol a *q* és a *p* (nem **void*** típusú) mutatók, az *i* pedig egész (**int** vagy **long**):

Művelet	*Kifejezés*	*Eredmény*
két mutató kivonható egymásból	*q - p*	egész
a mutatóhoz egész szám hozzáadható	*p + i*	mutató
a mutatóból egész szám kivonható	*p - i*	mutató

5.5. Összehasonlító és logikai operátorok

A C++ nyelvben a logikai típus is az egész típusok közé tartozik – a **false** értéke 0, a **true** értéke pedig 1. Az utasításokban szereplő feltételek tetszőleges kifejezések lehetnek, melyek nulla vagy nem nulla értéke szolgáltatja a logikai hamis, illetve igaz eredményt. A feltételekben gyakran kell összehasonlítanunk bizonyos értékeket, hogy a program további működéséről döntsünk. Az összehasonlítás elvégzésére az összehasonlító operátorokat használjuk, melyeket az alábbi táblázatban foglaltuk össze:

Matematikai alak	*C++ kifejezés*	*Jelentés*
a < b	a < b	*a* kisebb, mint *b*
a ≤ b	a <= b	*a* kisebb vagy egyenlő, mint *b*
a > b	a > b	*a* nagyobb, mint *b*
a ≥ b	a >= b	*a* nagyobb vagy egyenlő, mint *b*
a = b	a == b	*a* egyenlő *b*-vel
a ≠ b	a != b	*a* nem egyenlő *b*-vel

Bármelyik fenti kifejezés **int** típusú, és a kifejezés értéke 1, ha a vizsgált reláció igaz, illetve 0, ha nem.

Példaként írjunk fel egy olyan kifejezést, amely igaz (**true**, 1) lesz, ha az *x* változó értéke nem negatív!

```
x >= 0
```

Hogyan lehet megvizsgálni azt, hogy az *x* értéke *-5* és *5* közé esik-e? A matematikából ismert módon írjuk fel a kifejezést:

```
-5 < x < 5
```

A kifejezést különböző *x* értékek esetén kiértékelve mindig *1*-et kapunk. Hol a hiba? Az összehasonlító operátorok kiértékelése balról jobbra haladva történik. Először kiértékelésre kerül a *-5 < x* kifejezés, melynek értéke *0* vagy *1* lesz. A második lépésben ezt a *0*-át vagy *1*-et hasonlítjuk össze *5*-tel, így a kisebb reláció mindig igaz lesz.

Ahhoz, hogy bonyolultabb feltételeket is össze tudjunk hasonlítani, a relációs operátorok mellett szükségünk van a logikai operátorokra is. C++-ban a logikai *ÉS* (*&&*) és a logikai *VAGY* (||) műveletek használhatók a feltételek megfogalmazása során. Az előző feltétel egyszerűen felírható a logikai *ÉS* művelet felhasználásával:

```
-5 < x && x < 5
```

(Zárójeleket nem szükséges használnunk, hisz az összehasonlító operátorok magasabb precedenciával rendelkeznek a logikai operátoroknál.) Újra felhívjuk a figyelmet a „rövidzár" kiértékelésre, ami azt jelenti, hogy a balról-jobbra haladó kiértékelés azonnal leáll, ha a kifejezés logikai értéke egyértelműen eldönthető.

Vannak esetek, amikor valamely feltétel felírása helyett egyszerűbb az ellentett feltételt megfogalmazni és alkalmazni rá a logikai tagadás (*NEM*) operátorát (*!*). Az előző példában alkalmazott feltétel egyenértékű az alábbi feltétellel:

```
!(-5 >= x || x >= 5)
```

A logikai tagadás során minden relációt az ellentétes irányú relációra, az *ÉS* operátort pedig a *VAGY* operátorra (illetve fordítva) cseréljük.

A C++ programokban gyakran használjuk az

`ok == 0` kifejezés helyett a `!ok`

`ok != 0` kifejezés helyett az `ok`

kifejezést. Nem igazán lehet egyértelműen eldönteni, hogy melyik alakot érdemesebb alkalmazni. A második megoldás jobban olvasható („nem ok", „ok"), de bizonyos esetekben nehezebb megérteni az *ok* változó szerepét.

Mint láttuk a relációs operátorok mellett szükségünk van a logikai operátorokra is. C++-ban a logikai *ÉS* (&&), a logikai *VAGY* (||) és a logikai *NEM* (!) műveletek használhatók a feltételek megfogalmazása során. A logikai operátorok működését ún. igazságtáblával írjuk le:

a	*!a*
0	1
1	0

logikai tagadás

a	*b*	*a&&b*
0	0	0
0	1	0
1	0	0
1	1	1

logikai ÉS művelet

a	*b*	*a||b*
0	0	0
0	1	1
1	0	1
1	1	1

logika VAGY művelet

5.6. Léptető operátorok

A változók értékét léptető operátorok a magas szintű nyelvekben csak ritkán fordulnak elő. C++ nyelv lehetőséget biztosít valamely változó értékének eggyel való növelésére `++` (*increment*), illetve eggyel való csökkentésére `--` (*decrement*).

A léptető operátorok az aritmetikai típusokon kívül a mutatókra is alkalmazhatók, ahol azonban nem 1 bájttal való elmozdulást, hanem a szomszédos elemre való léptetést jelentik.

Az operátorok csak balérték operandussal használhatók, azonban mind az előrevetett, mind pedig a hátravetett forma alkalmazható:

```
int a;

// prefixes alakok:
++a;                    --a;

// postfixes alakok:
a++;                    a--;
```

Ha az operátorokat a fent bemutatott módon használjuk, nem látszik különbség az elővevetett és hátravetett forma között, hiszen mindkét esetben a változó értéke léptetődik. Ha azonban az operátort bonyolultabb kifejezésben alkalmazzuk, akkor a prefixes alak használata esetén a léptetés a kifejezés kiértékelése előtt megy végbe, és a változó az új értékével vesz részt a kifejezés kiértékelésében:

```
int x, n=5;
x = ++n;
```

A példában szereplő kifejezés kiszámítása után mind az *x*, mind pedig az *n* változó értéke *6* lesz.

```
double x, n=5.0;
x = n++;
```

A kifejezés feldolgozása után az *x* változó értéke *5.0*, míg az *n* változó értéke *6.0* lesz.

A léptető operátorok működését jobban megértjük, ha a bonyolultabb kifejezésket részkifejezésekre bontjuk. A

```
int a=2, b=3, c;
c = ++a + b--;          // a 3, b 2, c pedig 6 lesz
```

kifejezés az alábbi (egy, illetve többutasításos) kifejezésekkel megegyező eredményt szolgáltat:

```
a++, c=a+b, b--;            a++;  c=a+b;  b--;
```

Az eggyel való növelés és csökkentés hagyományos megadása

```
a = a + 1;
a = a - 1;
```

helyett mindig érdemes a megfelelő léptető operátort használni,

```
++a;     vagy   a++;
--a;     vagy   a--;
```

amely az áttekinthetőség mellett, gyorsabb kód létrehozását is eredményezi.

5.7. Bitműveletek

A C++ nyelv hat operátort tartalmaz, amelyekkel különböző bitenkénti műveleteket végezhetünk **char**, **short**, **int** és **long** típusú előjeles és előjel nélküli adatokon.

5.7.1. Bitenkénti logikai műveletek

A műveletek első csoportja, a bitenkénti logikai műveletek, lehetővé teszik,hogy biteket teszteljünk, töröljünk vagy beállítsunk:

Operátor	*Művelet*
~	1-es komplemens
&	bitenkénti *ÉS*
|	bitenkénti *VAGY*
^	bitenkénti kizáró *VAGY*

A bitenkénti logikai műveletek működésének leírását az alábbi táblázat tartalmazza, ahol a 0 és az 1 számjegyek a törölt, illetve a beállított bitállapotot jelölik.

a	*b*	*a & b*	*a | b*	*a ^ b*	*~a*
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

A bitenkénti logikai műveletek a C++ nyelv szintjén biztosítják a számítógép hardverelemeinek programozását. A perifériák többségének alacsony szintű vezérlése bizonyos bitek beállítását, illetve törlését jelenti. Ezeket a műveleteket összefoglaló néven „maszkolásnak" nevezzük.

Minden egyes művelethez megfelelő bitmaszkot kell készítenünk, amellyel aztán logikai kapcsolatba hozva a megváltoztatni kívánt értéket, végbemegy a kívánt bitművelet. Mielőtt sorra vennénk a szokásos bitműveleteket, meg kell ismerkednünk az 1-, 2- és 4-bájtos adatelemek bitjeinek sorszámozásával. A bitek sorszámozása mindig a legkisebb helyértékű bittől indulva 0-val kezdődik, és balra haladva növekszik. Mivel a példáinkban **short int** típusú adatelemeket használunk, nézzük meg ezen adatok felépítését:

15.	14.	13.	12.	11.	10.	9.	8.	7.	6.	5.	4.	3.	2.	1.	0.
0	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1

A fenti szám értéke hexadecimális számrendszerben 0x07D3 illetve decimális alakban 2003.

Az alábbi példákban a **short int** típusú *2525* szám *4.* és *13.* bitjeit kezeljük:

```
short int a = 2525;
```

Művelet	*Maszk*	*C++ utasítás*	*Eredmény*
Bitek 1-be állítása	0010 0000 0001 0000	`a = a | 0x2010;`	`0x29dd`
Bitek törlése	1101 1111 1110 1111	`a = a & 0xdfef;`	`0x09cd`

Bitek negálása	0010 0000 0001 0000	`a = a ^ 0x2010;`
		`a = a ^ 0x2010;`
			`0x29cd`
			`(10701)`
			`0x09dd`
			`(2525)`

Az összes bit negálása		`a = ~a;`	`0xf622`

Felhívjuk a figyelmet a kizáró vagy operátorra (^). Ha ugyanazzal a maszkkal kétszer végezzük el a műveletet, visszakapjuk az eredeti számot, esetünkben a 2525-öt.

5.7.2. Biteltoló műveletek

A bitműveletek másik csoportjába, a biteltoló (*shift*) operátorok tartoznak. Az eltolás balra (<<) és jobbra (>>) egyaránt elvégezhető. Az eltolás során a bal oldali operandus bitjei annyiszor lépnek balra (jobbra), amennyi a jobb oldali operandus értéke. A felszabaduló bitpozíciókba 0-ás bitek kerülnek, míg a kilépő bitek elvesznek.

```
short int a;
```

Értékadás	*Bináris érték*	*Művelet*	*Eredmény* *decimális (hexa)* *bináris*
`a=2525;`	`0000 1001 1101 1101`	`a=a<<2;`	`10100 (0x2774)` `0010 0111 0111 0100`
`a=2525;`	`0000 1001 1101 1101`	`a=a>>3;`	`315 (0x013b)` `0000 0001 0011 1011`
`a=-2525;`	`1111 0110 0010 0011`	`a=a>>3;`	`-316 (0xfec4)` `1111 1110 1100 0100`

Az eredményeket megvizsgálva láthatjuk, hogy az 1 bittel való balra eltolás során az *a* változó értéke kétszeresére (2^1) nőtt, míg két lépéssel jobbra eltolva, *a* értéke negyed (2^2) részére csökkent. Általánosan is megfogalmazható, hogy valamely egész szám bitjeinek *n* lépéssel történő balra tolása a szám (2^n) értékkel való megszorzását eredményezi. Az *m* bittel való jobbra eltolás pedig (2^m) értékkel elvégzett egész osztásnak felel meg.

Írjunk egy olyan programot, amely a beolvasott 16-bites egész számot két bájtra bontja! A feladat aritmetikai és bitenkénti műveletek felhasználásával egyaránt megoldható. Nézzünk egy lehetséges megoldást, amelyben bitműveleteket használtunk!

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

void main()
{
  short int num;
  unsigned short lo, hi;

  // A szám beolvasása
  cout<<"\nKerek egy egesz szamot [-32768,32767] : ";
  cin>>num;
  cout<<"A szam = "<<num<<endl;

  // Az alsó bájt meghatározása maszkolással
  lo=num & 0x00FF;
  cout<<"A szam also bajtja: "<<lo<<endl;

  // Az felső byte meghatározása biteltolással
  hi=num >> 8;
  cout<<"A szam felso bajtja: "<<hi<<endl;
  cin.get();
  cin.get();
}
```

5.8. Értékadó operátorok

Már említettük, hogy C++ nyelvben az értékadás egy olyan kifejezés, amely a bal oldali operandus által kijelölt tárolónak adja a jobb oldalon megadott kifejezés értékét, másrészt pedig ez az érték egyben az értékadó kifejezés értéke is. Ebből következik, hogy értékadás tetszőleges kifejezésben szerepelhet.

Ha az *a* és *b* **int** típusú változók, akkor az értékadás hagyományos formái

```
a = 23;
b = (a+12)*4-9;
```

során az *a* változó értéke *23*, míg a *b* változóé *131* lesz. Felírható azonban olyan, más nyelvektől idegen kifejezés is,

```
b=2*(a=4)-5;
```

ahol az *a* (*4*) és *b* (*3*) változók egyaránt értéket kapnak.

Az értékadó operátorok kiértékelése jobbról-balra haladva történik. Emiatt C++ nyelven használható a többszörös értékadás, melynek során több változó veszi fel ugyanazt az értéket:

```
a = b = 26;
```

Az értékadások gyakran használt formája, amikor egy változó értékét valamilyen művelettel módosítjuk, és a keletkező új értéket tároljuk a változóban:

```
a = a + 2;
```

Az ilyen alakú kifejezések tömörebb formában is felírhatók:

```
a += 2;
```

Általában elmondható, hogy a

kif_1 = kif_1 op kif_2

alakú kifejezések felírására az ún. összetett értékadás műveletét is használhatjuk:

kif_1 op = kif_2

A két felírás egyenértékű, attól a különbségtől eltekintve, hogy a második esetben a bal oldali kifejezés kiértékelése csak egyszer történik meg. Operátorként (*op*) az eddig megismert kétoperandusú műveleteket használhatjuk:

Hagyományos forma	*Tömör forma*
`a = a + b`	`a += b`
`a = a - b`	`a -= b`
`a = a * b`	`a *= b`
`a = a / b`	`a /= b`
`a = a % b`	`a %= b`
`a = a << b`	`a <<= b`
`a = a >> b`	`a >>= b`
`a = a & b`	`a &= b`
`a = a | b`	`a |= b`
`a = a ^ b`	`a ^= b`

Meg kell jegyeznünk, hogy az

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

kifejezésnek megfelelő hagyományos értékadás az

```
a = a * (b + 1);
```

nem pedig az

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

Az összetett értékadás használata általában gyorsabb kódot eredményez, és könnyebben értelmezhetővé teszi a forrásprogramot.

5.9. Pointerműveletek

A C++ nyelvben található két olyan speciális egyoperandusú művelet, amelyeket mutatókkal kapcsolatban használunk. A „címe" (*&*) művelet eredménye az operandusként megadott tároló címe:

```
int a=23, *ptr;
ptr = &a;
```

A „címe" operátort arra használjuk, hogy mutatóinkat már meglévő változókra irányítsuk. A másik mutatóoperátor (*) az indirekt hivatkozás elvégzéséhez szükséges:

```
*ptr = *ptr + 5;
```

A **ptr* kifejezés a *ptr* pointer által mutatott tárolót jelöli.

A mutatókkal a fentieken túlmenően további (aritmetikai) műveleteket is végezhetünk. A pointer léptetése a szomszédos elemre többféle módon is elvégezhető:

```
int *p, *q;
...
p  = p + 1;
p += 1;
p++;
++p;
```

Az előző elemre való visszalépésre szintén több lehetőség közül választhatunk:

```
p  = p - 1;
p -= 1;
p--;
--p;
```

A két mutató különbsége, vagyis a két mutató között elhelyezkedő elemek száma szintén meghatározható:

```
int h  = p - q;
```

A léptető és az indirektség operátorok együttes használatához kellő óvatosság szükséges. (A példában az *sp* mutatóval egy konstans karaktersorozatra mutatunk.)

```
char *sp = "Lafenita";
```

A definíció után az *sp* mutató a sztring első karakterére, az *'L'* betűre mutat. Mi lesz a helyzet a

```
cout<<++*sp<<endl;
cout<<sp<<endl;
```

utasítások végrehajtása után? A precedencia-táblázatban azt találjuk, hogy az indirektség és az előrevetett léptető operátorok azonos precedenciával rendelkeznek. A kérés megválaszolásánál az asszociativitási szabályt, a kiértékelés jobbról-balra történő menetét kell figyelembe vennünk. Ennek alapján először az indirektség operátorát értelmezi a rendszer. Ezt követi a léptetés operátorának feldolgozása, melynek hatására az első kiírás során az *'M'* betű jelenik meg, míg a második utasítással a „*Mafenita*" karakterlánc lesz látható.

A * és a ++ operátorok lehetséges elrendezéseit, amelyek a fentihez hasonló gondolatmenettel értelmezhetők, táblázatba rendeztük. Az első oszlop a kifejezéseket tartalmazza, míg a második és a harmadik oszlopban a két kiírás eredménye látható.

Kifejezés	*Karakter*	*Szöveg*
`*++sp`	`a`	`afenita`
`*sp++`	`L`	`afenita`
`++*sp`	`M`	`Mafenita`
`(*sp)++`	`L`	`Mafenita`
`++*sp++`	`M`	`afenita`

5.10. A sizeof operátor

A C++ nyelv tartalmaz egy olyan fordítás idején kiértékelésre kerülő egyoperandusú operátort, amely tetszőleges változó, típus, kifejezés méretét megadja. A

```
sizeof változó

sizeof (típusnév)
```

alakú kifejezések értéke egy olyan egész szám, amely megegyezik a megadott változó, illetve típus bájtban kifejezett méretével.

A változó tetszőleges egyszerű változó, tömb vagy objektum egyaránt lehet:

```
int a;
double d;

a = sizeof d;                    // A d változó mérete
```

A típusnév az alaptípusokon túlmenően tetszőleges származtatott típust is jelölhet:

```
a = sizeof (double);          // A double típus mérete
a = sizeof (char *);          // A mutató típus mérete
```

5.11. A vessző operátor

Egyetlen kifejezésben több, akár egymástól független kifejezés is elhelyezhető, a vessző operátor felhasználásával. A vessző operátort tartalmazó kifejezés balról-jobbra haladva értékelődik ki, a kifejezés értéke és típusa megegyezik a jobb oldali operandus értékével, illetve típusával.

Példaként tekintsük az

```
x = (y = 2 , y + 3);
```

kifejezést! A kiértékelés a zárójelbe helyezett vessző operátorral kezdődik, melynek során először az *y* változó kap értéket (*2*), majd pedig a zárójelezett kifejezés (*2+3=5*). Végezetül az *x* változó értékadással megkapja az *5* értéket.

A vessző operátort gyakran használjuk különböző változók kezdőértékének egyetlen utasításban (kifejezésben) történő beállítására:

```
int x, y;
double z;

x = 12, y = 23, z = 1.2345 ;
```

Ugyancsak a vessző operátort kell használnunk, ha két változó értékét egyetlen utasításban kívánjuk felcserélni (harmadik változó felhasználásával):

```
int a=12, b=23, c;

c = a, a = b, b = c;
```

Felhívjuk a figyelmet arra, hogy azok a vesszők, amelyeket a deklarációkban a változónevek, illetve a függvényhíváskor az argumentumok elkülönítésére használunk ***nem*** a vessző operátorok. Ezért ezekben az esetekben nem garantált a balról-jobbra haladó kiértékelési sorrend.

5.12. A feltételes operátor

A feltételes operátor (?:) három operandussal rendelkezik:

kif_1 ? kif_2 : kif_3

A feltételes kifejezésben először a kif_1 kifejezés kerül kiértékelésre. Amennyiben ennek értéke nem nulla (igaz), akkor a kif_2 értéke adja a feltételes kifejezés értékét. Ellenkező esetben a kettőspont után álló kif_3 értéke lesz a feltételes kifejezés értéke. Ily módon a kettőspont két oldalán álló kifejezések közül mindig csak az egyik értékelődik ki. A feltételes kifejezés típusa a nagyobb pontosságú részkifejezés típusával egyezik meg. Az

```
(n > 0) ? 3.141534 : 54321L;
```

kifejezés típusa, függetlenül az *n* értékétől mindig **double** lesz. A feltételes operátort a legkülönbözőbb célokra használhatjuk. Az esetek többségében a feltételes utasítást (**if**) helyettesítjük vele. A következő két példában az *a* és *b* értékek közül kiválasztjuk a nagyobbat:

Megoldás az **if** utasítás felhasználásával:

```
if (a > b)
   z = a;
 else
   z = b;
```

Feltételes kifejezéssel sokkal tömörebben oldható meg a feladat:

```
z = a > b ? a : b;
```

Nézzünk két jellegzetes példát a feltételes operátor alkalmazására! Az első esetben a *ch* karakter kiírásakor a felhasználandó formátumot feltételes kifejezéssel adjuk meg. Ha a *ch* karakter vezérlő karakter (kódja < 32), akkor a hexadecimális kódját, ellenkező esetben pedig magát a karaktert írjuk ki:

```
printf(ch < 32 ? "%02X\n" : "%2c\n", ch);
```

Az alábbi kifejezés segítségével a 0 és 15 közötti értékeket hexadecimális számjeggyé alakíthatjuk:

```
ch = n >= 0 && n <= 9 ? '0' + n : 'A' + n - 10;
```

5.13. Az érvényességi kör (hatókör) operátor

Az érvényességi kör operátor kettős szerepet tölt be a C++ nyelvben. A :: operátor segítségével a program tetszőleges blokkjából hivatkozhatunk a globális (fájl szintű) hatókörrel rendelkező nevekre.

```
int i = 1223;
void main() {
   double i = 3.14159265;
   {
      long i = 2, a;
      a=i*::i;    // az a változó értéke 2*1223=2446
      ::i=94;     // az ::i változó értéke 94
   }
}
```

Az érvényességi kör operátort használjuk akkor is, amikor valamely osztály adattagjaira, illetve tagfüggvényeire hivatkozunk. Ugyancsak ezt az operátort alkalmazzuk, ha valamely névterületen definiált neveket a **using namespace** deklarációt nélkül kívánjuk elérni:

```
#include <iostream>

void main()
{
  std::cout<<"C++ nyelv"<<std::endl;
  std::cin.get();
}
```

5.14. A new és a delete operátorok használata

A szabad memória dinamikus használata alapvető részét képezi minden C++ nyelven megírt programnak. A C programokban könyvtári függvényekkel végezhetjük el a szükséges memóriafoglalási (***malloc***(),...) illetve felszabadítási (***free***()) műveleteket. A C++ nyelvben a **new** és **delete** operátorok nyelvdefiníció szintjén helyettesítik a fenti könyvtári függvényeket.

A **new** operátor az operandusában megadott típusnak megfelelő méretű területet foglal a szabad memóriában, és a területre mutató pointert ad eredményül:

```
int * ip;
ip = new int;
```

A **delete** operátor a **new** operátor által lefoglalt területet felszabadítja:

```
delete ip;
```

A **new** segítségével nem csak egyetlen elemnek, hanem több egymás után elhelyezkedő elemnek is helyet foglalhatunk a memóriában. Ez így létrejövő adatstruktúrát dinamikus tömbnek nevezzük. (A hagyományos, statikus tömböket könyvünk 7. fejezetében tárgyaljuk.)

Nézzünk néhány példát a **new** művelet használatára! Először foglaljunk tárterületet egy 10-elemű egész tömb számára!

```
int *ap;
ap = new int [10];
```

A lefoglalt terület felszabadítására a **delete[]** operátort kell használnunk:

```
delete[] ap;
```

Nézzük meg, mi változik, ha 10-elemű egészre mutató pointertömböt kell dinamikusan létrehoznunk!

```
int **pa;
pa = new int * [10];
...
delete[] pa;
```

A memória-foglalásnál, főleg amikor nagyméretű dinamikus tömb számára kívánunk tárterületet foglalni, előfordulhat, hogy nem áll rendelkezésünkre elegendő memória. A C++ futtatórendszer ezt a helyzetet a ***bad_alloc*** kivétel (*exception*) létrehozásával jelzi. Így a kivételkezelés eszköztárát használva tehetjük programunkat biztonságossá.

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

void main() {
   long * data, size;
   cout << "\nKerem a tomb meretet: ";
   cin >> size;
   cin.get();

   // Memória-foglalás
   try {
     data = new long [size];
   }
   catch (bad_alloc) {
      // Sikertelen foglalás
      cerr << "\nNincs eleg memoria !\n" << endl;
      cin.get();
      exit(-1);  // Kilépünk a programból
   }
   // ...
   // A lefoglalt memória felszabadítása
   delete[] data;
   cin.get();
}
```

Amennyiben nem kívánunk élni a kivételkezelés adta lehetőségekkel, a **new** operátor után meg kell adnunk a ***nothrow*** memóriafoglalót, melynek hatására az operátor a kivétel létrehozása helyett 0 értékkel tér vissza.

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

void main() {
   long * data, size;
   cout << "\nKerem a tomb meretet: ";
   cin >> size;
   cin.get();

   // Memória-foglalás
   data = new (nothrow) long [size];
   // A foglalás sikerességének ellenőrzése
   if (!data) {
      cerr << "\nNincs eleg memoria !\n" << endl;
      cin.get();
      exit(-1);  // Kilépünk a programból
   }

   // ...

   // A lefoglalt memória felszabadítása
   delete[] data;
   cin.get();
}
```

A ***malloc()*** és a ***free()*** függvények kompatibilitási megfontolások miatt továbbra is elérhetők a C++ nyelvből, de helyettük ajánlott a **new** és a **delete** operátorokat alkalmazni. Ha a **new** és a **delete** operátorokkal foglalunk helyet valamely osztályobjektum számára, akkor a rendszer automatikusan meghívja az osztály konstruktorát a **new**, illetve a destruktorát a **delete** művelet végrehajtásakor.

További kellemes lehetősége a C++ nyelvnek, hogy sikertelen memóriafoglalás esetén egyetlen kezelőfüggvény bevezetésével elvégezhetők a szükséges lépések (például üzenet kiírása). Ehhez a ***set_new_handler()*** függvény segítségével kell az új kezelőfüggvényt beállítanunk.

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

// A new operátor új hibakezelője:
void mem_kezelo() {
   cerr << "\nNincs eleg memoria!";
   cin.get();
   exit(1);          // kilépés a programból
}
```

```
void main(void)
{
  set_new_handler(mem_kezelo); // Az új hibakezelő beállítása
  char *ptr = new char[10000000];
  cout << "\nElso memoria-foglalos: ptr = "<<hex<<long(ptr);
  ptr = new char[1900000000];
  cout << "\nMasodik memoria-foglalás: ptr = "<<hex<<long(ptr);
  // ...
  cin.get();
  set_new_handler(0); // Az eredeti hibakezelés visszaállítása.
}
```

Tudnunk kell azonban, hogy a *mem_kezelo()* függvényben elvégzett tevékenység után (például memória-felszabadítás) újra végrehajtódik a memóriafoglalás. Ennek elkerülése érdekében az ***exit()*** függvény hívásával kilépünk a programból.

A alfejezet végén a **new** operátor további lehetőségére kívánjuk felhívni a figyelmet. A **new** után közvetlenül egy zárójelben megadott mutató is állhat, melynek hatására az operátor a mutató címével tér vissza (vagyis nem foglal memóriát):

```
int *p=new int(12);
int *q=new(p) int(23);

char *s1="Nata";
char *s2=new(s1) char;
```

A fenti példákban a *q* a *p* által mutatott területre, míg az *s2* az *s1* karaktersorozatra hivatkozik. A mutatók eltérő típusúak is lehetnek:

```
long a=0x19792003;
char *p=new(&a) char;
```

5.15. Futásidejű típusazonosítás

C++-ban a **typeid** operátor egy **type_info** típusú objektum (*typeinfo*) referenciáját adja vissza. A **type_info** objektum az operandus típusáról tartalmaz információkat (***Run-Time Type Information***).

```
typeid (kifejezés)
typeid (típusazonosító)
```

A **typeid** operátor segítségével futásidejű típusazonosítást végezhetünk. (Hibás esetben ***bad_typeid*** kivétel keletkezik.)

```
#include <typeinfo>
using namespace std;
...
if (typeid(A) == typeid (B))
    cout << "A es B tipusa: " << typeid(A).name();
```

5.16. Típuskonverziók

A kifejezések kiértékelése során előfordulhat, hogy valamely kétoperandusú operátor különböző típusú operandusokkal rendelkezik. Ahhoz azonban, hogy a művelet elvégezhető legyen, a fordítónak azonos típusúra kell átalakítania a két operandust, vagyis típuskonverziót kell végrehajtania.

A típuskonverziók egy része automatikusan, a programozó beavatkozása nélkül megy végbe, a C++ nyelv definíciójában rögzített szabályok alapján. Ezeket a konverziókat *implicit* vagy *automatikus* konverzióknak nevezzük.

Explicit típuskonverziót azonban a programozó is előírhat a C++ programban, a típuskonverziós operátorok (*cast*) felhasználásával.

5.16.1. Implicit típuskonverziók

Általánosságban elmondható, hogy az automatikus konverziók során a „szűkebb" operandus információ-vesztés nélkül konvertálódik a "szélesebb" operandus típusára. Az alábbi kifejezés kiértékelése során az **int** típusú *i* operandus **float** típusú lesz, ami egyben a kifejezés típusát is jelenti:

```
int i=20, j;
float f=3.45;
i + f;
```

Az implicit konverziók azonban nem minden esetben mennek végbe információ-vesztés nélkül. Az értékadás és a függvényhívás paraméterezése során tetszőleges típusok közötti konverzió is előfordulhat. Ha a fenti példában az összeget a *j* változónak feleltetjük meg, vagy argumentumként átadjuk az ***abs()*** függvénynek

```
j = i + f;
```

akkor bizony adatvesztés lép fel, hiszen az összeg törtrésze elvész, és *23* lesz a *j* változó, illetve a függvényparaméter értéke.

A következőkben röviden összefoglaljuk a az *x op y* alakú kifejezések kiértékelése során automatikusan végrehajtódó konverziókat.

1. A **char**, a **wchar_t**, a **short**, a **bool** és az **enum** (felsorolt) típusú adatok automatikusan **int** típussá konvertálódnak. Ha az **int** típus nem alkalmas az értékük tárolására, akkor **unsigned int** lesz a konverzió céltípusa. Ez a konverziós szabály az „egész konverzió" (*integral promotion*) nevet viseli. Mivel a fenti konverziók érték- és előjelhelyes eredményt adnak, értékmegőrző konverzióknak is szokás nevezni azokat.

2. Ha az első lépés után a kifejezésben különböző típusok szerepelnek, életbe lép a típusok hierarchiája szerinti konverzió:

   int < unsigned < long < unsigned long < float < double< long double

   A típus-átalakítás során a „kisebb" típusú operandus a „nagyobb" típusúvá konvertálódik. Az átalakítás során felhasznált szabályok a „szokásos aritmetikai konverziók" nevet viselik.

A C++ implicit mutatókonverzióval is rendelkezik. Tetszőleges típusú mutató átalakítható általános (**void***) mutatótípussá. Ellenkező irányú konverzióhoz explicit típus-átalakítást kell használnunk. Ugyancsak érdemes megjegyezni, hogy a nulla (0) számértékkel tetszőleges típusú mutató inicializálható.

```
int x=5;
void * p=&x;
int *q=0;
q=static_cast<int*>(p);
cout<<*q<<endl;
```

5.16.2. Explicit típusátalakítások

A fordítás közben végrehajtódó (statikus), gyakran használt átalakítások kijelölésére több lehetőség közül is választhatunk:

típus-átalakítás	*(típusnév) kifejezés*	`(char *)p`
érték létrehozása	*típusnév (kifejezés)*	`int(a)`
ellenőrzött típus-átalakítás	**static_cast**<*típusnév*>*(kifejezés)*	**static_cast**<char *>(p)

Vegyük sorra a szabványos C++ nyelv típusátalakítási lehetőségeit, amelyek finomítják a C++ nyelv korábbi típusmódosító megoldásait!

Konstans típusátalakítás

```
const_cast < típus > (arg)
```

A **const_cast** operátort akkor használjuk, ha fel kívánjuk oldani a **const** és/vagy **volatile** típusmódosítók hatását egy adott változó esetén Ez a típusmódosítás fordítási időben történik. A **const_cast** használatakor a *típus* és az *arg* azonos típusúak kell legyenek. Az alábbi példában egy nem konstans **int** argumentumot váró függvényt hívunk **const** argumentummal:

```
const int cvaltozo=10;
fv(const_cast<int> (cvaltozo));
```

Dinamikus típusátalakítás

```
dynamic_cast < típus > (ptr)
```

A dinamikus típus-átalakítással futásidejű konverziókat végezhetünk. Ha a típusátalakítás nem hajtható végre, akkor kifejezés 0 értékű lesz, és ***bad_cast*** kivétel jön létre. A *típus* osztályra mutató pointer, referencia vagy **void*** típusú kifejezés, míg a *ptr* operandus pointer vagy referencia típusú kifejezés lehet.

Fontos megjegyeznünk, hogy a **dynamic_cast** végrehajtásához ún. futásidejű típusazonosításra (*run time type identification* - RTTI) van szükség.

„Veszélyes" típusátalakítások

```
reinterpret_cast < típus > (arg)
```

A kifejezésben a *típus* mutató, referencia, numerikus típus, függvénymutató vagy osztálytagra mutató pointer lehet. Az operátor segítségével például egy mutatót egész típusúvá, illetve egy egész típusú kifejezést mutatóvá alakíthatunk. Két inkompatibilis mutatótípus közötti konverzió elvégzésére is használható a **reinterpret_cast**.

```
int i =reinterpret_cast <int> (&x);
```

Statikus típusátalakítások

```
static_cast < típus > (arg)
```

A **static_cast** segítségével jól definiált, hordozható és visszafordítható típuskonverziókat hajthatunk végre. Mind a *típus*, mint pedig az *arg* típusának fordítási időben ismertnek kell lennie.

```
char ch =static_cast <char> ('A'+1.0);
```

Mivel a fenti esetekben a típusnév megjelenik a konverziós előírásban, *explicit* típuskonverzióról beszélünk.

5.17. Bővebben a konstansokról

A C nyelvben létezik ugyan a **const** definíció, de valójában ez is egy változó, amelyhez nem enged közvetlen hozzáférést a fordító (indirektet azonban igen).

```
const int ci = 1979;
int * pi;

ci  = 2003;  // Hibás C-ben és C++-ban
pi  = &ci;   // csak C++-ban hibás
*pi = 2003;
```

A C++ fordító sokkal szigorúbban ellenőrzi a **const** típusú konstansok felhasználását:

```
const double pi=3.14159265; // szabályos konstansdefiníció
```

Konstansra csak megfelelő mutatóval hivatkozhatunk:

```
const double ac=12.23;
double * pd = &ac;              // Hiba!

const double *pdc;              // double konstansra mutató pointer
const double dc=12.23;
double   d;
pdc  = &dc;                     // a pdc pointer a dc-re mutat
pdc  = &d;                      // a pdc pointert a d-re állítjuk
// A pointer segítségével a d változó sem változtatható meg.
*pdc = 9.4;                     // Hiba!
```

Konstans értékű pointer:

```
int ev;
int * const aktev=&ev;
// Az aktev pointer értéke nem változtatható meg,
// de a *aktev módosítható!
*aktev = 2003;
aktev = &ev;                    // Hiba!
```

Konstansra mutató konstans értékű pointer:

```
const int ev=1979;
const int ae=2003;
// int típusú konstansra mutató konstans pointer
const int * const aktev=&ev;
aktev = &ae;                    // Hiba!
*aktev= 2525;                   // Hiba!
```

Ugyancsak a **const** típusú konstansok használata mellett szól az a lehetőség, hogy statikus tömbök definíciójában is felhasználhatók:

```
const maxnum=100;
int   num[maxnum];
```

6. 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**,
Iterációs (ciklus) utasítások:	**do**, **for**, **while**

Az utasítások részletes tárgyalásánál nem a fenti csoportosítást követjük, hiszen az egyes utasítások használatakor különböző csoportokban elhelyezkedő utasításokat kell alkalmaznunk.

6.1. Utasítások és blokkok

Tetszőleges kifejezés utasítás lesz, ha pontosvesszőt (;) helyezünk mögé:

```
kifejezés;
```

A *kifejezés utasítás* végrehajtása a kifejezésnek az 5. fejezetben ismertetett szabályok szerint történő kiértékelését jelenti. Mielőtt a következő utasításra kerülne a vezérlés, a teljes kiértékelés (a mellékhatásokkal együtt) végbemegy. Nézzünk néhány kifejezés utasítást:

```
x = y + 3;              // értékadás
x++;                    // az x növelése 1-gyel
x = y = 0;              // többszörös értékadás
fv(arg1, arg2);         // void típusú függvény hívása
y = z = f(x) +3;        // függvényt hívó kifejezés
```

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 ({ és }) 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, melynek általános formája:

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

a következő három esetben használunk:

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

Az utasításblokkon belül az utasításokat és a definíciókat/deklarációkat tetszőleges sorrendben megadhatjuk. Felhívjuk a figyelmet arra, hogy blokkot nem kell pontosvesszővel lezárni.

A következő példa összetett utasításában felcseréljük az *a* és *b* változók értékét, amennyiben *a* nagyobb, mint *b*:

```
int a, b;
. . .
if ( a > b )
{
   int c = a;
       a = b;
       b = c;
}
```

6.2. Az if utasítás

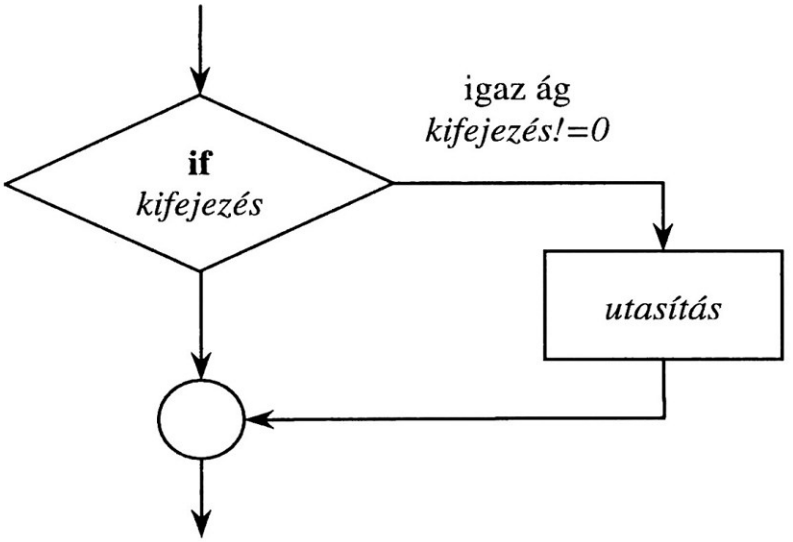
A C++ nyelv két lehetőséget biztosít a program kódjának feltételhez kötött végrehajtására - az **if** és a **switch** utasításokat. Az **if** utasítás segítségével valamely tevékenység (*utasítás*) végrehajtását egy *kifejezés* (feltétel) értékétől tehetjük függővé. Az **if** alábbi formájában az *utasítás*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

```
if (kifejezés)
        utasítás
```

A következő példaprogram egyetlen karaktert olvas a billentyűzetről az <*Enter*> gombbal lezárva. Ha a karakter az *escape* (<*Esc*>), akkor kilépés előtt hangjelzést ad, ellenkező esetben a program egyszerűen befejezi a futását.

```
#include <iostream>
using namespace std;
const int ESC = 27;

void main(void)
{
  char ch;
  cout<<"Kerek egy karaktert: ";
  ch=cin.get();
  if (ch == ESC)
     cout<<"\aEsc"<<endl;
}
```

A különböző vezérlési szerkezetek működésének grafikus szemléltetésére blokkdiagramot szokás használni. Az egyszerű **if** utasítás feldolgozását az alábbi ábrán követhetjük nyomon:



Mivel az **if** utasítás feltétele egy numerikus kifejezés nem nulla voltának tesztelése, a kód kézenfekvő módon egyszerűsíthető, ha az

```
if (kifejezés != 0)
```

helyett az

```
if (kifejezés)
```

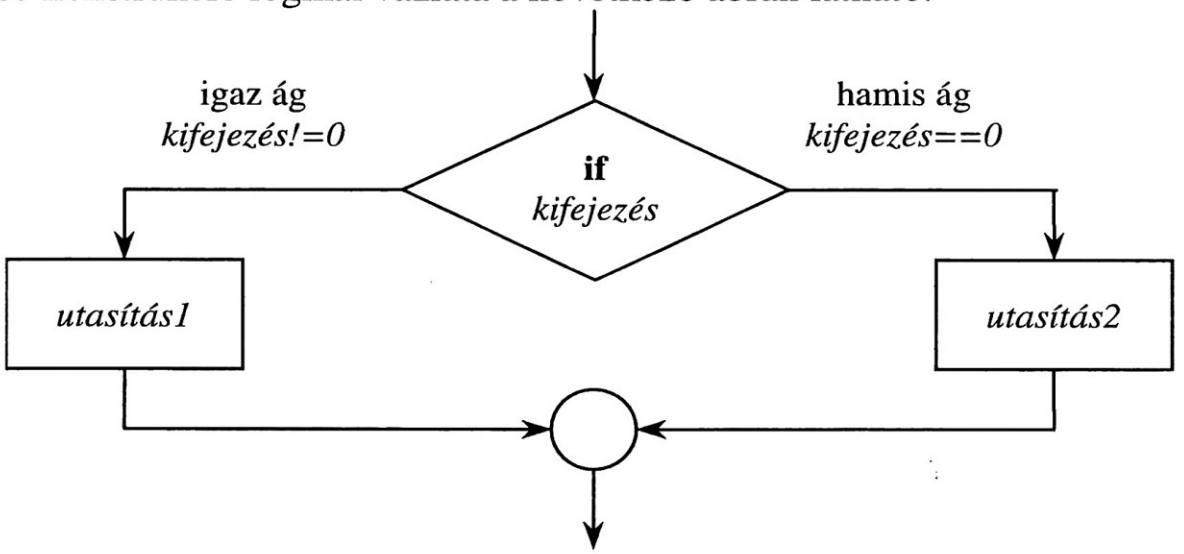
formát használjuk. Ez a megoldás általában világos, de bizonyos esetekben rejtélyesnek tűnhet. Külön felhívjuk a figyelmet arra, hogy a feltétel-kifejezést körülvevő zárójelet mindig ki kell tenni.

6.2.1. 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 zérus (hamis, **false**). Ha az *utasítás1* és az *utasítás2* nem összetett utasítások, akkor pontosvesszővel kell őket lezárni.

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

Az *if-else* konstrukció logikai vázlata a következő ábrán látható.



Az alábbi példában a beolvasott egész számról **if** utasítás segítségével döntjük el, hogy páros vagy páratlan:

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

void main()
{
  cout<<"Kerek egy egesz szamot: ";
  int n;
  cin>>n;
  cin.get();
  if (n % 2 == 0)
     cout<<"A szam paros!"<<endl;
  else
     cout<<"A szam paratlan!"<<endl;
  cin.get();
}
```

Az **if** utasítások egymásba is ágyazhatók. Ilyenkor azonban óvatosan kell eljárnunk, hisz a fordító nem mindig úgy értelmezi az utasítást, ahogy mi gondoljuk. Az alábbi példában azt várjuk az **if**-es szerkezettől, hogy a megadott egész számról megmondja, hogy az negatív páros szám-e, vagy nem negatív szám. A megoldás

```
if (n < 0)
  if (n % 2 == 0)
     cout<<"Negativ paros szam."<<endl;
else
     cout<<"Nem negativ szam."<<endl;
```

azonban nem működik helyesen, hiszen a negatív páratlan számokat is nem negatívnak mondja. Hol a hiba? A példában az **else**-ágat a külső **if**-hez kívántuk kapcsolni, azonban a fordító minden **if** utasításhoz a hozzá legközelebb eső **else** utasítást rendeli. A helyes működéshez kétféleképpen is eljuthatunk. Az egyik lehetőség, ha a belső **if** utasításhoz egy üres utasítást (;) tartalmazó **else**-ágat kapcsolunk:

```
if (n < 0)
  if (n % 2 == 0)
     cout<<"Negativ paros szam."<<endl;
  else ;
else
  cout<<"Nem negativ szam."<<endl;
```

A másik járható út, ha a belső **if** utasítást kapcsos zárójelek közé, azaz utasításblokkba helyezzük:

```
if (n < 0) {
    if (n % 2 == 0)
       cout<<"Negativ paros szam."<<endl;
}
else
   cout<<"Nem negativ szam."<<endl;
```

6.2.2. Az else-if szerkezet

Az egymásba ágyazott **if** utasítások gyakran használt formája, amikor az **else**-ágakban szerepel az újabb **if** utasítás:

```
if (kifejezés)
   utasítás
else if (kifejezés)
   utasítás
else if (kifejezés)
   utasítás
else
   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. Bizonyos esetekben nincs szükség alapértelmezés szerinti tevékenység végrehajtására - ekkor az

```
else
   utasítás
```

elhagyható. Külön kiemelést érdemel az **else-if** programrészlet olvashatósága. Az alábbi példában az *n* számról eldöntjük, hogy az negatív, nulla, vagy pozitív:

```
if (n > 0)
    cout<<"Pozitiv szam"<<endl;
else if (n==0)
    cout<<"Nulla"<<endl;
else
    cout<<"Negativ szam"<<endl;
```

Az **else-if** szerkezet speciális esete, amikor a felhasznált kifejezések egyenlőség-vizsgálatokat (==) tartalmaznak. Az alábbi példában egyszerű kalkulátort valósítottunk meg, amely a négy alapművelet elvégzésére képes. A program indítása után a kívánt műveletet például

```
12.23+34.45
```

alakban kell megadni. A programban nem vizsgáljuk nullával való osztást, ennek kezelését a futtató rendszerre bíztuk.

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

const char PROMPT = ':';

void main()
{
  double a,b,e;
  char op;
  cout.put(PROMPT);                  // a készenléti jel
  cin>>a>>op>>b;                     // beolvasás
  cin.get();
  if (op == '+')                     // összeadás?
       e = a + b;
  else if (op == '-')                // kivonás?
       e = a - b;
  else if (op == '*')                // szorzás?
       e = a * b;
  else if (op == '/')                // osztás?
       e = a / b;
```

```
    else {                              // hibás művelet!
         cerr<<"Hibas muvelet!"<<endl;
         cin.get();
         return;                        // kilépés a programból
    }
    cout<<a<<op<<b<<'='<<e<<endl;
    cin.get();
}
```

6.3. 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** (eset) címkére, amelyben a *konstans_kifejezés* értéke megegyezik a kiértékelt *kifejezés* értékével - 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. Ha a **switch** utasítás után álló utasítással kívánjuk folytatni a program futását, akkor a **break** utasítást használjuk.

A **switch** használatára tipikus példa az előző alfejezet kalkulátor programjának átírt változata:

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

const char PROMPT = ':';
```

```
void main() {
  double a,b,e;
  char op;
  cout.put(PROMPT);
  cin>>a>>op>>b;
  cin.get();
  switch (op) {
    case '+':
       e = a + b;
       break;
    case '-':
       e = a - b;
       break;
    case '*':
       e = a * b;
       break;
    case '/':
       e = a / b;
       break;
    default:
       cerr<<"Hibás muvelet!"<<endl;
       cin.get();
       return;
  } // a switch blokkjának vége
  cout<<a<<op<<b<<'='<<e<<endl;
  cin.get();
}
```

A **return** utasítással, mind a **switch** utasításból, mind pedig a ***main()*** függvényből kilépünk. Általában a **default** címke utáni utasításokat is **break**-kel zárjuk, azon egyszerű oknál fogva, hogy a **default** bárhol elhelyezkedhet a **switch** utasításon belül.

A következő példában azt mutatjuk be, hogyan lehet több esethez ugyanazt a programrészletet rendelni. A programban a válaszként beolvasott karaktert feldolgozó **switch** utasításban az *'i'* és *'I'*, illetve az *'n'* és *'N'* esetekhez tartozó **case** címkéket egymás után helyeztük el.

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

void main()
{
  cout<<"A valasz [I/N]?";
  char valasz=cin.get();

  switch (valasz) {
    case 'i':  case 'I':
       cout<<"A valasz IGEN."<<endl;
       break;
```

```
        case 'n':
        case 'N':
           cout<<"A valasz NEM."<<endl;
           break;
        default:
           cout<<"Hibas valasz!"<<endl;
           break;
     }
  }
```

6.4. A ciklusutasítások

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 (*kifejezés1*$_{opt}$; *kifejezés2*$_{opt}$; *kifejezés3*$_{opt}$) *utasítás*

do *utasítás* **while** (*kifejezés*)

A **for** utasítás esetén az *opt* index arra utal, hogy a megjelölt kifejezések használata opcionális.

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

Ezzel szemben a **do** ciklus legalább egyszer mindig lefut, hisz a vezérlőfeltétel ellenőrzése az *utasítás* végrehajtása után történik - *hátultesztelő* 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:

```
for (;;) utasítás;

while (1) utasítás;

do utasítás while (true);
```

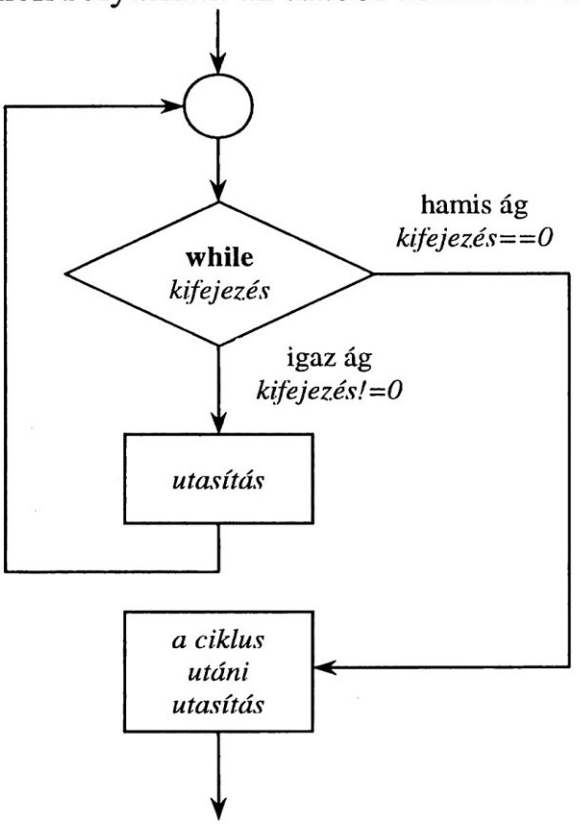
A ciklusokból a vezérlőfeltétel hamis értékének bekövetkezte előtt is ki lehet ugrani (a végtelen ciklusból is). Erre a célra további utasításokat biztosít a C++ nyelv, mint a **break**, a **return**, illetve a ciklus törzsén kívülre irányuló **goto**. A ciklus törzsének bizonyos utasításait átugorhatjuk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

6.4.1. 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
```

A **while** ciklus működésének folyamata az alábbi ábrán követhető nyomon.



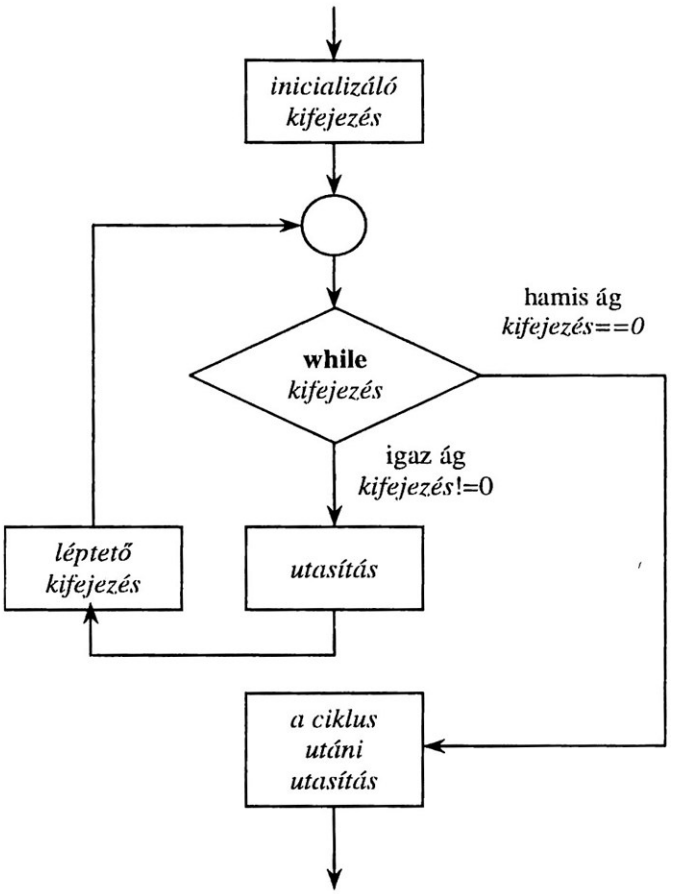
A következő példaprogramban meghatározzuk az első *n* egész szám összegét:

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

void main()
{
  long sum;
  int  n = 2003;

  cout<<"Az elso "<<n<<" egesz szam ";
  sum=0;
```

```
    while (n>0) {      // vagy :            //  vagy :
       sum += n;       // while (n>0)       //  while (sum += n--,n);
       n--;            //    sum += n--;
    }
    cout<<"osszege: "<<sum<<endl;
    cin.get();
}
```

6.4.2. A for ciklus

A **for** utasítást általában akkor használjuk, ha a ciklusmagban megadott utasítást adott számszor kívánjuk végrehajtani. A **for** ciklus működési vázlatát az alábbi ábra tartalmazza:



A **for** utasítás általános alakjában külön megjelöltük az egyes kifejezések szerepét:

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

A **for** utasítás valójában a **while** utasítás speciális alkalmazása, így a fenti **for** ciklus minden további nélkül átírható **while** ciklussá:

```
init_kif;
while (feltétel_kif) {
    utasítás
    léptető_kif;
}
```

Megjegyezzük azonban, hogy a **while** és a **for** ciklusok eltérően viselkednek, ha a ciklus törzsében a később ismertetésre kerülő **continue** utasítást használjuk.. Az ábrán jól nyomonkövethetők a **for** ciklus végrehajtásának lépései:

1. Megtörténik az *init_kif* kifejezés (amennyiben megadtuk) kiértékelése, melynek során a ciklusban használt változókat inicializáljuk. Semmilyen megkötés nincs az *init_kif* típusára vonatkozóan.
2. A következő lépésben a *feltétel_kif* (aritmetikai vagy mutató típusú) kifejezést dolgozza fel a rendszer (ha megadtuk). A *feltétel_kif* függvényében a ciklus az alábbi három eset valamelyikének megfelelően működhet:
   - Ha a *feltétel_kif* értéke igaz (nem nulla, **true**), akkor végrehajtódik az utasítás melyet a *léptető_kif* kifejezés kiértékelése követ. Minden további iterációt a *feltétel_kif* kiértékelése nyit meg, és a *léptető_kif* kiértékelése zár.
   - Ha a *feltétel_kif* nincs megadva, akkor annak értékét igaznak tételezi fel a rendszer, és a ciklus futása pontosan megegyezik az előző esetnél tárgyalttal. Ebben az esetben a ciklus leállítása csak a **break**, a **return** vagy a **goto** utasítások segítségével oldható meg.
   - Ha a *feltétel_kif* értéke hamis (0, **false**), akkor a **for** utasítás befejezi működését, és a vezérlés a program következő utasítására kerül.

Példaként a **for** ciklusra, írjuk át a **while** ciklussal megoldott, egész számok összegét meghatározó programot! Azonnal látható, hogy a megoldásnak ez a változata sokkal áttekinthetőbb és egyszerűbb:

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

void main()
{
  long sum;
  int  i, n = 2003;

  cout<<"Az elso "<<n<<" egesz szam ";
  for (i=1, sum=0 ; i<=n ; i++)
      sum += i;
  cout<<"osszege: "<<sum<<endl;
  cin.get();
}
```

A példában szereplő ciklus magjában csak egy kifejezés utasítás található, ezért az alábbi tömörítési lépések elvégezhetők:

```
for (i=1, sum=0 ; i<=n ; sum += i, i++) ;
```

illetve

```
for (i=1, sum=0 ; i<=n ; sum += i++) ;
```

A ciklusokat egymásba is ágyazhatjuk, hisz a ciklus *utasítása* újabb ciklusutasítás is lehet. A következő példában kettős ciklust használunk a megadott méretű szorzótábla megjelenítésére:

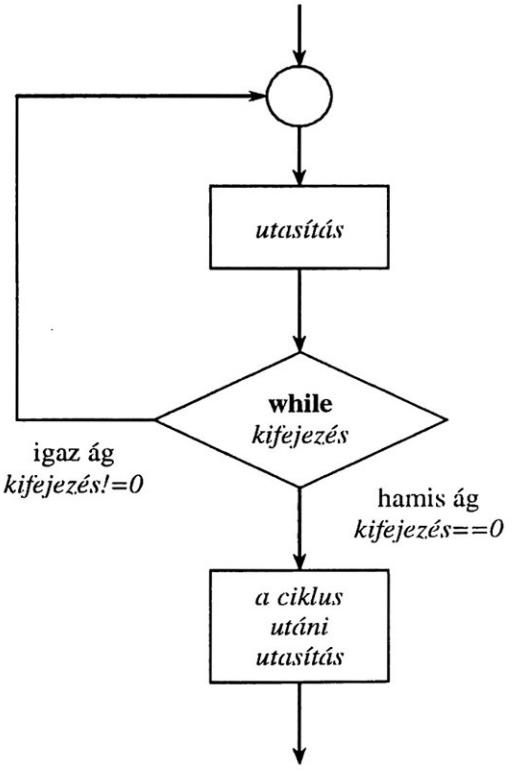
```
#include <iostream>
using namespace std;

const int MAX = 8;

void main()
{
   int i,j;

   for (j=1; j<=MAX; j++)
     cout<<'\t'<<j;
   cout<<endl;

   for (i=1; i<=MAX; i++)
   {
     cout<<i;
     for (j=1; j<=MAX; j++)
        cout<<'\t'<<i*j;
     cout<<endl;
   }
   cin.get();
}
```

6.4.3. 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. Ha a *kifejezés* értéke igaz (*nem 0*, **true**), akkor új iteráció kezdődik, míg hamis (*0*, **false**) érték esetén a ciklus befejezi működését. A **do-while** ciklus működését a mellékelt ábrán blokkdiagram segítségével ábrázolhatjuk:



Első példaként készítsük el az egész számokat összegző programnak a **do-while** ciklust használó változatát! Gyakorlatilag semmit sem kell megváltoztatnunk, a ciklus átszervezésén kívül:

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

void main(){
  long sum;
  int  n = 2003;

  cout<<"Az elso "<<n<<" egesz szam ";
  sum=0;
  do {                              // vagy : do
     sum += n;                      //             sum += n--;
     n--;                           //        while (n>0);
  } while (n>0);                    //
  cout<<"osszege: "<<sum<<endl;
  cin.get();
}
```

A második példánk egy játékprogram, amelyet futtatva a felhasználónak 0 és 100 közötti számot kell kitalálnia. A program **do-while** ciklusa egy **else-if** szerkezetet is tartalmaz, amely a felhasználó válaszát értékeli.

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

void main(){
  int szam, tipp, lepes = 0;
  // a véletlenszám-generátor inicializálása
  srand(unsigned(time(NULL)));
  szam = rand() % 101;
  cout<<"Gondoltam egy szamot 0 és 100 kozott..."<<endl;
  do {
     cout<<"Tipp: ";
     cin>>tipp;
     cin.get();
     if (tipp == szam)
        cout<<"\nKitalalta, a gondolt szam: "<<szam<<endl;
     else if (tipp > szam)
        cout<<"..Nem talalt, tul nagy!"<<endl;
     else
        cout<<"..Nem talalt, tul kicsi!"<<endl;
     lepes++;
  } while (tipp != szam);

  cout<<"Gratulalok, "<<lepes<<" lepesben sikerult!"<<endl;
  cin.get();
}
```

A példában a 0 és 100 közé eső gondolt szám előállítását a „pszeudovéletlen" számot szolgáltató ***rand ()*** függvénnyel végezzük:

```
szam = rand() % 101;
```

Ahhoz azonban, hogy minden futáskor új véletlen szám keletkezzen, a véletlenszám-generátort véletlenszerűen (az idővel) inicializáljuk a

```
srand(unsigned(time(NULL)));
```

hívás segítségével.

6.5. A break és a continue 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.

6.5.1. 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. A **break switch** utasításban történő felhasználását már bemutattuk 6.3. fejezetben, most csak a ciklusból való kiugrásra mutatunk példákat.

Az alábbi ciklus akkor lép ki, ha megtalálja a legnagyobb olyan 2003-nál kisebb egész számot, amely 123-mal osztható:

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

void main(){
  int n=2003;
  while (n>0) {
    if (!(n % 123))
       break;
    n--;
  }
  if (n)
     cout<<n<<endl;
  cin.get();
}
```

A következő példában egy konstans karaktersorozatra mutató pointert addig léptetünk végig a sztringen, míg szóköztől eltérő karaktert nem találunk. Ezzel a megoldással a szöveget bevezető szóközöket kiiktathatjuk a karaktersorozatból, így a program végén kiíratva a *p* által mutatott szöveget, a szóközök nem jelennek meg. A **p!=\0'* feltétel azt biztosítja, hogy a ciklus akkor is leáll, ha a sztringen végighaladva egyetlen szóköztől eltérő karaktert sem találunk.

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

void main(){
  char *q = "       A C++ programozasi nyelv";
  char *p;

  for (p = q; *p!='\0'; p++)
       if (*p != ' ')
          break;
  cout<<p<<endl;
  cin.get();
}
```

Felhívjuk a figyelmet arra, hogy egymásba ágyazott ciklus és **switch** utasítások esetén, mindig csak a legbelső utasításból lép ki a **break** utasítás. Az alábbi kis program kettős **for** ciklus segítségével derékszögű háromszög alakban írja ki a decimális számjegyeket:

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

void main(){
  int i,j;
  for (i = 0; i<10; i++){
     cout<<endl;
     for (j=-1 ; ; j++){
        if (i == j) break;
        cout<<setw(3)<<i;
     }
  }
  cout<<endl;
  cin.get();
}
```

```
0
1  1
2  2  2
3  3  3  3
4  4  4  4  4
5  5  5  5  5  5
6  6  6  6  6  6  6
7  7  7  7  7  7  7  7
8  8  8  8  8  8  8  8  8
9  9  9  9  9  9  9  9  9  9
```

6.5.2. 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.

A következő példában a **continue** segítségével értük el, hogy a *0*-tól *99*-ig egyesével lépkedő ciklusban csak a 7-tel vagy 11-gyel osztható számok jelenjenek meg:

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

void main()
{
  int i;
  for (i=0; i<100; i++) {
    if (i %7 && i%11)
       continue;
    cout<<i<<endl;
  }
  cin.get();
}
```

A **break** és a **continue** utasítások gyakori használata rossz programozói gyakorlatot jelent. Érdemes mindig átgondolnunk, hogy nem lehet-e az adott programszerkezetet **break**, illetve **continue** nélkül megvalósítani.

Az alábbi ciklus, amely az *<Enter>* lenyomásáig olvas és ír ki karaktereket

```
while ((ch = getchar()) != EOF)
{
  if (ch == '\n')
      break;
  putchar(ch);
}
```

minden további nélkül átírható a **break** felhasználása nélkül:

```
while ( (ch = getchar()) != EOF && ch != '\n')
  putchar(ch);
```

Hasonló a helyzet a **continue** utasítással. A következő ciklus a *<Ctrl+Z>* és *<Enter>* billentyűk lenyomásáig minden beírt karaktert visszaír a képernyőre, kivéve a soremelés karaktert:

```
while ((ch = getchar()) != EOF)
{
   if (ch == '\n')
       continue;
   putchar(ch);
}
```

A fenti működést a **continue** utasítás nélkül megvalósító ciklus:

```
while ( (ch = getchar()) != EOF)
   if (ch != '\n')
       putchar(ch);
```

6.6. A goto utasítás

A strukturált, jól áttekinthető (így valószínűleg hibátlan) programszerkezet kialakítása során nem szabad **goto** utasítást használnunk. A **goto** utasítás ugyanis kuszává, áttekinthetetlenné teszi a forrásprogramot. Vannak esetek azonban, amikor a **goto** segítségével jutunk el legegyszerűbben a megoldáshoz.

A **goto** utasítás felhasználásához utasításcímkével kell megjelölnünk azt az utasítást, ahova később ugrani szeretnénk. Az utasításcímke valójában egy azonosító, amelyet kettősponttal határolunk el az utána álló utasítástól:

```
azonosító: utasítás
```

A **goto** utasítás, amellyel a fenti címkével megjelölt sorra adhatjuk a vezérlést:

```
goto azonosító;
```

Szükséges megjegyeznünk, hogy a **goto** utasításnak és a célcímkének egyazon függvényen belül kell elhelyezkednie.

Nézzünk egy olyan példát, ahol a **goto** használata nélkül igen bonyolult programszerkezet áll elő! A feladat az, hogy két egymásba ágyazott ciklusból lépjünk ki, ha egy bizonyos feltétel bekövetkezik. Az egyszerű változatban a **goto** utasítást használjuk:

```
for ( ... ) {
  for ( ... ) {
       if ( ... )
          goto stop;
       . . .
  }
}

stop:
   cout<<"Hiba van a programban!"<<endl;
```

Amennyiben **goto** nélkül kívánjuk megoldani a fenti feladatot, segédváltozót kell bevezetnünk:

```
kesz = false;
for ( ... ) {
  for ( ... ) {
       if ( ... ) {
            kesz = true;
            break;
       }
       . . .
  }
  if (kesz) break;
}
cout<<"Hiba van a programban!"<<endl;
```

6.7. 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 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, 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 fejezetben foglalkozunk.

6.8. Kivételek kezelése

Kivételnek (*exception*) nevezzük azt a hibás állapotot vagy eseményt, amely megszakítja az alkalmazás szabályszerű futását. A kivételkezelés lehetővé teszi, hogy a C++ program futása során a vezérlés és az információ automatikusan ahhoz a programrészhez kerüljön, amelyik „hajlandó" az adott típusú kivétel kezelésére. A **throw** utasítás segítségével tetszőleges típusú kivételt (*exception*) továbbíthatunk („dobhatunk") a kezelőhöz (*exception handler*), amelyik azt elkapva (**catch**) elvégzi az esemény feldolgozását.

A C++ nyelv a kivételek kezelését a megszakításos (*termination*) modell alapján végzi. Ez azt jelenti, hogy a kivételt kiváltó programrészlet (függvény) futása megszakad az esemény bekövetkeztekor. A kivételkezelést általában hibakezelés, illetve hibatűrő (*fault-tolerant*) programok kialakításakor alkalmazzuk.

A C++ nyelv típusorientált kivételkezelései lehetőségeit az alábbi három elem jellemzi:

- kivételkezelés alatt álló programrészlet kijelölése (**try**-blokk),
- a kivételek továbbítása (**throw**),
- a kivételek „elfogása" és kezelése (**catch**).

A szabványos C++ nyelvben viszonylag kevés beépített kivétel érhető el, azonban a különböző osztálykönyvtárak előszeretettel alkalmazzák a kivételeket a hibás állapotok jelzésére. Az *exception* fejállomány tartalmazza az ***exception*** osztály leírását, mely a különböző logikai és futás-idejű kivételek alaposztálya.

A kivételkezelést a programunk adott részén belül lokálisan kell kialakítanunk a **try**, a **throw** és a **catch** kulcsszavak felhasználásával. A kivételkezelés csak a **try**-blokknak (próbálkozás-blokknak) nevezett utasításban megadott (illetve az abból hívott) kód végrehajtása esetén fejti ki hatását. A kijelölt kódrészleten belül a **throw** utasítással

```
throw kifejezés;
```

adhatjuk át a vezérlést a kifejezés típusának megfelelő kezelőnek (*handler*), amelyet a **catch** kulcsszót követően adunk meg.

```
try // kivételfigyelés alatt álló utasítások
{
     utasítások;
}
catch (kivétel_1_deklaráció)  // a kivétel_1 kezelője
{
     utasítások;
}
catch (kivétel_2_deklaráció)  // a kivétel_2 kezelője
{
     utasítások;
}
```

Függvények esetén a függvény fejlécében megadhatjuk, hogy mely kivételek továbbítódjanak a függvényen kívüli kezelőhöz:

```
// minden kivétel továbbítódik
int fv1();

// csak char* típusú kivételek továbbítódnak
int fv2() throw(char *);

// egyetlen kivétel sem továbbítódik
int fv3() throw();
```

A beérkező kivételek a típusuk alapján a megfelelő kezelőhöz irányítódnak:

```
catch (char *s)          // a char* kivétel kezelője
{
       // a kezelő törzse
}

catch (...)              // minden más nem kezelt kivétel kezelője
{
      // a kezelő törzse
}
```

Amikor egy kivételt a **throw** utasítással továbbítunk, akkor az utasításban megadott kifejezés értéke átmásolódik a **catch** utasítással kijelölt kezelő paraméterébe, így a kezelőben lehetőség nyílik ezen érték feldolgozására.

Amikor egy olyan kivételt továbbítunk, amelynek nincs kezelője, a futtató rendszer a ***terminate()*** függvényt aktivizálja, amely kilépteti a programunkat (***abort()***). Ha egy olyan kivételt továbbítunk, amelyik nincs benne az adott függvény által továbbítandó kivételek listájában, akkor az ***unexpected()*** rendszerhívás állítja le a programunkat. Mindkét kilépési folyamatba beavatkozhatunk saját kezelők definiálásával, melyek regisztrációját az *except* fejállományban deklarált ***set_terminate()***, illetve ***set_unexpected()*** függvényekkel végezhetjük el.

Az alábbi példában a számológép programunkat felkészítettük a kivételes események (kilépés, hibás adatbevitel stb.) kezelésére. A megoldásban függvényeket használtunk, melyekkel a 8. fejezetben ismerkedünk meg részletesen.

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

void beolvas(double & a, double &b, char &op)  throw(bool, char*)
{
   char p[100];
   cout<<":";

   // biztonságos adatbevitel
   cin.getline(p,100);
   int db=sscanf(p,"%lf%c%lf",&a, &op, &b);

   if (toupper(p[0])=='X')  // kilépés
       throw true;
   if (db!=3)
       throw "Hibás adatbevitel.";
}
```

```
double szamol(double a, double b, char op) throw (int, char *)
{
   double e=0;
   switch (op) {
      case '+': e=a + b;
                break;
      case '-': e=a - b;
                break;
      case '*': e=a * b;
                break;
      case '^': e=pow(a,b);
                break;
      case '/': if (!b)
                   throw 1;
                 else
                   e=a / b;
                break;
       default: throw "Hibás operátor";
   }
   return e;
}

void main()
{
   double x,y;
   char op;
   while (true) {
      try {
          beolvas(x,y,op);
          cout<<'='<<szamol(x,y,op)<<endl;
      }
      catch (bool) {
          cout<<"Viszlat!"<<endl;
          break;   // while
      }
      catch (int) {
          cout<<"Szamolasi hiba."<<endl;
      }
      catch (char *s) {
          cout<<s<<endl;
      }
      catch (...) {
          cout<<"Ismeretlen kivetel"<<endl;
      }
   }
   cin.get();
}
```

6.9. Definíciók bevitele az utasításokba

A C nyelvben a változók deklarációját (definícióját) a blokkok elején az utasítások előtt kell elhelyeznünk. A C++ 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.

```
// Változók deklarációja az első felhasználás közelében
#include <iostream>
using namespace std;

void main()
{
  cout << "Kerek egy szamot: ";
  int n;
  cin  >> n;
  for (int i=1; i<=n; i++)
    cout<<i <<endl;
}
```

A példában a **for** ciklusba helyeztük a ciklusváltozó definícióját. Bizonyos esetekben a változó-definíciót feltételt használó utasításokba is bevihetjük, amennyiben a változót azonnal inicializáljuk, például véletlen számmal:

```
if (int s=rand()%12) {
    cout<<s<<endl;
}

while (int n=rand()%23) {
    cout<<n<<endl;
}
```

Felhívjuk a figyelmet arra, hogy az utasításokban definiált változók csak az utasításon belül használhatók. Így ha a ciklusváltozó értékére cikluson kívül is szükségünk van, nem szabad ezt a megoldást alkalmaznunk.

7. Származtatott adattípusok

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

7.1. Tömbök, sztringek és mutatók

A tömb (*array*) azonos típusú adatok halmaza, amelyek a memóriában folytonosan helyezkednek el. Az elemek elérése a tömb nevét követő idexelés operátorban megadott elemsorszám (index) segítségével történik. A tömb tehát a változók készlete, amely változókra 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.

7.1.1. Egydimenziós tömbök

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.

Általánosan elmondható, hogy *T* típus esetén a *T[méret]* típus - *méret* darab, *T* típusú elem tárolására alkalmas - egydimenziós tömb (vektor) típusa. Az elemeket *0*-tól (*méret-1*)-ig indexeljük.

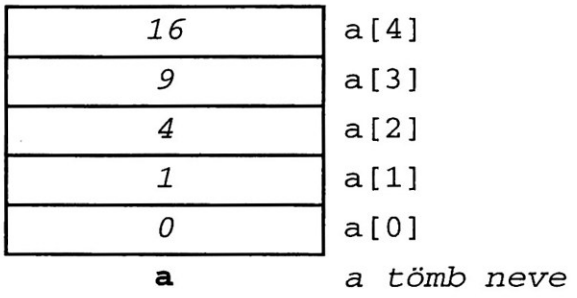
Példaként tekintsünk egy 5 elemet tartalmazó egész tömböt, melynek elemeit az indexek négyzetével töltjük fel! A tömb definíciója:

```
int a[5];
```

A tömb elemeinek egymás után történő elérésére általában a **for** ciklust használjuk, melynek változója a tömb indexe. A tömb elemeire pedig az indexelés operátorának (*[]*) segítségével hivatkozunk.

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

Az alábbi ábrán felvázoltuk a tömb elhelyezkedését a memóriában. Minden elem esetén feltüntettük az elem értékét a ciklus lefutása után, és az elemre való hivatkozás módját.



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

```
int a[10] ;
int n = sizeof(a) / sizeof(a[0]) ;
```

Felhívjuk a figyelmet arra, hogy a C++ nyelv semmilyen ellenőrzést nem tartalmaz a tömb indexeire vonatkozóan. Az indexhatár átlépése a legkülönbözőbb hibákhoz vezethet, melyek felderítése sok időt vesz igénybe.

```
double nap[23];
nap[-1] = 12.23;            // hiba!
nap[23] = 356.23;           // hiba!
```

Az indexhatárok átlépése általában nem közvetlen módon történik, hanem például a kezelő ciklus rossz beállításával. Sajnos ez a hiba csak ritkán vezet jól érzékelhető hibaüzenethez, ehelyett legtöbbször csak a változóink misztikus megváltozásából következtethetünk a hibára.

Az alábbi példában a 4-elemű vektorba beolvasott számokat átlagoljuk, majd kiírjuk az átlagot és az egyes elemek eltérését ettől az átlagtól. Gyakori megoldás, hogy a tömb méretét makró vagy konstans segítségével adjuk meg. Így, ha a méretet később meg kell változtatnunk, akkor csak egyetlen helyen kell a programot módosítanunk.

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

const int NUM = 4 ;

void main()
{
 double szam[NUM];
 double atlag = 0.0;
 int i;

 for (i = 0; i < NUM; i++)          // az elemek beolvasása
 {
     cout<<"szam["<<i<<"] = ";
     cin>>szam[i];
     atlag += szam[i];              // az elemek összegzése
 }
 cin.get();
 atlag /= NUM;                      // az átlag kiszámítása
 cout<<endl<<"Az atlag: "<<atlag<<endl;

 for (i = 0; i < NUM; i++)          // az eltérések kiírása
     cout<<i<<".\t"<<szam[i]<<'\t'<<atlag-szam[i]<<endl;
 cin.get();
}
```

A program futási eredményeinek tanulmányozásakor felhívjuk a figyelmet az adatbevitel megvalósítására:

```
szam[0] = 1.2
szam[1] = 0.9
szam[2] = 2.3
szam[3] = 7.9

Az atlag: 3.075
0.      1.2     1.875
1.      0.9     2.175
2.      2.3     0.775
3.      7.9     -4.825
```

7.1.1.1. Az egydimenziós tömbök inicializálása

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

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

Nézzünk néhány példát vektorok inicializálására!

```
int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
char szo[8] = { 'a', 'l', 'm', 'a'};

float szamok[] = { 12.3, 23.4, 34.5, 45.6 };
```

A második példában az inicializációs lista kevesebb értéket tartalmaz, mint a tömb elemeinek száma. Ekkor a *szo* tömb első 4 eleme felveszi a megadott értékeket, míg a többi elem értéke a tárolási osztálytól függően vagy 0 (**extern**, **static**) vagy pedig határozatlan (**auto**) lesz.

Az utolsó példában a *szamok* tömb elemeinek számát az inicializációs listában megadott konstansok számának (4) megfelelően állítja be a fordítóprogram. Jól használható ez a megoldás, ha a tömb elemeit fordításonként változtatjuk. A kérdés csak az, hogyan tudjuk meg a programon belül a tömb méretét? A választ az elemszámok előzőekben bemutatott meghatározása adja:

```
float szamok[] = { 12.3, 23.4, 34.5, 45.6 };

#define NSZAM (sizeof(szamok) / sizeof(szamok[0]))
```

vagy

```
const int nszam = sizeof(szamok) / sizeof(szamok[0]);
```

A blokkon belül létrehozott (**auto**) tömbök esetén az inicializációs lista tetszőleges futásidejű kifejezést tartalmazhat:

```
double a[3]= {sqrt(2), exp(1), sin(3.14159265/3)};
```

7.1.1.2. Egydimenziós tömbök és a typedef

Mint már említettük a programunk olvashatóságát növeli, ha a bonyolultabb típusneveket szinonim nevekkel helyettesítjük. Erre származtatott típusok esetén is a **typedef** biztosít lehetőséget.

Legyen a feladatunk két 8 elemű egész vektor összegzése egy harmadik vektorban! Az összegzés után az eredményvektort másoljuk át egy negyedik vektorba! A feladat megoldásához szükséges tömböket kétféleképpen is létrehozhatjuk:

```
int a[8], b[8], c[8], d[8];
```

vagy

```
typedef int vektor8[8];
vektor8 a, b, c, d;
```

A C++ nyelv az indexelésen kívül semmilyen más műveletet sem definiál a tömbökre vonatkozóan. Ezért az összegzést és az elemek átmásolását (tömb-tömb közötti értékadás) saját magunknak kell elvégezni.

A példaprogramban az *a* és *b* vektorokat konstansokkal inicializált vektorként definiáltuk:

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

typedef int vektor8[8];

vektor8 a = {1, 2, 3, 4, 5, 0, 1, 3};
vektor8 b = {4, 3, 2, 1, 0, 5, 4, 2};

void main()
{
 vektor8 c, d;
 int i;

 for (i = 0; i < 8; i++)    // az elemek összegzése
     c[i] = a[i] + b[i];

 for (i = 0; i < 8; i++)    // a c vektor másolása d-be
     d[i] = c[i];

 cout<<"\nd = (";           // a d vektor kiírása
 for (i = 0; i < 8; i++)
     cout<<d[i]<<i <<( i<8-1 ? ',' : ')' );
 cout<<endl;
 cin.get();
}
```

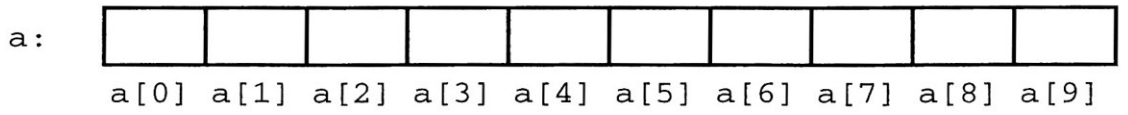
7.1.2. Mutatók és a tömbök

A C++ nyelvben a mutatók és a tömbök között szoros a kapcsolat. Minden művelet, ami tömb-indexeléssel elvégezhető, mutatók segítségével szintén megvalósítható. Az egydimenziós tömbök (vektorok) és az egyszeres indirektségű mutatók között 100%-os (tartalmi és formai) analógia áll fenn. A többdimenziós tömbök és a többszörös indirektségű mutatók között ez a kapcsolat azonban csak formai.

Nézzük meg, honnan származik ez a vektorok és az egyszeres indirektségű mutatók között fennálló szoros kapcsolat! Definiáljunk egy 10 elemű egész vektort!

```
int a[10];
```

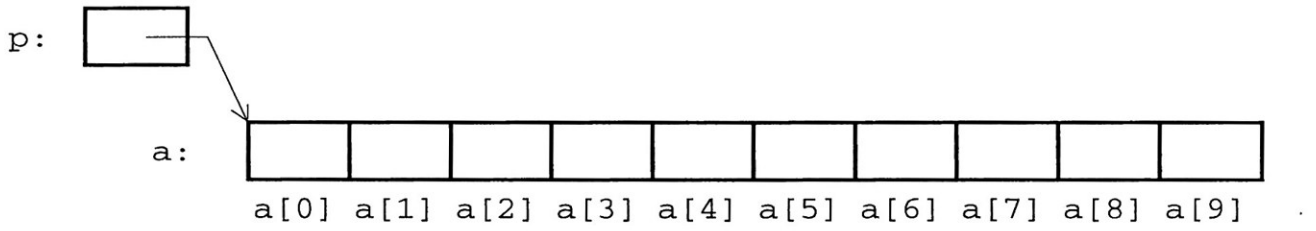
A vektor elemei a memóriában adott címtől kezdve folytonosan helyezkednek el. Mindegyik elemre *a[i]* formában hivatkozhatunk:



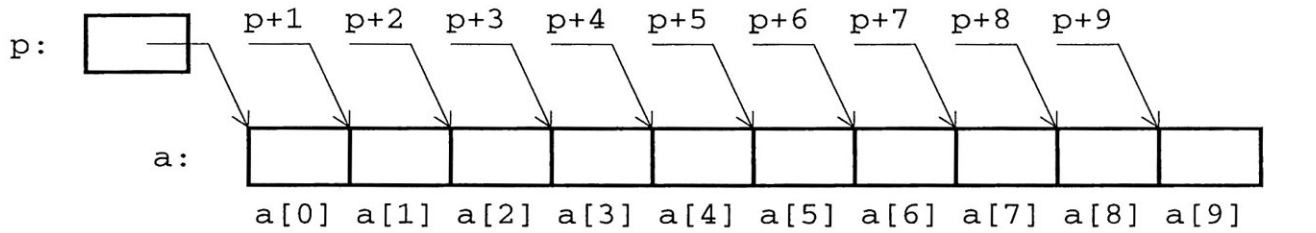
Most vegyünk fel egy *p* egészre mutató pointert, majd a „címe" operátor segítségével állítsuk az *a* tömb elejére (a 0. elemre)!

```
int *p;

p = &a[0];
```

Ezek után, ha hivatkozunk a *p* mutató által kijelölt *(*p)* objektumra (változóra), akkor valójában az *a[0]* elemet érjük el.



Ha *p* memóriaobjektumra (változóra) mutat, akkor a mutatóaritmetika szabályai alapján a *p+1*, a *p+2* stb. címek az adott elem után elhelyezkedő elemeket jelölik ki. (Megjegyezzük, hogy negatív számokkal az objektumot megelőző elemeket címezhetjük meg.) Ennek alapján a **(p+i)* kifejezéssel a tömb minden elemét elérhetjük:



A *p* mutató szerepe teljesen megegyezik az *a* tömbnév szerepével, hisz mindkettő az elemek sorozatának kezdetét jelöli ki a memóriában. Lényeges különbség azonban a két mutató között, hogy míg a *p* mutató változó (tehát értéke tetszőlegesen módosítható), addig az *a* egy konstans mutató, amelyet a fordító rögzít a memóriában.

Ezek után általánosíthatunk tetszőleges típusú *a* tömbre és ugyanolyan típusú *p* mutatóra. Ha a mutatót az alábbi módszerek valamelyikével a tömb első elemére irányítjuk,

```
p = &a[0];          vagy          p = a;
```

akkor azonosak a következő, egy sorban elhelyezkedő hivatkozások:

- A tömb i-dik elemének címe:
  `&a[i]` `&p[i]` `a+i` `p+i`
- A tömb 0-dik eleme:
  `a[0]` `p[0]` `*a` `*p` `*(a+0)` `*(p+0)`
- A tömb i-dik eleme:
  `a[i]` `p[i]` `*(a+i)` `*(p+i)`

A C++ fordító az *a[i]* hivatkozásokat automatikusan **(a+i)* alakúra alakítja, majd ezt a pointeres alakot lefordítja. Az analógia azonban visszafelé is igaz, vagyis az indirektség (*) operátora helyett mindig használhatjuk az indexelés (*[ ]*) operátorát.

7.1.3. Karaktersorozatok (sztringek)

Az egydimenziós tömböket leggyakrabban karaktersorozatok tárolására használjuk. A C++ nyelv nem rendelkezik önálló sztringtípussal, ezért a karaktertömbökben valósítja meg a sztringek tárolását. A sztring tehát olyan karaktertömb (***char[]***), melyben a karaktersorozat végét *nulla* értékű bájt (*0*) jelzi:

'C'	'+'	'+'	' '	'n'	'y'	'e'	'l'	'v'	'\0'

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

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

A programozás során gyakran használunk kezdőértékkel ellátott sztringeket. A kezdőérték megadására használhatjuk a vektoroknál bemutatott megoldásokat, azonban nem szabad megfeledkeznünk a '\0' karakter megadásáról:

```
char st1[10] = { 'N', 'A', 'T', 'A', '\0' };

char st2[] = { 'N', 'A', 'T', 'A', '\0' };
```

Az *st1* sztring számára 10 bájt helyet foglal a fordító, és az első 5 bájtba bemásolja a megadott karaktereket. Az *st2* azonban pontosan annyi bájt hosszú lesz, ahány karaktert megadtunk az inicializációs listában.

A karaktertömbök inicializálása azonban sokkal biztonságosabban elvégezhető a sztringliterálok (sztringkonstansok) felhasználásával:

```
char st1[10] = "NATA";

char st2[] = "NATA";
```

A sztringek kezelésére karaktermutatókat is használhatunk, azonban a mutatókkal óvatosan kell bánni. Tekintsük az alábbi gyakran használt definíciókat! Első esetben a fordító létrehozza a 16-elemű *s* tömböt, majd oda bemásolja az inicializáló szöveg karaktereit, valamint a 0-ás karaktert. A második esetben a fordító az inicializáló szöveget eltárolja a sztringliterálok számára fenntartott területen, majd a sztring kezdőcímével inicializálja a létrejövő *ps* mutatót.

```
char s[16] = "alfa";

char * ps = "gamma";
```

A *ps* értéke a későbbiek folyamán természetesen megváltoztatható (ami a jelen példában a *"gamma"* sztring elvesztését okozza):

```
ps = "iota";
```

Ekkor valójában mutató-értékadás történik, hisz a *ps* felveszi az új sztringkonstans címét. Ezért az s tömb nevére irányuló értékadás fordítási hibához vezet:

```
s = "iota";   // hibás!
```

A C++ nyelv a sztringekre vonatkozóan szintén nem tartalmaz semmilyen műveletet (értékadás, összehasonlítás stb.). Azonban a karaktersorozatok kezelésére szolgáló könyvtári függvények sokkal több lehetőséget biztosítanak a programozónak, mint más nyelvek sztringműveletei. Nézzünk néhány sztringekre vonatkozó alapművelet elvégzésére szolgáló függvényt!

Művelet	***Függvény***
szöveg beolvasása	***scanf(), gets()***
	cin>>, cin.getline()
szöveg kiírása	***printf(), puts()***
	cout<<
Értékadás	***strcpy()***
Hozzáfűzés	***strcat()***
sztring hosszának lekérdezése	***strlen()***
sztringek összehasonlítása	***strcmp()***

(Sztringkezelő függvények használata esetén a *cstring* deklarációs állományt be kell építenünk a forrásprogramba.)

Amennyiben egy sztringen karakterenként kell végiglépkedni, akkor választhatunk a tömbös és a pointeres megközelítés között. A következő példaprogramban a beolvasott karaktersorozatot először titkosítjuk a kizáró vagy művelet felhasználásával, majd

visszaállítjuk az eredeti tartalmát. (A titkosításnál a karaktertömb, míg a visszakódolásnál a mutató értelmezést használjuk.)

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

const unsigned int KULCS=0xE7;

void main()
{
  char s[80], *p;
  cout<<"Kerek egy szoveget    : ";
  cin.getline(s,80);

  for (int i = 0; s[i]; i++)      // titkosítás
      s[i] ^= KULCS;
  cout<<"A titkositott szoveg : "<<s<<endl;

  p=s;
  while (*p)                      // visszaállítás
     *p++ ^= KULCS;
  cout<<"Az eredeti szoveg : "<<s<<endl;
  cin.get();
}
```

Mind a két esetben a ciklusok leállási feltétele a sztringet záró nullás bájt elérése volt.

7.1.3.1. A C++ könyvtár string típusa

A C++ nyelv szabványos sablonkönyvtárában (*STL*) a szövegkezelést támogató osztályokat is találunk. Az előzőekben bemutatott megoldások a C stílusú karaktersorozatokra vonatkoznak, most azonban megismerkedünk a ***string*** típus lehetőségeivel. (A későbbiekben mindkét sztringtípust használjuk a példaprogramokban).

A C++ stílusú karaktersorozat-kezelés eléréséhez a *string* nevű fejállományt kell a programunkba beépíteni:

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

Ezt követően a ***string*** típussal definiált objektumokon keresztül egy sor kényelmes szövegkezelő művelet áll a rendelkezésünkre, operátorok és tagfüggvények formájában. Nézzünk néhányat ezek közül! (A táblázatban a tagfüggvények előtt pont szerepel. A tagfüggvények nevét az objektum neve után, ponttal elválasztva adjuk meg.)

Művelet	*C++ megoldás*
szöveg beolvasása	*cin>>, getline()*
szöveg kiírása	*cout<<*
értékadás	*=, .assign()*
összefűzés	+, +=
sztring hosszának lekérdezése	*.size()*
sztringek összehasonlítása	*.compare()*
átalakítás C stílusú karaktersorozattá	*.c_str(), .data(),.begin()*

Írjuk át szövegtitkosító programunkat C++ stílusú sztringkezelés felhasználásával!

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

const unsigned int KULCS=0xE7;

void main()
{
  string s;
  char *p;
  cout<<"Kerek egy szoveget   : ";
  getline(cin, s);

  for (int i = 0; s[i]; i++)      // titkosítás
      s[i] ^= KULCS;
  cout<<"A titkositott szoveg : "<<s<<endl;

  p=(char *)s.c_str();
  while (*p)                      // visszaállítás
     *p++ ^= KULCS;
  cout<<"Az eredeti szoveg : "<<s<<endl;
  cin.get();
}
```

7.1.4. Többdimenziós tömbök

A C++ nyelv támogatja a többdimenziós tömbök használatát. A többdimenziós tömbök deklarációjának általános formája:

```
típus tömbnév[méret1][méret2]...[méretn];
```

ahol dimenziónként kell megmondani a méreteket. A gyakorlatban a többdimenziós tömbök helyett általában mutatótömböket alkalmazunk.

A fejezetben csak a kétdimenziós tömbök bemutatására és használatára szorítkozunk, azonban a kétdimenziós tömbök (mátrixok) esetén alkalmazott megoldások több dimenzióra is általánosíthatók.

Tároljuk az alábbi 3x5-ös, egész elemeket tartalmazó mátrixot, a C++ program megfelelő adatstruktúrájában!

$$\begin{bmatrix} 4 & 26 & 90 & 19 & 11 \\ 13 & 30 & 70 & 79 & 9 \\ 7 & 87 & 60 & 23 & 12 \end{bmatrix}$$

A definícióban kezdőértékként megadhatjuk a mátrix elemeit, csak arra kell ügyelnünk, hogy sorfolytonos legyen a megadás.

```
matrix[3][5] = { { 4, 26, 90, 19, 11 },
                 {13, 30, 70, 79,  9 },
                 { 7, 87, 60, 23, 12 } };
```

A tömb elemeinek eléréséhez az indexelés operátorát használjuk méghozzá kétszer. A

```
matrix[2][3]
```

hivatkozással a *2.* sor *3.* sorszámú elemét (*23*) jelöljük ki. Nem szabad megfeledkeznünk arról, hogy az indexek értéke minden dimenzióban *0*-val kezdődik!

Az alábbi ábrán a kétdimenziós *matrix* tömb elemei mellett feltüntettük a sorok és az oszlopok (*s/o*) indexeit is.

A sorok címe:		*s/o*	*0*	*1*	*2*	*3*	*4*
`matrix[0]`	→	*0*	4	26	90	19	11
`matrix[1]`	→	*1*	13	30	70	79	9
`matrix[2]`	→	*2*	7	87	60	23	12

A táblázat első oszlopában található kifejezések, amelyeket a második dimenzió elhagyásával kapunk, a tömb sorainak kezdőcímét szolgáltatják. Így érthetővé válik a megállapítás, hogy a kétdimenziós tömb valójában egy olyan vektor (egydimenziós tömb), melynek elemei vektorok (mutatók). Ennek ellenére a többdimenziós tömbök mindig folytonos memóriaterületen helyezkednek el. A példánkban a mátrix sorai képezik azokat a vektorokat, amelyekből a *matrix* vektor felépül.

Használva a vektorok és a mutatók közötti formai analógiát, az indexelés operátorai minden további nélkül átírhatók indirektség operátorává. Az alábbi kifejezések a kétdimenziós tömb ugyanazon elemére hivatkoznak:

```
matrix[1][2]    *(matrix[1] + 2)    *(*(matrix+1)+2)
```

A következő programrészletben először megkeressük a *matrix* legnagyobb (*emax*), illetve legkisebb (*emin*) elemét:

```
int emax, emin;
emax=emin=matrix[0][0];
for (int i = 0; i < 3; i++)
  for (int j = 0; j < 5; j++){
     if (matrix[i][j] > emax )
         emax = matrix[i][j];
     if (matrix[i][j] < emin )
         emin = matrix[i][j];
  }
```

Majd mátrixos alakban megjelenítjük a *matrix* nevű kétdimenziós tömb tartalmát:

```
for (int i = 0; i < 3; i++) {
   for (int j = 0; j < 5; j++)
       cout<<'\t'<<matrix[i][j];
   cout<<endl;
}
```

7.1.5. Mutatótömbök, sztringtömbök

A C++ programok többsége tartalmaz olyan szövegeket, például üzeneteket, amelyeket adott index (hibakód) alapján kívánunk kiválasztani. Az ilyen szövegek tárolására a legegyszerűbb megoldás a sztringtömbök használata.

A sztringtömbök kialakítása során választhatunk a kétdimenziós tömb és a mutatótömb között. Kezdő C++ programozók számára sokszor gondot jelent ezek megkülönböztetése.

Tekintsük az alábbi két definíciót!

```
int a[5][10];

int *b[5];
```

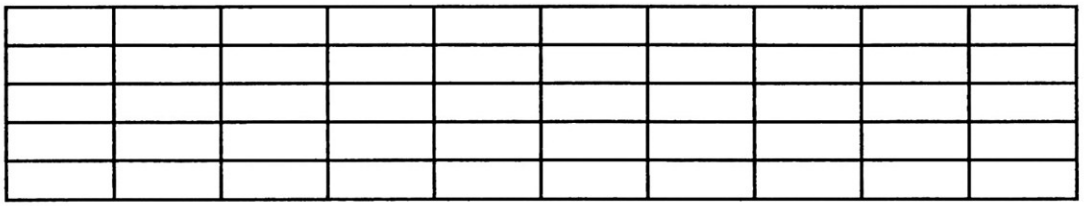
Formailag az *a[2][5]* és a *b[2][5]* hivatkozások egyaránt helyesek, hiszen mindkét esetben egyetlen **int** típusú elemet jelölnek. Azonban az *a* egy „igazi" (statikus) kétdimenziós tömb, amely számára a fordító 50 darab **int** típusú elem tárolására alkalmas folytonos területet foglal le a memóriában. Az elemek helyének meghatározására a fordító az alábbi hagyományos kifejezést használja:

```
10 * sor + oszlop
```

Ezzel szemben a *b* 5-elemű mutatóvektor. A fordító csak az 5 darab mutató számára foglal helyet a definíció hatására. A inicializáció további részeit a programból kell elvégeznünk. Inicializáljuk úgy a mutatótömböt, hogy az alkalmas legyen 5x10 egész elem tárolására!

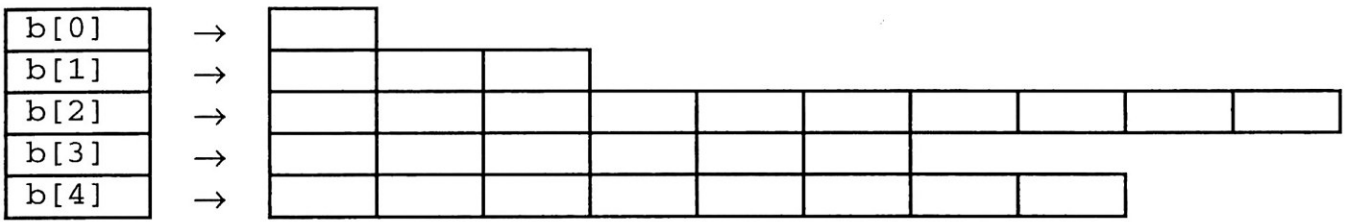
```
int s1[10], s2[10], s3[10], s4[10], s5[10];
int *b[5] = { s1, s2, s3, s4, s5 };
```

Látható, hogy az 50 **int** elem tárolására szükséges memóriaterületen felül, további területet is felhasználtunk (a mutatók számára). Joggal vetődik fel a kérdés, hogy mi az előnye a mutatótömbök használatának. A választ a sorok hosszában kell keresni. Míg a kétdimenziós tömb esetén minden sor ugyanannyi elemet tartalmaz,



addig a mutatótömb esetén az egyes sorok mérete tetszőleges lehet. A alábbi definíciónak megfelelő struktúrát szintén ábrázoltuk.

```
static int s1[1], s2[3], s3[10], s4[6], s5[8];
int *b[5] = { s1, s2, s3, s4, s5 };
```



A mutatótömb másik előnye, hogy a felépítése összhangban van a dinamikus memóriafoglalás lehetőségeivel, így fontos szerepet játszik a dinamikus helyfoglalású tömbök kialakításánál.

A sztringtömböket általában kezdőértékek megadásával definiáljuk. Nézzünk példát sztringtömbök kialakítására kétdimenziós karaktertömb, illetve karakter típusú mutatókat tartalmazó pointertömb felhasználásával!

```
char nyar1[][10] = { "?????",
                     "Június",
                     "Július",
                     "Augusztus" };
```

Az első definíció során egy 4x10-es karaktertömb jön létre, amelyben a sorok számát a fordítóprogram az inicializációs lista alapján határozza meg. A kétdimenziós karaktertömb sorai a memóriában folytonosan helyezkednek el:

nyar1:

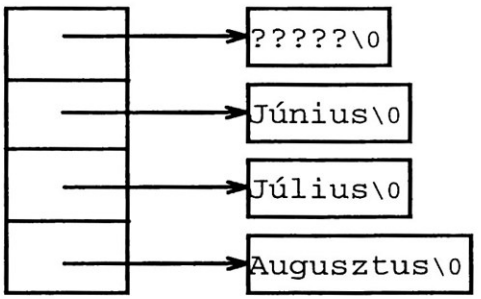
?????\0	Június\0	Július\0	Augusztus\0
0	10	20	30

A második esetben mutatótömböt használunk a nyári hónapok neveinek tárolására. Érdemes összehasonlítani a két megoldást mind a definíció, mind pedig a memóriahasználat szempontjából. A sztringtömb definíciója,

```
char *nyar2[]   = { "?????",
                    "Június",
                    "Július",
                    "Augusztus" };
```

amellyel a memóriában négy különálló területet foglaltunk le, melyeket teljesen feltölt a fordító:

nyar2:



Mindkét esetben az első index megadásával a sztringekre hivatkozunk, míg mindkét index használatával a kiválasztott sztring adott karakterét érjük el.

```
cout<<"Honap : "<<nyar1[3]<<endl;
cout<<"Honap : "<<nyar2[3]<<endl;

cout<<"A honap elso betuje: "<<nyar1[3][0]<<endl;
cout<<"A honap elso betuje: "<<nyar2[3][0]<<endl;
```

A képernyőn az alábbi sorok jelennek meg az utasítások végrehajtása után:

```
Honap : Augusztus
Honap : Augusztus
A honap elso betuje: A
A honap elso betuje: A
```

7.1.6. Dinamikus helyfoglalású tömbök

Tömböket használó programok fejlesztése során nagyon hamar memóriakorlátokba ütközhetünk. Ezért a C++ programban a nagyobb tömbök létrehozását és felszabadítását nem bízzuk a fordítóra, hanem a dinamikus memóriakezelés lehetőségeit kihasználva magunk gondoskodunk e műveletek elvégzéséről. Mint már említettük, a dinamikus memóriahasználat alapgondolata, hogy az adataink számára csak akkor foglalunk helyet, amikor szükségünk van rájuk. Amennyiben az adatok feleslegessé válnak, az általuk elfoglalt memóriaterületet azonnal felszabadítjuk.

C++ nyelven a dinamikus memóriakezeléshez mutatókat használunk. A tömbök és mutatók közötti (tartalmi és formai) analógia tisztázása után nincs akadálya annak, hogy tömbök számára dinamikusan foglaljunk memória-területet. A memória-foglalás elvégzéséhez a **new** operátort, míg a felszabadításához a **delete**[] operátort használjuk (lásd az 5.14. alfejezetet).

A dinamikus tömbök felhasználási lehetőségeinek részletezése helyett három példaprogramot mutatunk, melyek jól szemléltetik az elmondottakat. A feladat nevek beolvasása és névsorba rendezése. Mindhárom megoldásban sztringtömböt használunk a nevek tárolására, amit különböző módon hozunk létre.

Megoldás statikus kétdimenziós karaktertömb használatával

A megoldáshoz használt *nev[20][51]* tömb felépítése:

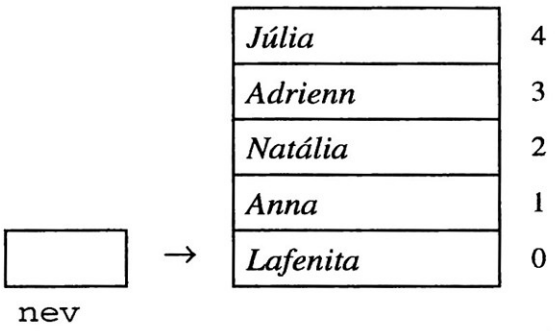
20.	
19.	
...	
4.	*Júlia*
3.	*Adrienn*
2.	*Natália*
1.	*Anna*
0.	*Lafenita*

A 20 sort, soronként 51 karaktert tartalmazó tömböt statikusan hozzuk létre. Emiatt a nevek maximális száma korlátozott, így azt ellenőriznünk kell az adatbevitel során. A megoldásban aláhúzással kiemeltük a lényeges programsorokat.

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

void main() {
  const int maxn=20;
  char nev[maxn][51];
  int db;
```

```
  // A nevek darabszámának bekérése
  cout<<"Hany nevet kivan rendezni: ";
  cin>>db;  cin.get();
  if (db>maxn || db<1) return;              // kilépés

  for (int i=0; i<db; i++) {                // a nevek beolvasása
    cout<<i<<". nev: ";
    cin.getline(nev[i],51);
  }

  char sv[51];                              // rendezés
  for (int i=0; i<db-1; i++)
   for (int j=i+1; j<db; j++)
     if (strcmp(nev[i],nev[j])>0) {
        strcpy(sv,nev[i]);
        strcpy(nev[i],nev[j]);
        strcpy(nev[j],sv);
     }

  for (int i=0; i<db; i++)                  // kiírás
    cout<<nev[i]<<endl;
  cin.get();
}
```

Megoldás egydimenziós 51-karakteres szöveges elemeket tartalmazó dinamikus tömbbel

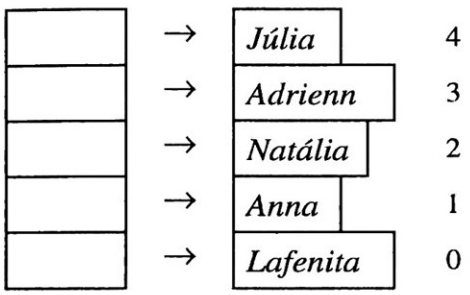


A sorok száma tetszőleges, azonban a nevek legfeljebb 50 karakteresek lehetnek. Valójában egydimenziós dinamikus tömböt használunk, azonban a tömbelemek maguk is egydimenziós karaktertömbök.

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

void main() {
  typedef char tsor[51];
  tsor *nev;
  int db;

  // A nevek darabszámának bekérése
  cout<<"Hany nevet kivan rendezni: ";
  cin>>db;  cin.get();
  if (db<1) return;

  nev=new tsor[db];                  // dinamikus memóriafoglalás
```

```
  for (int i=0; i<db; i++) {        // a nevek beolvasása
  cout<<i<<". nev: ";
    cin.getline(nev[i],51);
  }
  char * sv=new char[51];           // rendezés
  for (int i=0; i<db-1; i++)
   for (int j=i+1; j<db; j++)
     if (strcmp(nev[i],nev[j])>0) {
         strcpy(sv,nev[i]);
         strcpy(nev[i],nev[j]);
         strcpy(nev[j],sv);
     }
  delete[] sv;

  for (int i=0; i<db; i++)          // kiírás
    cout<<nev[i]<<endl;

  // A lefoglalt memória felszabadítása
  delete [] nev;
  cin.get();
}
```

Megoldás dinamikus mutatótömb felhasználásával



A sorok száma és a nevek hossza egyaránt tetszőleges. Felhívjuk a figyelmet a rendezés egyszerűsödésére - a szövegek másolása helyett mutatókat másolunk.

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

void main() {
  typedef char * string;
  string *nev;
  int db;

  // A nevek darabszámának bekérése
  cout<<"Hany nevet kivan rendezni: ";
  cin>>db; cin.get();
  if (db<1) return;

  // Dinamikus memóriafoglalás a mutatótömb számára
  nev=new string[db];
```

```
// A nevek beolvasása (méréthatár az inputpuffer mérete)
  char sv[251];
  for (int i=0; i<db; i++) {
    cout<<i<<". nev: ";
    cin.getline(sv,251);
    // Helyfoglalás a név számára
    nev[i]=new char[strlen(sv)+1];
    // A név másolása
    strcpy(nev[i],sv);
  }

  // Rendezés a mutatók felcserélésével
  for (int i=0; i<db-1; i++)
   for (int j=i+1; j<db; j++)
     if (strcmp(nev[i],nev[j])>0) {
        char * p=nev[i];
        nev[i]=nev[j];
        nev[j]=p;
     }

  // Kiírás
  for (int i=0; i<db; i++)
    cout<<nev[i]<<endl;

  // A nevek számára lefoglalt memóriaterület felszabadítása
  for (int i=0; i<db; i++)
   delete[] nev[i];

  // A mutatótömb felszabadítása
  delete [] nev;
  cin.get();
}
```

Megoldás dinamikus objektumtömb felhasználásával

Végezetül nézzük meg a feladatnak egy igazi C++-os megoldását, ahol az utolsó változat adatszerkezetéhez hasonló struktúrát használunk, azonban alkalmazzuk a szabványos sablonkönyvtár lehetőségeit! A sorok száma és a nevek hossza egyaránt tetszőleges lehet.

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

void main() {
  string *nev;
  int db;
```

```
    // A nevek darabszámának bekérése
    cout<<"Hany nevet kivan rendezni: ";
    cin>>db; cin.get();
    if (db<1) return;

    // Dinamikus memóriafoglalás az objektumtömb számára
    nev=new string[db];

    // A nevek beolvasása
    for (int i=0; i<db; i++) {
      cout<<i<<". név: ";
      getline(cin,nev[i]);
    }

    // Rendezés
    for (int i=0; i<db-1; i++)
     for (int j=i+1; j<db; j++)
       if (nev[i].compare(nev[j])>0)
          swap(nev[i],nev[j]);

    // Kiírás
    for (int i=0; i<db; i++)
      cout<<nev[i]<<endl;

    // Az objektumtömb felszabadítása
    delete [] nev;
    cin.get();
}
```

A programunk még tovább egyszerűsödik a ***sort()*** algoritmus felhasználásával. Az *algorythm* fejállomány beépítése után a rendezést végző programrész egyetlen utasítássá alakítható:

```
    // Rendezés
    sort(&nev[0], &nev[db]);
```

7.2. Felhasználó által definiált adattípusok

A C++ nyelv megengedi, hogy a nyelv meglévő típusait felhasználva újabb típusokat hozzunk létre. Az eddigiek folyamán már többször éltünk ezzel a lehetőséggel, amikor a **typedef** segítségével szinonim típusneveket vezettünk be. Ugyancsak ide tartoznak a felsorolt (**enum**) típusok, amelyeket csoportos, egymással kapcsolatban álló, konstansok létrehozására használunk. Az **enum** típusnév önmagában semmire sem használható, hiszen a felsorolt típust a C++ nyelv felhasználójának (a programozónak) kell definiálnia, az alábbi formában:

```
enum valasz { nem, igen };
```

Ebben a fejezetben a struktúra, az osztály, a bitmező és az unió típusokkal foglalkozunk. A tömb és a felhasználói típusokat közös néven összeállított (*aggregate*) típusoknak nevezzük. A fenti típusok közül az ismerkedést a struktúrával (**struct**) kezdjük. A struktúra típussal kapcsolatos fogalmak és megoldások minden további nélkül alkalmazhatók az osztály, a bitmező és az unió típusra is.

7.2.1. A *struct* struktúratípus

A programozás során 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 (*datamember*). (A más nyelveken megszokott mező (*field*) elnevezést a később ismertetésre kerülő bitstruktúrák esetén használja a C++ nyelv.)

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;      // kezdőérték nélkül!
                    típus2 tag2;
                       . . .
                    típusN tagN;
                    };
```

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

```
struct struktúratípus struktúra_változó;    // C/C++

struktúratípus struktúra_változó;           // C++
```

A C++ nyelvben a **struct**, **union** és **class** kulcsszavak után álló név típusnévként használható a kulcsszó megadása nélkül.

Nézzünk egy konkrét példát a struktúra típus megadására! Könyvtárkezelő program készítése során jól alkalmazható a következő adatstruktúra:

```
struct book {
        char nev [20 ];   // a szerző neve
        char cim [40 ];   // a mű címe
        int ev;           // a kiadás éve
        float ar;         // a könyv ára
};
```

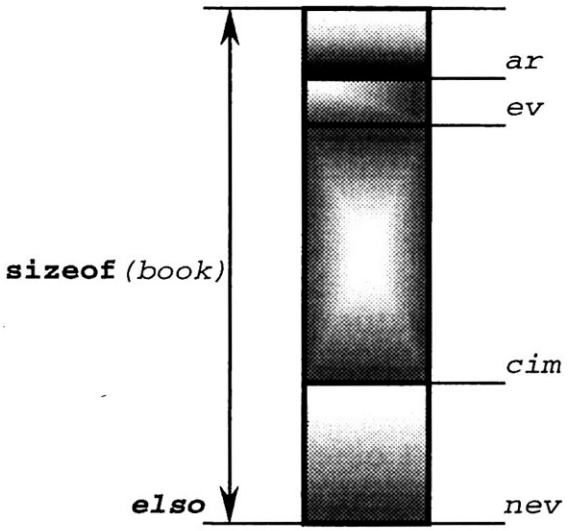
A típusdeklaráció után változókat is létrehozhatunk:

```
book macska, gyerek, cprog;
```

A struktúra definíciójával létrehoztunk egy új felhasználói típust. A struktúra típusú változó adattagjait a fordító a deklaráció sorrendjében tárolja a memóriában. Az alábbi ábrán grafikusan ábrázoltuk a

```
book elso;
```

definícióval létrehozott adatstruktúra felépítését.



Az ábráról is leolvasható, hogy az adattagok nevei a struktúra elejétől mért távolságokat jelölnek. A struktúra mérete általában megegyezik az adattagok méretének összegével. Bizonyos esetekben azonban (optimalizálás sebességre, adatok memóriahatárra való igazítása stb.) „lyukak" keletkezhetnek a struktúra tagjai között. A **sizeof** operátor használatával azonban mindig a pontos méretet kapjuk.

7.2.1.1. Hivatkozás a struktúra adattagjaira

A struktúra szót gyakran önállóan is használjuk, ilyenkor azonban nem a típusra, hanem az adott struktúratípussal létrehozott változóra gondolunk. Definiáljunk néhány változót az előzőekben deklarált *book* típus felhasználásával!

```
book s1, s2, *ps ;
```

Az *s1* és az *s2 book* típusú statikus helyfoglalású változók, amelyek tárolásához szükséges memória-terület lefoglalásáról a fordító gondoskodik. A *ps* egy pointer, amely *book* típusú objektumra mutathat. Ahhoz, hogy a *ps* mutatóval is hivatkozhassunk a struktúrára, két lehetőség közül választhatunk. Az első, kevésbé hasznos esetben a *ps*-t egyszerűen ráirányítjuk az *s1* struktúrára:

```
ps = &s1;
```

A második lehetőség a dinamikus memória-foglalás használatát jelenti. Az alábbi programrészletben memóriát foglalunk a *book* struktúra számára, majd pedig felszabadítjuk azt:

```
ps = new (nothrow) book
if (!ps) exit(-1);
// ...
delete ps;
```

A megfelelő definíciók megadása után három struktúrával rendelkezünk: *s1, s2* és **ps.* Nézzük meg, hogyan lehet értéket adni a struktúráknak! Erre a célra a C++ nyelvben a pont (.) operátor használható:

```
strcpy(s1.nev, "Bjarne Stroustrup");
strcpy(s1.cim, "The C++ Programming Language");
s1.ev = 2002;
s1.ar = 23.12;
```

A pont operátor bal oldali operandusa a struktúraváltozó, a jobb oldali operandusa pedig a struktúrán belül jelöli ki az adattagot.

Amikor a pont operátort a *ps* által mutatott objektumra alkalmazzuk, a precedencia-szabályok miatt zárójelek között kell megadnunk a **ps* kifejezést:

```
strcpy((*ps).nev, "Bjarne Stroustrup");
strcpy((*ps).cim, "The C++ Programming Language");
(*ps).ev = 2002;
(*ps).ar = 23.12;
```

Mivel a C++ nyelvben (főleg a C nyelv első változataiban) gyakran használunk mutató által kijelölt struktúrákat (például függvények argumentumaként), a nyelv ezekben az esetekben egy önálló operátort - a nyíl (->) operátor - biztosít az adattag-hivatkozások elvégzésére. (A nyíl operátor két karakterből, a mínusz és a nagyobb jelből áll.) A nyíl operátor használatával olvashatóbb formában írhatjuk fel *ps* által kijelölt struktúra adattagjaira vonatkozó értékadásokat:

```
strcpy(ps->nev, "Bjarne Stroustrup");
strcpy(ps->cim, "The C++ Programming Language");
ps->ev = 2002;
ps->ar = 23.12;
```

A nyíl operátor baloldali operandusa a struktúraváltozóra mutató pointer, míg a jobboldali operandusa - a pont operátorhoz hasonlóan - a struktúrán belül jelöli ki az adattagot. Ennek megfelelően a *ps->ar* kifejezés jelentése: "*a **ps** mutató által kijelölt struktúra **ar** adattagja*".

A „címe" (&) operátort az *s1* struktúrára alkalmazva szintén mutatóhoz jutunk, amellyel már használható a nyíl operátor:

```
(&s1)->ev=2002;
```

Láthatjuk, hogy a pont és a nyíl operátorok mindkét esetben - közvetlen és a közvetett hivatkozás esetén - egyaránt használhatók. Szem előtt tartva a program olvashatóságát és helyességét javasoljuk, hogy a pont operátort csak közvetlen (a struktúra típusú változó adattagjára történő) hivatkozás esetén, míg a nyíl operátort kizárólag közvetett (mutató által kijelölt struktúra adattagjára vonatkozó) hivatkozás esetén használjuk.

A struktúrára vonatkozó értékadás speciális esete, amikor egy struktúra típusú változó tartalmát egy másik struktúra típusú változónak kívánjuk megfeleltetni. Ezt a műveletet adattagonként is elvégezhetjük,

```
strcpy(s2.nev, s1.nev);
strcpy(s2.cim, s1.cim);
s2.ev = s1.ev;
s2.ar = s2.ar;
```

azonban a C++ szabvány értelmezi a struktúraváltozókra vonatkozó értékadás (=) műveletét is:

```
s2 = s1 ;       // Ez megfelel meg a fenti 4 értékadásnak
*ps = s2 ;
s1 = *ps = s2 ;
```

Az értékadásnak ez a módja valójában egyszerűen a struktúra által lefoglalt memóriablokk átmásolását jelenti. Az értékadás művelete azonban gondot okoz akkor, amikor a struktúra olyan mutatót tartalmaz, amellyel külső memóriablokkra hivatkozunk:

```
struct string {
        char *p;
        int len;
} st1, st2;

st1.p = "Hello";
st2   = st1;
```

A másolás végeztével a külső memóriablokk (sztring) mindkét struktúrához hozzátartozik, hiszen csak a mutatók tartalma (a blokk címe) másolódott át. A fentiekhez hasonló problémák megoldására több lehetőség közül is választhatunk. Egyrészt az adattagonkénti értékadás módszerét használva magunknak kell kiküszöbölni a problémát, másrészt pedig a másolás műveletének átdefiniálásával (*operátor overloading*) saját értékadó operátort definiálhatunk a struktúrához.

Az alábbi példában billentyűzetről töltjük fel a *book* típusú struktúrát, majd megjelenítjük a tárolt adatokat:

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

struct book {
          char nev[20];
          char cim[40];
          int ev;
          float ar;
};

void main() {

   book wb;

   // az adatok beolvasása
   cout<<"\nKerem a konyv adatait!\n";
   cout<<" Szerzo      : ";    cin.getline(wb.nev,20);
   cout<<" Cim         : ";    cin.getline(wb.cim,40);
   cout<<" Kiadas eve  : ";    cin>>wb.ev;
   cout<<" Ar (Ft)     : ";    cin>>wb.ar;
   cin.get();

   // az adatok megjelenítése
   cout<<"\nA konyv adatai :\n";
   cout<<" Szerzo      : "<<wb.nev<<endl;
   cout<<" Cim         : "<<wb.cim<<endl;
   cout<<" Kiadas eve  : "<<wb.ev<<endl;
   cout<<" Ar (Ft)     : "<<wb.ar<<endl;
   cin.get();
}
```

7.2.1.2. Kezdőértékadás a struktúrának

A tömbökhöz hasonlóan a struktúra definíciójában is szerepelhet kezdőértékadás. Az egyes adattagokat inicializáló kifejezések vesszővel elválasztott listáját kapcsos zárójelek közé kell zárni. Példaként lássuk el kezdőértékkel *book* típusú *s1* struktúrát!

```
book s1 = {"Bjarne Stroustrup", "The C++ Programming Language",
           2002, 23.12 };
```

Amennyiben a struktúra valamely adattagja tömb, akkor a kezdőértékek listájában a tömb inicializálását végző részt külön kapcsos zárójelek között adjuk meg. A zárójelek használata nem kötelező, de az inicializálás biztonságosabbá tehető vele:

```
struct Vektor {
          int nelem;
          int v[20];
};

Vektor a = {5, {1, 2, 3, 4, 5} };

Vektor b = {4, 10, 20, 30, 40 };
```

7.2.1.3. Egymásba ágyazott struktúrák

Már említettük, hogy a struktúráknak tetszőleges típusú adattagjai lehetnek. Ha egy struktúrában valamilyen más struktúra típusú adattagot használunk, ún. egymásba ágyazott struktúrát kapunk.

Tételezzük fel, hogy síkbeli geometriai objektumok adatait struktúra felhasználásával kívánjuk feldolgozni! Az alábbi struktúra a geometriai alakzat helyét meghatározó pont koordinátáinak tárolására alkalmas:

```
struct pont {
       int x;
       int y;
};
```

A kört definiáló struktúrában a kör középpontját az előzőleg deklarált *pont* adattagban tároljuk:

```
struct kor {
       pont kp;
       int r;
};
```

Hozzunk létre két kört, méghozzá úgy, hogy az egyiknél használjunk kezdőérték-adást, míg a másikat adattagonkénti értékadással inicializáljuk!

```
kor k1 = { { 100, 100 }, 50 }, k2;

k2.kp.x =  50;
k2.kp.y = 150;
k2.r    = 200;
```

A kezdőérték-adásnál a belső struktúrát inicializáló konstansokat szintén nem kötelező kapcsos zárójelek közé helyezni. A *k2* struktúra adattagjait inicializáló értékadás során az első pont (.) operátorral a *k2*-ben elhelyezkedő *kp* struktúrára hivatkozunk, majd ezt követi a belső struktúra adattagjaira vonatkozó hivatkozás.

Ha a *pont* struktúrát máshol nem használjuk, akkor névtelen struktúraként közvetlenül beépíthetjük a *kor* struktúrába:

```
struct kor {
            struct {
                int x;
                int y;
            } kp;
            int r;
};
```

Bonyolultabb dinamikus adatszerkezetek (például lineáris lista) kialakításánál adott típusú elemeket kell láncba fűznünk. Az ilyen elemek általában valamilyen adatot és egy mutatót tartalmaznak. A C++ nyelv lehetővé teszi, hogy a mutatót az éppen deklarálás alatt álló struktúra típusával definiáljuk. Az ilyen struktúrákat, amelyek önmagukra mutató pointert tartalmaznak adattagként, önhivatkozó struktúráknak nevezzük. Példaként tekintsük az alábbi *listaelem* deklarációt!

```
struct listaelem {
            int adattag;
            listaelem * kapcsolat;
};
```

Ez a rekurzív deklaráció mindössze annyit tesz, hogy a *kapcsolat* mutatóval az adott struktúrára mutathatunk. A fenti megoldás nem ágyazza egymásba a két struktúrát, hiszen az a struktúra, amelyre a későbbiek során a mutatóval hivatkozunk, valahol máshol fog elhelyezkedni a memóriában.

A C++ fordító számára a deklaráció elsősorban azért szükséges, hogy a deklarációnak megfelelően tudjon memóriát foglalni, vagyis hogy ismerje a létrehozandó objektum méretét. A fenti deklarációban a létrehozandó objektum egy mutató, amelynek mérete független a struktúra méretétől.

7.2.1.4. Struktúratömbök

Már láttunk példát arra, hogyan lehet struktúrában tömb adatelemet elhelyezni. Most azonban azt nézzük meg, hogy milyen módon lehet struktúraelemeket tartalmazó tömböket használni! Struktúratömböt pontosan ugyanúgy kell definiálni, mint bármilyen más típusú tömböt. Példaként az előzőekben deklarált *book* típust használva hozzunk létre egy 100-kötetes „könyvtárat"!

```
book lib[100];
```

A kérdés már csak az, hogyan tudunk hivatkozni a tömbelem struktúrák adattagjaira. Ebben az esetben a pont és az indexelés operátorát együtt kell használunk:

```
lib[23].ar = 12.23;
```

Mivel két operátornak azonos a precedenciája, a kiértékelés során a balról-jobbra szabályt használja a fordító. Tehát először a tömbelem kerül kijelölésre (az indexelés), amit az adattagra való hivatkozás (pont operátor) követ. Így zárójelek használata nem szükséges, hiszen a fenti kifejezés az alábbi kifejezéssel egyenértékű:

```
(lib[23]).ar = 12.23;
```

A struktúratömböt a definiálásakor a szokásos módon inicializálhatjuk. A áttekinthetőség érdekében ajánlott az egyes struktúrák kezdőértékét kapcsos zárójelben elkülöníteni:

```
book mlib[] ={ { "0. Iro" , "0. Konyv", 2001, 1000 },
               { "1. Iro" , "1. Konyv", 2002, 2000 },
               { "2. Iro" , "2. Konyv", 2003, 3000 } };
```

Amennyiben dinamikusan kívánjuk a „könyvtárat" létrehozni, mutatót kell használnunk az azonosításra:

```
book * dlib;
```

A struktúraelemek számára a **new** operátorral foglalhatunk helyet a dinamikusan kezelt memória-területen:

```
dlib = new book[100];
```

A tömbelemben tárolt struktúrára a pont operátor segítségével hivatkozhatunk:

```
dlib[23].ar = 12.23;
```

Ha már nincs szükségünk a struktúra elemeire, akkor a **delete**[] operátorral felszabadítjuk a lefoglalt memória-területet:

```
delete[] dlib;
```

Bizonyos könyvtárműveletek (például rendezés) hatékonyabban elvégezhetők, ha mutatótömbben tároljuk a dinamikusan létrehozott könyvek mutatóit:

```
book * plib[100];
```

A struktúraelemek számára az alábbi ciklus segítségével foglalhatunk helyet a dinamikusan kezelt memória-területen:

```
for (int i=0; i<100; i++)
  plib[i]=new book;
```

A tömbelem által kijelölt struktúrára a nyíl operátor segítségével hivatkozhatunk:

```
plib[23] -> ar = 12.23;
```

Ha már nincs szükségünk a struktúra elemeire, akkor az egyes elemeken végighaladva felszabadítjuk a lefoglalt memória-területet:

```
for (int i = 0; i < 100; i++)
   delete plib[i];
```

Az alábbi példában a dinamikusan létrehozott, 100 könyvet tartalmazó könyvtárból, kigyűjtjük az 1979 és 2003 között megjelent műveket. (A programnak az a része, amely a könyvtár véletlen adatokkal való feltöltését végzi, a későbbiek során fájlkezelési műveletekkel helyettesíthető.)

```
#include <iostream>
#include <cstring>
#include <cstdio>
#include <ctime>
#include <cstdlib>

using namespace std;

struct book {
          char nev[20];
          char cim[40];
          int ev;
          float ar;
};

const int KSZAM = 100;

int main()
{
  int i;
  // Helyfoglalás ellenőrzéssel
  bool * talalat = new (nothrow) bool[KSZAM];
  book * dlib = new (nothrow) book[KSZAM];
  if (!dlib) {
          cerr<<"\a\nNincs eleg memoria!\n";
          return -1;
  }
```

```
    // A könyvek véletlenszerű feltöltése
    srand(unsigned(time(NULL)));
    for (int i = 0; i < KSZAM; i++) {
        talalat[i]=false;
        sprintf(dlib[i].nev,"%03d Anonymous", i);
        sprintf(dlib[i].cim,"Nothing %03d", i);
        dlib[i].ev = 1900+rand()%104;
        dlib[i].ar = 500+rand()%6500 * 1.5;
  }
    // A keresett kötetek kiválogatása
    int db = 0;  // Nincs találat
    for (int i = 0; i < KSZAM; i++)
        if (dlib[i].ev >=1979 && dlib[i].ev <= 2003) {
           talalat[i] = true;
           ++db;
        }
  // A keresés eredményének kijelzése
  if (db) {
    cout<<"A talalatok szama: "<<db<<endl;
    for (int i=0; i<KSZAM; i++)
       if (talalat[i])
         cout<<dlib[i].nev<<'\t'<<dlib[i].cim<<'\t'<<
               dlib[i].ev<<'\t'<<dlib[i].ar<<endl;
    }
  else
    cout<<"Nincs a feltetelnek megfelelo konyv!"<<endl;
    // A lefoglalt területek felszabadítása
    delete [] dlib;;
    cin.get();
}
```

7.2.2. A *class* osztálytípus

A C++ nyelvben az objektum-orientált programépítés megvalósításához egyrészt kibővítették a C nyelv **struct** típusát, másrészről pedig egy új **class** típust vezettek be. Mindkét típus alkalmas arra, hogy osztályt definiáljunk segítségükkel. (Az osztályban az adattagok mellett általában tagfüggvényeket is elhelyezünk.) Felmerül a kérdés, hogy miért volt szükséges az új kulcsszó bevezetése? A magyarázat az osztály adattagjainak (és tagfüggvényeinek) elérhetőségében keresendő.

A C nyelvvel való kompatibilitás megtartásának érdekében a struktúra tagjainak korlátozás nélküli (nyilvános, *public*) elérését meg kellett tartani. Az objektum-orientált programozás alapelveinek azonban a zárt struktúra felel meg, melynek tagjait alapértelmezés szerint nem lehet elérni. Annak érdekében, hogy mindkét követelménynek megfeleljen a C++ nyelv, bevezették a **class** kulcsszót. A **class** segítségével olyan struktúrát definiálhatunk, melynek (*private*) tagjai alaphelyzetben nem érhetők el kívülről.

Az előző részben a **struct** típus ismertetésében elmondottak, majdnem teljes egészében megállják a helyüket a **class** típus esetén is. Egyetlen különbség éppen a tagok elérhetőségében rejlik. Az osztálytagok szabályozott elérése érdekében a struktúra-, illetve az osztály-deklarációkban **public** (nyilvános), **private** (privát) és **protected** (védett) kulcsszavakat helyezhetünk el. Az elérhetőség megadása nélkül (alapértelmezés szerint), a **class** típusú osztály tagjai kívülről nem érhetők el (**private**), míg a **struct** típusú osztály tagjai elérhetők (**public**).

Az elmondottak fényében megállapíthatjuk, hogy az alábbi táblázat soraiban megadott típusdefiníciók azonosnak tekinthetők:

```
struct cmplx {
    double re,im;
};
```

```
class cmplx {
  public:
    double re,im;
};
```

```
struct cmplx {
  private:
    double re,im;
};
```

```
class cmplx {
    double re,im;
};
```

Az osztályok használatával kapcsolatosan most csak a kezdőértékadás kérdésével foglalkozunk. Általánosan is elmondhatjuk, hogy (**struct** vagy **class**) osztálytípusú változók definíciójában nem használunk kezdőértékeket. Meg kell jegyeznünk azonban, hogy csupa nyilvános adattaggal rendelkező osztályok esetén a kezdőérték-adás alkalmazható:

```
class cmplx {
  public:
    double re,im;
};
cmplx a={3,4}, b;
```

A **struct** és a **class** felhasználói (absztrakt) adattípusokkal könyvünk II. részében részletesen foglalkozunk, hiszen ezeken alapulnak a C++ nyelv objektum-orientált lehetőségei.

7.2.3. A *union* típusú adatstruktúrák

A C nyelv kidolgozásakor a takarékos memóriahasználat céljából olyan lehetőségeket is beépítettek a nyelvbe, amelyek jóval kisebb jelentőséggel bírnak, mint a dinamikus memória-kezelés. Nézzük meg, miben áll a következő két fejezetben bemutatásra kerülő megoldások lényege!

- Helyet takarítunk meg, ha ugyanazt a memória-területet több változó közösen használja (de nem egyidejűleg). Az ilyen változók összerendelése a C++ struktúra típusával rokon **union** (unió - egyesítés) típussal valósítható meg.

- A másik lehetőség, hogy az olyan változókat, amelyek értéke 1 bájtnál kisebb területen is elfér, egyetlen bájtban helyezzünk el. Ehhez a megoldáshoz a C++ nyelv a bitmezőket biztosítja. Azt, hogy milyen (hány bites) adatok kerüljenek egymás mellé, szintén a **struct** típussal rokon bitstruktúra deklarációval adhatjuk meg.

Az unió és a bitstruktúra megoldásokkal nem lehet jelentős memória-megtakarítást elérni, viszont annál inkább romlik a programunk hordozhatósága. A memóriaigény csökkentését célzó eljárások hordozható változata a dinamikus memóriafoglalás. Fontos megjegyeznünk, hogy az unió és a bitstruktúra tagjainak nyilvános elérése nem korlátozható.

Napjainkban a **union** és bitstruktúra felhasználásának célja valamelyest megváltozott. Az uniót elsősorban gyors és hatékony gépfüggő adatkonverziók megvalósítására, míg a bitstruktúrát a hardver különböző elemeinek vezérlését végző parancsszavak előállítására használjuk.

A **union** típussal igazából nincs sok dolgunk, mivel a **struct** típussal kapcsolatban ismertetett formai megoldások, kezdve a deklarációtól, a pont és nyíl operátoron át egészen a struktúratömb kialakításáig, a **union** típusra is alkalmazhatók. Egyetlen és egyben lényegi különbség az adattagok elhelyezkedése között van. Míg a struktúra adattagjai a memóriában egymás után helyezkednek el, addig az unió adattagjai közös címen kezdődnek (átlapoltak). A **struct** típus méretét az adattagok összmérete (a kiigazításokkal korrigálva) adja, míg a **union** mérete megegyezik a leghosszabb adattagjának méretével.

Az alábbi példában az uniótípus segítségével, az **unsigned long int** típusú adatokat leggyorsabban négy **unsigned char** típusú részre bontjuk:

```
union conv {
   unsigned char ch[4];
   unsigned long szam;
};
conv u, u1={0x78, 0x56, 0x34, 0x12}; // kezdőértékadás
cout<<hex<<u1.szam<<dec<<endl;       // 12345678
for (int i=0; i<4; i++)
  u.ch[i]=48+i;
cout<<hex<<u.szam<<dec<<endl;        // 33323130
```

A következő példánkban a **struct** és a **union** típus együttes használatát mutatjuk be. Sokszor szükség lehet arra, hogy egy állomány rekordjaiban tárolt adatok rekordonként más-más felépítésűek legyenek. Tételezzük fel, hogy minden rekord tartalmaz egy nevet és egy értéket, amely hol szöveg, hol pedig szám! Helytakarékos megoldáshoz jutunk, ha a struktúrán belül unióba egyesítjük a két lehetséges értéket (variáns rekord):

```
struct vrekord  {
     char tipus;
     char nev[25];
     union  {
       char szoveg[30];
       unsigned long szam;
    } ertek;
};

vrekord vr1={'c',"ComputerBooks", "Tartsay Vilmos u. 12"};

vrekord vr2={'d', "ComputerBooks"};
vr2.ertek.szam=3751564;

cout<<vr1.ertek.szoveg<<endl;
cout<<vr2.ertek.szam<<endl;

cout<<"Név         :  "<<vr1.nev<<endl;
switch (vr1.tipus) {
   case 'd' :
         cout<<"Szám        :  "<<vr1.ertek.szam<<endl;
         break;
   case 'c' :
         cout<<"Szöveg      :  "<<vr1.ertek.szoveg<<endl;
         break;
   default  :
         cout<<"Hibas adattipus!"<<endl;
}
```

7.2.3.1. Névtelen unionok használata

A C++ nyelv lehetővé teszi, hogy a struktúrába (osztályba) névtelen uniót ágyazzunk, melynek adattagjai a struktúra (osztály) tagjaivá válnak. A fenti példát az elmondottak szerint módosítottuk:

```
struct vrekord  {
     char tipus;
     char nev[25];
     union  {
       char szoveg[30];
       unsigned long szam;
    };                                  // nincs tagnév!
};

vrekord vr1={'c',"ComputerBooks", "Tartsay Vilmos u. 12"};
vrekord vr2={'d', "ComputerBooks"};
vr2.szam=3751564;

cout<<vr1.szoveg<<endl;
cout<<vr2.szam<<endl;

cout<<"Nev         :  "<<vr1.nev<<endl;
```

```
switch (vr1.tipus) {
   case 'd' :
         cout<<"Szam        :  "<<vr1.szam<<endl;
         break;
   case 'c' :
         cout<<"Szoveg      :  "<<vr1.szoveg<<endl;
         break;
   default  :
         cout<<"Hibas adattipus!"<<endl;
}
```

7.2.4. A bitmezők alkalmazása

A legtöbb programozási nyelvtől eltérően a C++ nyelv beépített módszert tartalmaz a bájton belüli bitek elérésére. Ez a megoldás több szempontból is hasznos lehet:

- Helytakarékos, bitméretű (logikai *KI/BE*) változók használata - több változó tárolása egyetlen bájtban,
- A hardverelemek programozásához használt bitsorozatok magas szintű kezelése.

A már megismert bitenkénti operátorok segítségével szintén elvégezhetők a szükséges műveletek, azonban a bitmezők használatával strukturáltabb kódot kapunk.

A bitmezők kezelése a bitstruktúrán keresztül valósul meg, melynek általános felépítését az alábbiakban láthatjuk:

```
struct struktúratípus {
            típus  név1 : bithossz;
            típus  név2 : bithossz;
                  . . .
            típus  névN : bithossz;
            };
```

A deklarációban a bitmezők neve előtt csak **unsigned int**, **signed int** vagy **int** típus szerepelhet. Ennek megfelelően a *bithossz* maximális értékét az adott számítógépen az **int** típus hossza határozza meg. A struktúratípusban a bitmezők és az adattagok vegyesen is használhatók.

Első példaként a *book* típusunkhoz kapcsoljunk kölcsönzési információkat!

- kölcsönözhető (1 - igen),
- kikölcsönözték (1 - igen),
- maximum hány hétre vihető el (max. 8 hét),
- a kölcsönzés dátuma (**long**).

A fenti követelményeknek megfelelő *libbook* adatstruktúra deklarációja:

```
struct book {
   char nev [20 ];   // a szerző neve
   char cim [40 ];   // a mű címe
   int ev;           // a kiadás éve
   float ar;         // a könyv ára
};

struct libbook {
   book konyv;
   long datum;
   unsigned kolcsonozheto : 1;
   unsigned kikolcsonozve : 1;
   unsigned het           : 4;
};
```

Kezdőértékek beállítása mellett definiáljunk egy változót ezzel a típussal,

```
libbook cppprog = { { "Bjarne Stroustrup",
                      "The C++ Programming Language",
                      2002, 23.12 },
                      20021223L, 1, 0, 8};
```

majd pedig adjunk értéket a bitmezőknek!

```
cppprog.kikolcsonozve = 1;
cppprog.kolcsonozheto = 1;
cppprog.het           = 3;
```

A példából kitűnik, hogy a bitmezők segítségével egyszerűen lehet a bonyolult bitműveleteket elvégezni. A megoldás helytakarékossága szintén látható, hiszen 1 bájtot használtunk 3 **char** típus tárolásához szükséges 3 bájt helyett.

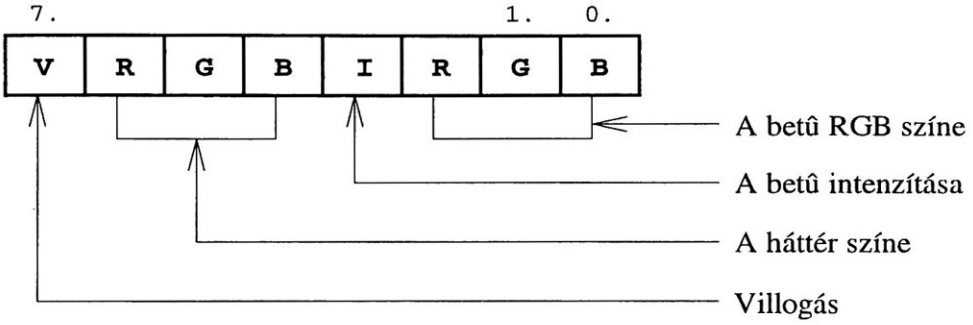
A fenti megállapítás csak akkor igaz, ha beavatkozunk a fordítóprogramok alapértelmezés szerinti memóriahatárra igazításába. Például, a 32-bites fordítók „szeretik" a struktúratagokat (32-bites) duplaszó-határra igazítani a gyorsabb elérés érdekében. Ennek következtében a fenti példában a bitenként összeállított 1 bájt számára 4 bájtot használ el a fordító. A fordítóprogram működésének szabványos vezérlésére a #pragma előfordító direktívát használhatjuk, melynek lehetőségei teljes egészében implementációfüggők. Például a Borland C++ Builder rendszerben a bájthatárra való igazításhoz az alábbi direktívákat kell megadnunk a struktúradefiníció előtt:

```
#pragma nopackwarning
#pragma pack(1)
```

A *pack* paramétereit 1,2,4,8 és 16 értékek közül választhatjuk meg. Az argumentum nélküli esetben az alapértelmezés szerinti 4 értéket használja a fordítóprogram.

Az adatterület megtakarítása mellett a kódterület nagyobb lett, mintha a **char** típust használtuk volna. A futási idő szintén megnövekedett, hiszen a bitműveletek sokkal lassúbbak (nem beszélve a kiigazítás miatti lassabb memória-hozzáférésről), mint egy **char** típusú adattagra való hivatkozás elvégzése.

Végezetül nézzünk egy olyan alkalmazási területet (a hardver programozása), amelynél nem vitathatók a bitmezők használatának előnyei! Tekintsük az IBM PC számítógépek színes karakteres képernyőn való megjelenítéshez használt karakter- és attribútum bájtot. Az attribútum bájt egy bitstruktúra, melynek felépítése az alábbi ábrán látható.



Az alábbi struktúra a karakterkódot tartalmazó és az attribútum bájtot egyaránt lefedi:

```
#pragma pack(1)
struct kepbetu {
        char kod;
        unsigned betu : 3;
        unsigned      : 1;  // Nem használjuk ezt a mezőt
        unsigned alap : 3;
        int           : 1; // A villogást sem használjuk
};
```

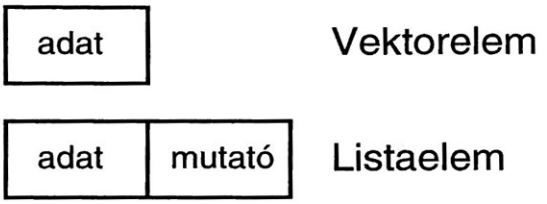
Ha a bitstruktúra deklarációjában nem adunk nevet a bitmezőnek, akkor a megadott bithosszúságú területet nem tudjuk elérni (hézagpótló bitek). Amennyiben a névtelen bitmezők hosszát 0-nak adjuk meg, akkor az ezt követő adattagot (vagy bitmezőt) **int** határra igazítja a fordítóprogram.

Végezetül foglaljuk össze a bitmezők használatának hátrányait!

- A keletkező forráskód nem hordozható, hiszen a különböző rendszerekben a bitek bájt-, szóbeli szervezése eltérő lehet.
- A bitmezők címe nem kérdezhető le (&), hiszen nem biztos, hogy bájthatáron helyezkednek el.
- Mivel a bitmezőkkel egy tárolási egységben több változót is elhelyezhetünk, a fordító kiegészítő kódot generál a változók kezelésére (lassul a program futása és nő a kód mérete.)

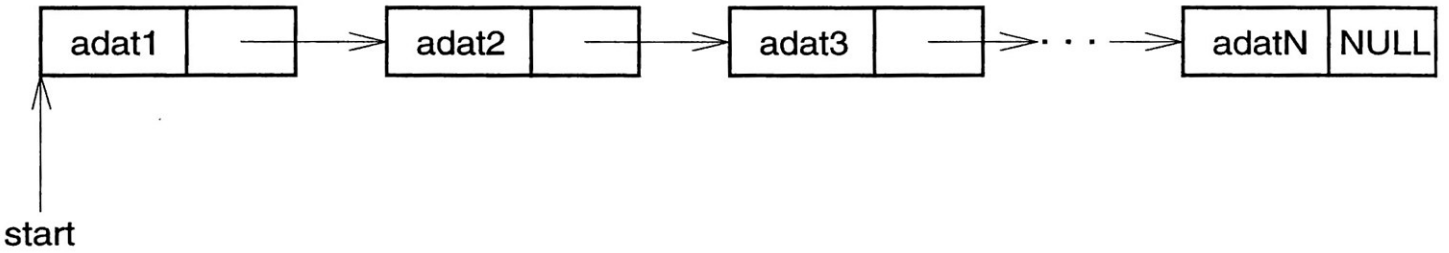
7.2.5. Esettanulmány: önhivatkozó struktúrák használata - listaszerkezet

A programozási munka során kitüntetett szereppel rendelkeznek azok a megoldások, amelyekkel az adatok tárolását valamilyen listaszerkezettel valósítjuk meg, a lista elemeit pedig dinamikus memóriafoglalással hozzuk létre. Nézzük meg, milyen előnyökkel jár a lista használata a vektorral (egydimenziós tömbbel) összehasonlítva. A vektor mérete definiáláskor eldől, a lista mérete azonban dinamikusan növelhető, illetve csökkenthető. Ugyancsak lényeges eltérés van az elemek beszúrása és törlése között. Míg a listában ezek a műveletek csupán néhány mutató másolását jelentik, addig a vektorban nagymennyiségű adat mozgatását igénylik. További lényeges különbség van a tárolási egység, az elem felépítésében:



A vektor elemei csak a tárolandó adatokat tartalmazzák. Lista esetében azonban az adaton kívül a kapcsolati információk tárolására mutató(k)ra is szükség van.

Ezen kis bevezető után ismerkedjünk meg a legegyszerűbb listaszerkezettel, a lineáris listával, melynek elemei egyirányú, egyszeres láncolással (mutatókkal) vannak összekapcsolva:



A lista azonosítására a *start* mutató szolgál, ezért ennek értékét mindig meg kell őriznünk. A lista végét nulla értékű mutatóval (***NULL***) jelezzük. C++ nyelven a listaelemeket a már bemutatott önhivatkozó struktúrákkal hozhatjuk létre. Példaként tegyük önhivatkozóvá a fejezet elején deklarált *book* típust a *book* * típusú *kovetkezo* adattag bevezetésével:

```
struct book {
        char  nev [20];
        char  cim [40];
        int   ev;
        float ar;
        book * kovetkezo;
};
```

A következőkben áttekintjük a listakezelés alapvető műveleteit.

A lista kezelése során szükségünk van segédváltozókra, illetve a lista kezdetét jelölő start mutatóra:

```
book *start=NULL, *elozo, *aktualis, *kovetkezo;
```

Amikor a lista adott elemével (*aktualis*) dolgozunk, szükségünk lehet az megelőző (*elozo*) és a rákövetkező (*kovetkezo*) elemek helyének ismeretére is.

A példában 10 elemet tartalmazó listát építünk fel. A lista felépítése során minden egyes elem esetén három jól elkülöníthető tevékenységet kell elvégeznünk:

1. helyfoglalás (ellenőrzéssel) a listaelem számára,
2. a listaelemben tárolt adatok feltöltése,
3. a listaelem hozzáfűzése a listához (a végéhez). A hozzáfűzés során az első és nem első elemek esetén más-más lépéseket kell végrehajtanunk.

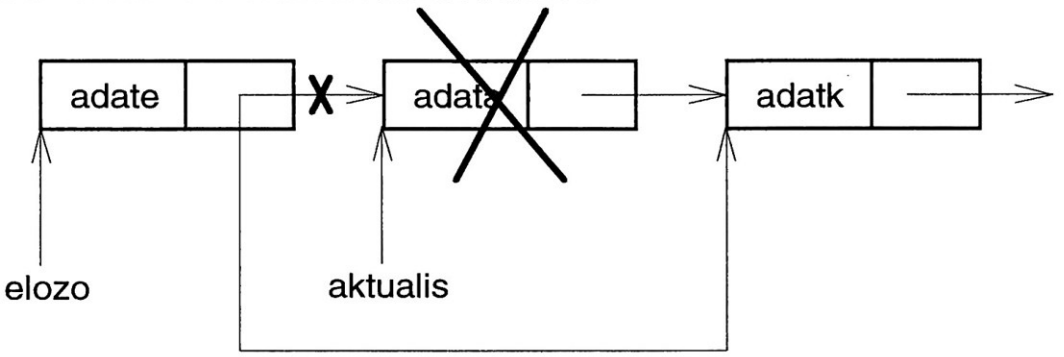
```
// A lista felépítése és az elemek véletlen feltöltése
srand(unsigned(time(NULL)));
for (int index = 0;index < 10;index++) {
   // ❶ Memória-foglalás ellenőrzéssel
   aktualis = new (nothrow) book;
   if (!aktualis) {
      cerr<<"\a\nNincs eleg memoria!\n";
      return -1; }
   // ❷ A listaelem adattagjainak feltöltése
   sprintf(aktualis->nev,"%03d Anonymous", index);
   sprintf(aktualis->cim,"Nothing %03d", index);
   aktualis->ev = 1900+rand()%104;
   aktualis->ar = 500+rand()%6500 * 1.5;
   aktualis->kovetkezo = NULL;
   // ❸ Láncolási és léptetési műveletek
   if (index == 0)                        // első elem
      start = elozo = aktualis;
   else {                                 // további elemek
      // az új elem láncolása az előző elemhez
      elozo->kovetkezo = aktualis;
      elozo = aktualis;                   // továbblépés a listában
   }
}
```

A ciklus lefutása után kész van a 10 elemet tartalmazó listánk. Most következhetnek az ún. listakezelési műveletek. Az első ilyen művelet legyen a lista végigjárása, melynek során az egyes elemek tartalmát megjelenítjük a képernyőn. A bejárás során a *start* mutatótól indulunk, és a ciklusban mindaddig lépkedünk a következő elemre, amíg el nem érjük a lista végét jelző nulla (***NULL***) mutatót:

```
// A listában lépkedve az elemek adatainak kiírása
aktualis = start;
cout<<endl;
do {
   cout<<aktualis->nev<<'\t'<<aktualis->cim<<'\t'<<
         aktualis->ev<<'\t'<<aktualis->ar<<endl;
   // Lépés a következő elemre
   aktualis = aktualis->kovetkezo;
} while (aktualis != NULL);
```

Gyakran használt művelet a ***listaelem törlése***. A példában a törlendő elemet a listaelem sorszáma alapján azonosítjuk (A sorszámozás 0-val kezdődik a *start* által kijelölt elemtől kezdődően - a programrészlet nem alkalmas a 0. és az utolsó elem törlésére!) A törlés művelete szintén három tevékenységre tagolható:

1. az adott sorszámú elem lokalizálása a listában,
2. a törlés elvégzése,
3. a törölt elem területének felszabadítása.

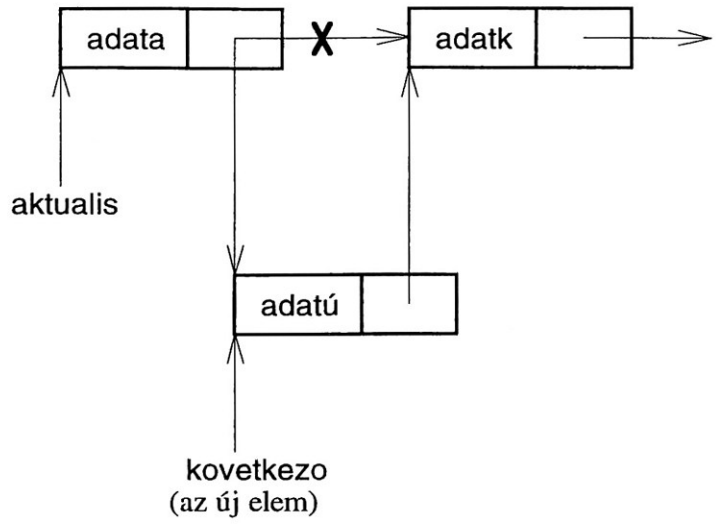


A példában a 4. sorszámú elemet töröljük a listából:

```
// A 4. sorszámú elem lokalizálása és törlése
// ❶ Az elem helyének meghatározása
aktualis = start;
for (int index = 0; index<4; index++) {
    elozo    = aktualis;
    aktualis = aktualis->kovetkezo;
}
// ❷ A törlés - kifűzés a láncból
elozo->kovetkezo = aktualis->kovetkezo;
// ❸ A terület felszabadítása
delete aktualis;
```

A törléssel ellentétes művelet - ***új elem beillesztése*** a listába, két meglévő elem közé. A beszúrás helyét annak az elemnek a sorszámával azonosítjuk, amely mögé az új elemet beillesztjük. A beszúrás műveletét a következő négy lépésben végezzük:

1. a beszúrás helyének lokalizálása a megelőző elem sorszáma alapján,
2. helyfoglalás az új listaelem számára,
3. a listaelem feltöltése adatokkal,
4. az elem beillesztése a sorszámmal kijelölt elem után.



A példában a 3. sorszámú elem mögé illesztünk új listaelemet:

```
// A 3. sorszámú elem lokalizálása és mögé új elem beszúrása
// ❶ A megelőző elem helyének meghatározása
aktualis = start;
for (int index = 0; index<3; index++)
    aktualis = aktualis->kovetkezo;
// ❷ Területfoglalás az új elem számára
kovetkezo = new (nothrow) book;
if (!kovetkezo) {
  cerr<<"\a\nNincs eleg memoria!\n";
  return -1;
}
// ❸ Az új elem adatainak feltöltése
strcpy(kovetkezo->nev,"!!! Anonymous");
strcpy(kovetkezo->cim,"Nothing !!!");
kovetkezo->ev = 1989;
kovetkezo->ar = 1223;
// ❹ Az új elem befűzés a láncba
kovetkezo->kovetkezo = aktualis->kovetkezo;
aktualis ->kovetkezo = kovetkezo;
```

A listakezelő programok a fenti lépéseket használják, esetleg kibővítve az itt bemutatott lehetőségeket. Helyes programozási gyakorlat az, hogy kilépés előtt felszabadítjuk a dinamikusan foglalt memóriaterületeket. Nézzük meg, hogyan történik a lista elemeinek megszüntetése. Ebből a célból szintén végig kell mennünk a listán, ügyelve arra, hogy még az aktuális listaelem megszüntetése előtt kiolvassuk a következő elem helyét:

```
aktualis = start;
do {
    kovetkezo = aktualis->kovetkezo;
    delete aktualis;
    aktualis = kovetkezo;
} while (kovetkezo != NULL);
start = NULL;  // Nincs lista!
```

8. 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 könyvünkben az ANSI szabvány által javasolt *paraméterek* és *argumentumok* elnevezést használjuk.

A függvényhívás során a vezérlés a hívó függvénytől átkerül az aktivizált függvényhez. Az argumentumok (amennyiben vannak) szintén átadódnak a hívott 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.

8.1. Függvények definíciója és deklarációja

A saját készítésű 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 függvény-definíció általános formáját! A függvény fejsorában elhelyezkedő *„paraméter-deklarációs lista"* az egyes paramétereket vesszővel elválasztva tartalmazza. Minden egyes paraméter előtt szerepel annak típusa.

```
⟨visszatérési típus⟩ függvénynév (⟨paraméter-deklarációs lista⟩)
{
   // a függvény törzse
   ⟨lokális definíciók és deklarációk⟩
   ⟨utasítások⟩
}
```

A ⟨ ⟩ jelek között megadott részek hiányozhatnak a definícióból. Példaként készítsük el a nem negatív egész számok egész kitevőre történő hatványozását végző függvényt!

```
int uihatvany( int alap, int exp )
{
   int hv = 1;
   if (exp >0)
      for ( ; exp; exp--)
          hv*=alap;
   return hv;
}
```

A függvények definíciójában a visszatérési típus előtt megadhatjuk a tárolási osztályt is. Függvények esetén az alapértelmezés szerinti tárolási osztály az **extern**, amely azt jelöli, hogy a függvény más modulból is elérhető. Amennyiben a függvény elérhetőségét az adott modulra kívánjuk korlátozni, a **static** tárolási osztályt kell használnunk (a paraméterek deklarációjában csak a **register** tárolási osztály specifikálható). Ha a függvényt saját névterületen szeretnénk elhelyezni, úgy a függvény definícióját, illetve a prototípusát a kiválasztott névterület (*namespace*) blokkjába kell vinnünk. (A tárolási osztályok és a névterületek részletes ismertetését a következő fejezet tartalmazza.)

Mint említettük a függvény prototípusa, amely általában megelőzi a függvény definícióját, meghatározza a függvény nevét, visszatérési típusát (továbbá - amennyiben megadjuk - a tárolási osztályt és a függvény attribútumait), valamint információt tartalmaz a paraméterekről:

```
⟨visszatérési típus⟩ függvénynév (⟨paraméter-deklarációs lista⟩);
```

A függvény prototípusát mindig pontosvesszővel kell lezárni. Nézzük meg, milyen mechanizmusok érvényesülnek a program fordításakor a prototípust használata során!

- A prototípus definiálja a függvény visszatérési típusát, amennyiben az eltér az **int** típustól.
- Az argumentumok konverziója a prototípusban definiált típusoknak megfelelően, nem pedig az automatikus konverziók szerint megy végbe.
- A paraméterlista és az argumentumlista összevetésével a fordító ellenőrzi a paraméterek számának és típusainak összeférhetőségét.
- A prototípus függvénymutató inicializálására is felhasználható.

A prototípus gyakorlatilag megegyezik a függvénydefiníció első sorával (a függvényfejjel), amelyet pontosvesszővel zárunk. A prototípus általában csak a paraméterek típusát tartalmazza, amennyiben a paraméterek nevét is megadjuk, akkor azokat figyelmen kívül hagyja a fordító:

```
⟨visszatérési típus⟩ függvénynév (⟨típuslista⟩);
```

Az elmondottak alapján az alábbi két prototípus megegyezik:

```
int uihatvany( int, int);

int uihatvany( int alap, int exp );
```

Felhívjuk a figyelmet arra, hogy a paraméterrel nem rendelkező függvények prototípusát eltérő módon értelmezi a C és a C++ nyelv:

Deklaráció	*C értelmezés*	*C++ értelmezés*
`típus f();`	`típus f(...);`	`Típus f(void);`
`típus f(...);`	`típus f(...);`	`típus f(...);`
`típus f(void);`	`típus f(void);`	`típus f(void);`

A C++ nyelv lehetővé teszi, hogy a legalább egy paramétert tartalmazó paraméterlistát a **,...** deklaráció zárja. Az így definiált függvény legalább egy, de különben tetszőleges számú és típusú argumentummal meghívható. Példaként tekintsük a ***printf()*** függvény prototípusát!

```
int printf( const char * formatum, ...);
```

8.2. A függvények paraméterezése és a függvényérték

A C++ függvény-definícióban szereplő paraméterlistában minden paraméter előtt ott áll a paraméter típusa. A paraméterek deklarációs sorrendje követi a paraméterek sorrendjét, és semmilyen összevonás sem lehetséges.

```
int uihatvany( int alap, int exp ) {
// ...
}
```

A deklarált paramétereket a függvényen belül a függvény lokális változóiként használhatjuk, azonban a függvényen kívülről nem érhetők el. A paraméterek típusa a skalár (**bool**, **char**, **wchar_t**, **short**, **int**, **long**, **float**, **double**, felsorolási, referencia és mutató), a struktúra-, az unió- és a tömbtípusok közül kerülhet ki.

A visszatérési típus meghatározza a függvényérték típusát, amely tetszőleges skalár vagy strukturált (**struct**, **class**, **union**) típus lehet. (Nem lehet azonban tömbtípus.)

A függvény a **return** utasítás feldolgozásakor ad vissza értéket, amelyet (ha szükséges) a visszatérési típusra konvertál. A visszaadott érték az utasításban szereplő kifejezés értéke:

```
return kifejezés;
```

Ha a függvény definíciójában nem adjuk meg a visszatérési típust, akkor alapértelmezés szerint **int** típusú lesz a függvényérték.

A függvényen belül tetszőleges számú **return** utasítás elhelyezhető. Az alábbi faktoriálist számító függvényekben egy, illetve két **return** utasítást használunk. (A függvények önmagukat hívó, rekurzív függvények.)

```
int fact1(int n)
{
  return (n>1) ? n*fact1(n-1) : 1;
}

int fact2(int n)
{
   if (n>1)
      return n * fact2(n-1);
   else
      return 1;
}
```

A **void** típus felhasználásával olyan függvényeket is készíthetünk, amelyek nem adnak vissza értéket. (Más programozási nyelveken ezeket az alprogramokat eljárásoknak nevezzük.) Ebben az esetben a függvényből való visszatérésre a **return** utasítás

kifejezés nélküli alakját használjuk. A **void** függvényekben gyakran a függvény törzsét záró kapcsos zárójelet használjuk visszatérésre. Az alábbi *sorminta* függvény a megadott karaktert adott számszor kiírja egymás mellé:

```
void sorminta(int db, char ch)
{
  for (register int i=0; i<db; i++)
   cout<<ch;
}
```

A különböző C++ változatokban a visszatérési típus után különböző függvényattribútumok is szerepelhetnek, amelyek a függvény alapértelmezés szerinti működését módosítják. Például a *Borland C++ Builder* rendszerben az alábbi attribútumokat (módosítókat) használhatjuk:

__pascal - Pascal függvényhívási konvenciók,
__cdecl - C függvényhívási konvenciók,
__fastcall - Delphi függvényhívási konvenciók,
__stdcall - Windows által alkalmazott függvényhívási konvenciók.

8.3. A függvényhívás

A függvényhívás olyan kifejezés, amely átadja a vezérlést és az argumentumokat (ha vannak) az aktivizált függvénynek. A függvényhívás általános alakja:

kifejezés1(⟨*kifejezés2*⟩)

ahol a *kifejezés1* a függvény neve (vagy a függvény címét szolgáltató kifejezés), míg az opcionálisan megadható *kifejezés2* az argumentum-kifejezések vesszővel tagolt listája.

```
int fv( int a, float b );   // prototípus

x = fv( 4, 5.67 );          // függvényhívás
```

Az argumentumok kiértékelésének sorrendjét nem definiálja a C++ nyelv. Egyetlen dolgot garantál mindössze a függvényhívás operátora, hogy mire a vezérlés átadódik a hívott függvénynek, az argumentumlista teljes kiértékelése (a mellékhatásokkal együtt) végbemegy. Azon függvények esetén, ahol a prototípusban a paraméterlista helyén **void** kulcsszó szerepel, a híváskor nem adható meg egyetlen argumentum sem:

```
int fv(void);               // prototípus

x = fv();                   // függvényhívás
```

Mint láttuk a függvényhívásnál használt argumentumok skalár, struktúra, osztály vagy unió típusúak lehetnek. A C++ nyelvben az argumentumok érték szerint adódnak át a hívott függvénynek. Ez azt jelenti, hogy az argumentum másolatát veszi fel a megfelelő paraméter értékként. Ennek következtében, ha függvényen belül a paraméteren valamilyen műveletet végzünk, annak nincs kihatása a híváskor megadott argumentumra.

Ezek után felmerül a kérdés, hogyan lehet C++-ban olyan függvényt készíteni, amely felcseréli két egész típusú változó értékét? Az érték szerint argumentum-átadás látszólag ezt nem teszi lehetővé. Ha azonban az átadott érték valamely változónak a címe vagy referenciája, akkor ezek felhasználásával lehetőség nyílik arra, hogy a függvényből „kihivatkozva" megváltoztassuk a változó értékét. Nézzük példaként az egész változók értékének felcserélését megvalósító programot!

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

void csere1(int *, int *);     // prototípusok
void csere2(int &, int &);

void main() {
  int x=7, y=30;
  csere1( &x, &y );           // függvényhívások
  // x=30, y=7
  csere2( x, y );
  // x=7, y=30
}

// A csere1 függvény definíciója
void csere1 ( int * p, int *q ) {
   int sv = *p;
   *p = *q;
   *q = sv;
}

// A csere2 függvény definíciója
void csere2 ( int & a, int & b) {
   int sv = a;
   a = b;
   b = sv;
}
```

A *csere1()* függvény argumentumai nem a változók értékei, hanem a változók címei (*&x* és *&y*), és ezek adódnak át, amiket egészre mutató pointerekbe, mint paraméterekbe, veszünk át (***int** *p* és **int** *q*). A cserét ezek után a **p* és a **q* tárolók között végezzük el, egy *sv* segédváltozó bevezetésével.

A referencia típus igazi lehetőségét a hivatkozás szerinti paraméter-átadás jelenti. A *csere2()* függvény referencia típusú paraméterei (***int &a***, ***int &b***) a függvényen belül a híváskor átadott argumentumokra (*x*, *y*) hivatkoznak. A függvény blokkjában az *a* paraméter az *x* változó, a *b* paraméter pedig az *y* változó második neveként jelenik meg. (A háttérben a fordító valójában címeket másol, azonban a forrásprogramból ez nem látható.)

Nézzünk néhány érdekes példát mutatók és a referenciák paramétersorban való használatára! Az elsőben egy **void** típusú függvény segítségével megvalósítjuk az egész típusú operandussal rendelkező „címe" operátort (&). A megoldásban az okoz nehézséget, hogy mutatóra mutató pointert (**) kell ahhoz átadnunk a *cime* függvénynek, hogy az a függvényen belül értéket kapjon. Másrészt az egész változó címének átadásáról (*) is gondoskodnunk kell.

```
void cime(int ** pp, int * p) {
  *pp = p;
}

void main()
{
  int a = 79120;
  int *ap;
  cime( &ap , &a );   // Mint az ap = &a
  *ap += 23;          // Az a értéke 79143 lesz!
}
```

A feladat jobb megoldásához jutunk, ha a mutatót (***int** *) és az egész változót egyaránt referenciával adjuk át a függvénynek:

```
void cimer(int * & pp, int & p) {
  pp = &p;
}

void main()
{
  int a =79120;
  int *ap;
  cimer( ap , a );    // Mint az ap = &a
  *ap += 23;          // Az a értéke 79143!
}
```

Felhívjuk a figyelmet arra, hogy referenciához nem készíthetünk se mutatót (***int** & **), se referenciát (***int** & &).

A következőben olyan függvényt mutatunk be, amely mutató típusú visszatérési értékkel rendelkezik. A függvény az argumentumként átadott címet egyszerűen visszaadja függvényértékként:

```
int * kozvetit1( int * p)
{
   return p;
}

void main()
{
   int  a=79120;
   *kozvetit1(&a) += 23;  // mint az a += 26;
   // az a értéke 79143 lesz!
}
```

Az előző függvény helyett sokkal biztonságosabb az alábbi használata, amely referenciát fogad a paraméterében, és ezt függvényértékként vissza is adja:

```
int & kozvetit2( int & r)
{
   return r;
}

void main()
{
   int  a=79120;
   kozvetit2(a) += 23;  // mint az a += 26;
  // az a értéke 79143 lesz!
}
```

Figyeljünk arra, hogy függvényen belül definiált lokális tárolók (*auto*) címének, illetve referenciájának kiadása a függvényből súlyos következményekkel járhat! Ennek oka, hogy a függvényből kilépve ezek a változók megszűnnek létezni, így címükkel (referenciájukkal) érvénytelen memóriaterületre hivatkozunk.

8.4. Különböző típusú paraméterek használata

A függvények készítése során a legkülönbözőbb típusú adatok átadására lehet szükség. Lényeges, hogy mindig az igényeknek megfelelően válasszuk meg a függvényeink paramétereit. Ebben segítségünkre lehetnek az alábbiak, ahol áttekintjük a különböző típusú paraméterek használatának szabályait.

8.4.1. Aritmetikai típusú paraméterek

Az alábbiakban elmondottak **bool**, **char**, **wchar_t**, **int**, **enum**, **float** és **double** típusok, illetve ezek típusmódosítókkal (**signed**, **unsigned**, **short**, **long**) ellátott változataikra egyaránt érvényesek. A felsorolt típusokkal minden további nélkül készíthetünk paramétereket, illetve a függvényértékét is definiálhatjuk.

Egyszerűség kedvéért tekintsük a két **double** szám összegzését végző függvény különböző változatait! Mivel a függvényünk egyetlen értéket szolgáltat eredményül, ezt megteheti függvényértékként (*osszeg1()*), illetve egy referencia (vagy pointer) paraméteren keresztül (*osszeg2()*).

```
double osszeg1(double a, double b)
{
   return a+b;
}

void osszeg2(double a, double b, double &c)
{
   c=a+b;
}
```

Természetesen eltérő módon kell hívnunk a két függvényt:

```
cout<<osszeg1(10,20.5)<<endl;
double d;
osszeg2(10,20.5,d);
cout<<d<<endl;
```

8.4.2. Felhasználói típusú paraméterek

A C++ nyelv felhasználói típusainak (**struct**, **class**, **union**) paraméterlistában, illetve függvényértékként való felhasználására az aritmetikai típusoknál ismertetett szabályok érvényesek. Ennek alapja, hogy a nyelv definiálja az azonos típusú osztályok, illetve uniók közötti értékadást. Az ok, amiért mégis részletesen foglalkozunk ezzel a kérdéssel, az objektum-orientált programépítés eszköztárában keresendő.

A felhasználói típusú argumentumok függvénynek való átadása során az érték szerinti, a referenciával, illetve a mutató segítségével megvalósított megoldások között választhatunk. Az szabványos C++ nyelvben a függvény visszatérési értéke felhasználói típusú is lehet. A lehetőségek közül általában ki kell választanunk az adott feladathoz legjobban illeszkedő megoldást. A választásnál figyelembe vehetjük, hogy a memóriaigény, illetve a futási idő szempontjából melyik megoldás a leghatékonyabb.

Példaként vegyük a komplex számok tárolására alkalmas struktúrát!

```
struct complex {
       double re, im;
       };
```

Készítsünk függvényt két komplex szám összeadására (*csum1()*), amelyben a tagok és az eredmény tárolására szolgáló struktúrát mutatójuk segítségével adjuk át a függvénynek! Mivel a bemenő paramétereket nem kívánjuk a függvényen belül megváltoztatni, **const** típuselőírást használunk.

```
void csum1(const complex *pa, const complex *pb, complex *pc)
{
  pc->re = pa->re + pb->re;
  pc->im = pa->im + pb->im;
}
```

A második függvény (*csum2()*) visszatérési értékként szolgáltatja a két érték szerint átadott komplex szám összegét. Az összegzést egy lokális struktúrában végezzük el, melynek értékét a **return** utasítással adjuk vissza.

```
complex csum2(complex a, complex b)
{
  complex c;
  c.re = a.re + b.re;
  c.im = a.im + b.im;
  return c;
}
```

A második megoldás természetesen sokkal biztonságosabb, és sokkal jobban kifejezi a művelet lényegét, mint az első. Minden más szempontból (memóriaigény, sebesség) az első függvényt kell választanunk. A referenciatípus használatával azonban olyan megoldáshoz juthatunk, amely magában hordozza a *csum2()* függvény előnyös tulajdonságait, azonban az *csum1()* függvénnyel is felveszi a versenyt.

```
complex csum3(const complex & a, const complex & b)
{
  complex c;
  c.re = a.re + b.re;
  c.im = a.im + b.im;
  return c;
}
```

A konstans referenciák az egyirányú paraméterátadásra utalnak, melynek során az argumentum-struktúra tartalma nem másolódik, csak hivatkozunk rá.

A három különböző megoldáshoz két különböző hívási mód tartozik. Az alábbi programrészlet mind a három összegző függvény hívását tartalmazza:

```
void main()
{
  complex c1 = {4, 7}, c2 = {26, 13}, c3;

  csum1(&c1, &c2, &c3); // mindhárom argumentum pointer

  c3 = csum2(c1, c2);   // két struktúra argumentum

  c3 = csum3(c1, c2);   // két struktúra-referencia argumentum
}
```

8.4.3. Tömbök átadása függvénynek

A következőkben megnézzük, hogy milyen lehetőségeket biztosít a C++ nyelv tömbök függvénynek való átadására. Már a legelején le kell szögeznünk, hogy tömböt ***nem lehet érték szerint*** (a teljes tömb átmásolásával) függvénynek átadni, illetve függvényértékként megkapni. Sőt különbség van az egydimenziós (vektorok) és a többdimenziós tömbök argumentumként való átadása között.

8.4.3.1. Vektorargumentumok

Egydimenziós tömbök (vektorok) függvényargumentumként való megadása esetén a tömb első elemére mutató pointer adódik át. Más szavakkal a *T[]* típusú vektorargumentum *T** típusú mutatóként adódik át a hívott függvénynek. Ebből a megoldásból viszont az is következik, hogy a vektor elemein, a függvényen belül végrehajtott változtatások a függvényből való visszatérés után is érvényben maradnak.

A vektorok átadásának fenti módszere elegendő ahhoz, hogy a vektor elemeit elérjük (gondoljunk a vektor és a mutatók közötti analógiára), azonban semmilyen információ nem jut el a függvényhez a vektor méretével (elemeinek számával) kapcsolatban. Ezt az adatot egy második paraméter felhasználásával adhatjuk meg. Kivételt képeznek azok az egydimenziós karaktertömbök, amelyekben sztringet tárolunk, hisz karaktersorozatok esetén a memória tartalmazza a sztring végét jelölő 0-ás bájtot.

Nézzünk néhány jellegzetes megoldást a vektorokat feldolgozó függvények kialakítására! Az alábbi példák mindegyike az átadott egész típusú vektor elemeinek átlagát határozza meg, és adja vissza függvényértékként. A hívás bemutatásához az alábbi 10-elemű vektort használjuk:

```
int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Az első példában **int*** mutatóként fogadjuk a vektor kezdőcímét, és a függvény törzsében is pointer-műveletekkel érjük el az elemeket. Amennyiben a vektor elemeit nem kívánjuk megváltoztatni, konstansra mutató pointert definiálunk paraméterként: (**const int** * *vektor*). Ha emellett a mutató állandóságát is biztosítani szeretnénk, konstansra mutató konstans pointerben fogadjuk a tömb címét: (**const int** * **const** *vektor*).

```
double atlag1 (const int * vektor, int n)
{
  long sum = 0;
  for (int i = 0; i < n; i++)
      sum += *(vektor+i);        // vagy: sum += vektor[i];
  return (double)sum / n;
}
```

A következő példában **int**[] típussal deklaráljuk a függvény *vektor* paraméterét. Az **int**[] deklaráció csak annyit közöl a vektorról, hogy egész elemeket tartalmaz, így hatása megegyezik az **int** * **const** deklarációéval. (Ha a zárójelek között valamilyen egész kifejezést adunk meg, akkor azt figyelmen kívül hagyja a fordító.) Az elemek állandóságát itt is a **const** típusmódosítóval biztosíthatjuk: **const int** *vektor[]*.

```
double atlag2 (const int vektor[], int n)
{
  long sum = 0;

  for (int i = 0; i < n; i++)
      sum += vektor[i];
  return (double)sum / n;
}
```

Mindkét függvényt ugyanúgy kell hívni:

```
double b1 = atlag1 (a, 10);

double b2 = atlag2 (a, 10);
```

A vektor címét és méretét más módon is megadhatjuk, például egy struktúrában egyesítve a két adatot:

```
struct vec {
    int *ptr;      // a vektor címe
    int size;      // az elemek száma
};
```

Ezt a struktúrát értékként adjuk át az átlagot számító függvénynek, melynek prototípusa:

```
double atlag3 (vec vs);
```

A hívás módja természetesen eltér az előző függvényekétől, hisz a hívás előtt fel kell töltenünk egy *vec* típusú struktúrát a szükséges adatokkal:

```
vec v={a,10};
double b2  = atlag3 (v);
```

A programozás során gyakran mutatókat tartalmazó vektort használunk kétdimenziós tömbök helyett. Az ilyen vektor argumentumként való átadása az előzőekben bemutatott megoldások bármelyikével elvégezhető. Az alábbi függvények az átadott sztringtömb (karakterre mutató pointerek vektora) elemeit egymás mellé írja, szóközzel tagolva. (A sztringtömbben a sztringeket ***NULL*** pointer zárja.) A megoldást pointeres (*print1()*) és vektoros (*print2()*) felfogásban az alábbi program tartalmazza.

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

void print1(char **p)
{
  while (*p)
     cout<<*p++<<' ';
  cout<<endl;
}

void print2(char *p[])
{
  int i=0;
  while (p[i])
        cout<<p[i++]<<' ';
  cout<<endl;
}

void main()
{
  char *Desc[] ={ "Cogito", "ergo", "sum.", NULL };
  print1(Desc);
  print2(Desc);
  cin.get();
}
```

8.4.3.2. Kétdimenziós tömb argumentumok

A kétdimenziós tömb elnevezés alatt most csak a fordító által (statikusan) létrehozott tömböket értjük:

```
int a[3][4];
```

A tömb elemeire való hivatkozás (*a[i][j]*) mindig átírható a **((**int***)a+(i*4)+j)* formula szerint (ezt teszik a fordítók is). Ebből a kifejezésből is látszik, hogy a kétdimenziós tömb második dimenziója (*4*) alapvető fontossággal bír a fordító számára, míg a sorok száma tetszőleges lehet.

Végső célunk olyan függvény készítése, amely tetszőleges méretű kétdimenziós tömb elemeit mátrixos formában jeleníti meg. Első lépésként írjuk meg a függvény 3x4-es tömbök megjelenítésére alkalmas változatát!

```
void PrintMat34(const int matrix[3][4]) // hívás: PrintMat34(a);
{
  for (int i=0; i<3; i++) {
     for (int j=0; j<4; j++)
       cout<<'\t'<<matrix[i][j];
     cout<<endl;
  }
}
```

A kétdimenziós tömb is a terület kezdőcímét kijelölő mutatóként adódik át a függvénynek. Azonban a fordító az elemek elérése során

```
*( (int *)mx + (i*4)+j )
```

figyelembe veszi azt, hogy a sorok 4 elemet tartalmaznak. Ezért a fenti függvény egyszerűen átalakítható olyan függvénnyé, amely *nx4*-es tömb kiírására alkalmas, csak a sorok számát kell átadni második argumentumként:

```
void PrintMatn4 (const int matrix[][4], int n)
// hívás: PrintMatn4(a, 3);
{
  for (int i=0; i<n; i++) {
     for (int j=0; j<4; j++)
       cout<<'\t'<<matrix[i][j];
     cout<<endl;
  }
}
```

Arra azonban nincs lehetőség a C++ nyelvben, hogy a második dimenziót is elhagyjuk, hiszen akkor a fordító nem képes a tömb sorait azonosítani. Egyetlen dolgot tehetünk az általános megoldás megvalósításának érdekében, hogy átvesszük a tömbterület elérését a fordítótól (a fenti kifejezés felhasználásával):

```
void PrintMatnm(void *mx, int n, int m)
// hívás: PrintMatnm(a, 3, 4);
{
  for (int i=0; i<n; i++) {
     for (int j=0; j<m; j++)
        cout<<'\t'<<*((int *)mx+i*m+j);
     cout<<endl;
  }
}
```

8.4.4. Sztringargumentumok

A karaktersorozat-argumentumok a már ismertetett módon, egydimenziós karaktertömbökként adódnak át a függvényeknek. Az ok, amiért mégis külön alfejezetet szentelünk e témának, a sztringek feldolgozásánál használt fogások bemutása. A sztringek kezelése során alapvetően két megközelítési módot használhatunk. Az első esetben, mint vektort kezeljük a sztringet (indexek alapján), a másik megközelítés szerint mutató segítségével végezzük el a szükséges műveleteket.

Az első példában mindkét megoldást bemutatjuk az ***strcpy()*** (sztringmásolás) könyvtári függvény működését megvalósító függvényben. A további példákban azonban felváltva használjuk a két megközelítési módot.

A szabványos könyvtár ***strcpy()*** függvénye

```
char * strcpy (char * celstr, const char * forrasstr);
```

a *forrasstr* sztring tartalmát átmásolja a *celstr* sztringbe, és a célsztringre mutató pointerrel tér vissza. A műveletet mutatókkal megvalósító függvény:

```
char * pstrcpy(char *p, const char *q)
{
   char *s = p;    // a célterület
   while (*p++ = *q++);
   return s;
}
```

A *pstrcpy()* függvényben a *p* és *q* mutatókkal dolgozunk. Ahhoz, hogy a célterületet kijelölő mutatót a függvény végén vissza tudjuk adni, az *s* segédváltozóban megőrizzük azt. A függvénynek nincs külön paramétere, amely a forrássztring hosszát tartalmazná. Erre nincs is szükségünk, hisz a sztringet záró '\0' karakter (0-ás bájt) vizsgálatával végig tudunk lépkedni a sztringen. A sztringeken való végighaladást (++) és a karakterek másolását (=) egyetlen **while** ciklusba sűrítettük. A ciklus akkor áll le, amikor a 0-ás bájt is átmásolódott. A ciklust természetesen kevésbé tömör formában is fel lehet írni:

```
while (*p = *q) {
      p++;
      q++;
}
```

vagy

```
while (*q) {
     *p = *q;
      p++;
      q++;
}
*p=0;
```

Amennyiben vektorként kívánjuk a sztringet feldolgozni, szükségünk van egy indexváltozóra, amellyel a vektorokat indexeljük:

```
char * vstrcpy(char p[], const char q[])
{
   int i=0;          /* indexeléshez
   while (p[i] = q[i]) i++;
   return p;
}
```

A másolást leállító feltétel szintén a sztringet záró 0-ás bájt elérése. Ehhez a bájthoz az indexváltozó (*i*) léptetésével jutunk el. Összehasonlítás kedvéért írjuk fel a másoló ciklus kevésbé tömör alakját is!

```
for (int i=0 ; q[i]; i++)
   p[i] = q[i];
p[i] = 0;
```

Láthatjuk, hogy mutatók segítségével tömörebben lehet megfogalmazni a feladat megoldását. Azonban ez a tömörség a program olvashatóságát lényegesen rontja.

A következő függvény az első argumentumként átadott sztringhez hozzámásolja a másik argumentumban megadott sztring tartalmát (*pstrcat()*):

```
char * pstrcat(char *p, const char *q)
{
  char *s = p;
   while (*p) p++;          // lépkedés a célsztring végére
   while (*p++ = *q++);     // másolás
   return s;
}
```

A *vindex()* függvény az *s1* sztringben megkeresi az *s2* sztring első előfordulását, és visszatér a helyet azonosító indexszel. A *-1* függvényérték azt jelzi, hogy nem található meg az *s2* az *s1*-ben:

```
int vindex(const char s1[], const char s2[])
{
   int j, k;
   for (int i=0; s1[i]; i++)    // Lépkedés az s1-ben
   {
      // Az s1 i. pozíciójától van-e az s2 ?
      for (j=i, k=0; s2[k] && s1[j] == s2[k]; j++, k++);
      // Ha a leállás feltétele az s2 vége - benne van!
      if (s2[k] == '\0') return i;
   }
  return -1; // Nincs benne
}
```

8.4.5. A függvény, mint argumentum

Matematikai alkalmazások készítése során jogos igény, hogy egy jól megvalósított algoritmust különböző függvények esetén tudjuk használni. Ehhez a szükséges függvényt mint argumentumot kell átadni az algoritmust megvalósító függvénynek.

8.4.5.1. A függvénytípus és a typedef

Mielőtt megismerkednénk a függvényre mutató pointerekkel, nézzük meg a **typedef** tárolási osztály felhasználását függvények esetén! A **typedef** segítségével a függvény típusát egyetlen szinonim névvel jelölhetjük. A függvénytípus deklarálja azt a függvényt, amely az adott számú és típusú paraméterhalmazzal rendelkezik, és a megadott adattípussal tér vissza. Tekintsük például a faktoriálist számító függvényt, melynek prototípusa és definíciója:

```
unsigned long fakt(int);          // prototípus

unsigned long fakt(int n)         // definíció
{
   unsigned long f = 1;
   for ( ; n > 0 ; n--) f *= n;
   return f;
}
```

Most pedig vegyük a függvénydefiníció fejsorát, tegyük elé a **typedef** kulcsszót, majd pontosvesszővel zárjuk le azt! A keletkező új típus neve legyen *faktfv*!

```
typedef unsigned long faktfv(int n);
```

A **typedef** deklarációban, ellentétben a prototípussal, a paraméterek nevét is érdemes megadni, mivel ekkor a típusnév a függvény definíciójában is alkalmazható. A *faktfv* típus felhasználásával a *fakt()* függvény prototípusa és definíciója az alábbi alakban írható fel:

```
faktfv fakt;                      // prototípus

faktfv fakt                       // definíció
{
   unsigned long f = 1;
   for ( ; n > 0 ; n--) f *= n;
   return f;
}
```

8.4.5.2. Függvényre mutató pointerek

A C++ nyelvben a függvényneveket kétféle módon használhatjuk. A függvénynevet a függvényhívás operátor bal oldali operandusaként megadva függvényhívás kifejezést kapunk

```
fakt(7)
```

melynek értéke a függvény által visszaadott érték. Ha azonban a függvénynevet önállóan használjuk

```
fakt
```

akkor egy mutatóhoz jutunk, melynek értéke az a memóriacím, ahol a függvény kódja elhelyezkedik (kódpointer), típusa pedig a függvény típusa.

Definiáljunk egy olyan mutatót, amellyel a *fakt()* függvényre mutathatunk, vagyis értékként felveheti a *fakt()* függvény címét! A definíciót egyszerűen megkapjuk, ha a *fakt* függvény fejsorában szereplő nevet a (**fptr*) kifejezésre cseréljük:

```
unsigned long (*fptr) (int);
```

Az *fptr* olyan pointer, amely **unsigned long** visszatérési értékkel és egy **int** típusú paraméterrel rendelkező függvényre mutathat.

A definíció azonban sokkal olvashatóbb formában is megadható, ha használjuk a **typedef** segítségével előállított *faktfv* típust:

```
faktfv *fptr;
```

Az *fptr* nem csak mutathat, hanem rá is mutat az alábbi értékadás végrehajtása után a *fakt()* függvényre:

```
fptr = fakt;
```

Ezek után a *fakt* függvény az *fptr* mutató felhasználásával indirekt módon is meghívható:

```
f10 = (*fptr) (10);     vagy     f10 = fptr (10);
```

A **ftpr* kifejezést azért kell zárójelben használnunk, mert a függvényhívás operátora erősebb precedenciájú az indirekt hivatkozás operátoránál. Az *fptr* mutató természetesen tetszőleges, a *fakt()* függvénnyel megegyező típusú függvény címét felveheti.

A függvény neve és a függvényre mutató pointer között hasonló összefüggés van, mint a tömb neve és a tömbelem típusára mutató pointer között. (Mind tömbnév, mind pedig a függvénynév konstans mutatóként viselkedik.) Csak emlékeztetőül a tömb és a mutatók közötti kapcsolat:

```
int a[10], *p = a;

a[0]   = a[1];              p[0]   = p[1];
*(a+0) = *(a+1);            *(p+0) = *(p+1);
```

Nézzük meg, mi a helyzet az *fv* függvény és a rá hivatkozó *pfv* esetén:

```
void fv (int);
void (*pfv) (int) = fv;

// A lehetséges függvényhívások:
fv(2);                      pfv(2);
(*fv)(2);                   (*pfv)(2);
```

Az elmondottak alapján megérthetjük a ***qsort()*** könyvtári függvény prototípusát:

```
void qsort(void *base, size_t nelem, size_t width,
           int (*fcmp)(const void *, const void *));
```

A függvénnyel *base* címen kezdődő, *nelem* elemszámú, elemenként *width* bájtot foglaló tömböt rendezhetünk sorba. A rendezés során hívott összehasonlító függvényt magunknak kell megadni az *fcmp* paraméterben. Az alábbi példában a ***qsort()*** függvényt egy egész, illetve egy sztringtömb rendezésére használjuk:

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

int icmp(const void *p, const void *q) {
 return *(int *)p-*(int*)q;
}

int scmp(const void *p, const void *q) {
 return strcmp((char *)p,(char *)q);
}

void main()
{
    int m[5]={3,0,6,1,7};
    char nevek[3][20]={"Dennis Ritchie", "Bjarne Stroustrup",
                       "Ken Thompson"};

    qsort(m,5,sizeof(int), icmp);
    for (int i=0; i<5; i++)
      cout<<m[i]<<endl;

    qsort(nevek,3,20, scmp);
    for (int i=0; i<3; i++)
      cout<<nevek[i]<<endl;
    cin.get();
}
```

A következő példaprogramokban szereplő *tabloz()* függvény tetszőleges **double** típusú paraméterrel és **double** visszatérési értékkel rendelkező függvény értékeinek táblázatos megjelenítésére alkalmas. A *tabloz()* függvény paraméterei között szerepel még az intervallum két határa és a lépésköz.

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

// A tabloz() függvény prototípusa
void tabloz(double (*)(double), double, double, double);

double sqr(double);  // a saját függvény prototípusa
double sqrt(double); // a könyvtári függvény prototípusa
```

```
void main() {
   cout<<"\n\nAz sqr() fuggveny ertekei ([-2,2] dx=0.5)"<<endl;
   tabloz(sqr, -2, 2, 0.5);
   cin.get();
   cout<<"\n\nAz sqrt() fuggveny ertekei ([0,2] dx=0.2)"<<endl;
   tabloz(sqrt, 0, 2, 0.2);
   cin.get();
}

// A tabloz() függvény definíciója
void tabloz(double (*fp)(double), double a, double b,
            double lepes){
   double x;
   for (x=a; x<=b; x+=lepes)
       printf("%13.5f \t %15.10f\n", x, (*fp)(x));
}

// Az sqr() függvény definíciója
double sqr(double x) {
   return x * x;
}
```

8.4.5.3. Függvényre mutató pointerek tömbje

C++ nyelven gyakran használt, hasznos adatstruktúra a függvényre mutató pointerek tömbje, amely szintén lehet egy- vagy többdimenziós. Elsősorban olyan esetekben használjuk ezeket a tömböket, amikor valamilyen választás (például menüből menüpont kiválasztása) eredményét gyorsan kívánjuk feldolgozni.

Tegyük fel, hogy a programunk vezérlését egyszerű menüvel szeretnénk megoldani!

```
1. Beolvas
2. Szamol
3. Kiir
----------
0. Kilep
```

A hagyományos megoldás során ciklusban fogadjuk a karaktereket, és az **if** vagy a **switch** utasítással kiválasztjuk a megfelelő esetet

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

// A menü definíciója
char * menu[] = { "\n", "1. Beolvas",
                        "2. Szamol",
                        "3. Kiir",
                        "----------",
                        "0. Kilep" , "\n", NULL };
```

```
// A függvények prototípusa
void beolvas(void);
void szamol(void);
void kiir(void);
// Globális változók, közös használatra
int a, b, c;
void main() {
  char ch , **p;

  do {
      // a menü kiírása
      p = menu;
      while (*p)
           cout<<*p++<<endl;
      // A választás feldolgozása
      cin>>ch; cin.get();
      switch (ch) {
         case '0':   break;
         case '1':   beolvas();
                     break;
         case '2':   szamol();
                     break;
         case '3':   kiir();
                     break;
         default:    cout<<'\a';
      }
  } while (ch != '0');
}

// A menüpontnak megfelelő függvények definíciója
void beolvas(void) {  // biztonságos adatbevitel
  char sor[64],ch;
  cout<<"Kerek ket szamot [a,b]: ";
  cin.getline(sor,64);
  istrstream ist(sor);
  ist>>a>>ch>>b;
}

void szamol(void) {
  c = a + b;
}

void kiir(void) {
  cout<<a<<" + "<<b<<" = "<<c<<endl;
}
```

Nézzük meg, hogyan módosul a ***main***() függvény a mutatótömb bevezetésével! A fenti három (azonos típusú) függvény címének tárolására használható *menupont* vektor definíciója (kezdőértékek megadásával):

```
void (*menupont[3]) (void) = { beolvas, szamol, kiir };
```

A ***main()*** függvény sokkal rövidebb és áttekinthetőbb lett. Meg kell jegyeznünk, hogy minél több a menüpont (például kétdimenziós menü esetén), annál hatékonyabb ez a megoldás. A tömbelem által kijelölt függvény (például a *szamol*) hívása:

```
(* menupont[1])();   vagy   menupont[1]();
```

A módosított ***main()*** függvény:

```
void main()
{ char ch , **p;
  void (*menupont[3]) (void) = { beolvas, szamol, kiir };
  do {
      p = menu;
      while (*p)
          cout<<*p++<<endl;
      cin>>ch; cin.get();
      if (ch > '0' && ch < '4')
          menupont[ch-'1'] ();
      else if (ch != '0')
              cout<<'\a';
  } while  (ch!='0');
}
```

Hasonlóan érdekes probléma az egyszerű, kötött szerkezetű leírónyelven megírt programok (például NC, HP plotter stb.) gyors értelmezése. A parancsok nagy száma miatt a vezérlést végző függvény áttekinthetetlen és kusza lesz, ha a szokásos **if** vagy **switch** szerkezetet használjuk. Szép megoldáshoz jutunk azonban olyan mutatótömb kialakításával, melynek elemei a parancsoknak megfelelő függvényekre mutatnak.

Tegyük fel, hogy a bemutatásra kerülő program által feldolgozott leírónyelv utasításai „***An*** *paraméterek*" szerkezetűek! Az első betű ('A'-'Z') a parancs csoportját, míg a második számjegykarakter ('0'-'9') a parancsot jelöli ki. Ez összesen 26x10=260 különböző parancs megadását teszi lehetővé. A kétkarakteres parancsot szóközzel elválasztva követik a parancs paraméterei.

A feldolgozó függvények egy mutatót kapnak argumentumként, amely az utasításban a *paraméterek* rész elejére mutat. Ezért e függvények prototípusa:

```
void parancs(char *);
```

A 260 parancsot kezelő függvény címének tárolására kétdimenziós tömböt használunk, így a parancs azonosítását az utasítás első két karaktere alapján elvégezhetjük:

```
void (* fvt[26][10]) (char *);
```

A feladat egy lehetséges megoldása az alábbiakban tanulmányozható:

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

void p0(char *);
void p1(char *);
void q0(char *);
void ures(char *);

// A feldolgozandó programrészlet
char *prog[]={ "P0 100,300",
               "P1 Hello",
               "P2 100,LEFT",
               "P0 200,500",
               "Q1 500",
               "P1 Bye",
               "Q0 "};

void  main() {
   void (* fvt[26][10]) (char *);
   int i, j, n;

for (i = 0; i < 26; i++ )        // inicializálás:  ures
      for (j = 0; j < 10; j++ )
          fvt[i][j] = ures;

  fvt['P'-'A']['0'-'0'] = p0;    // inicializálás:  p0, p1 és q0
  fvt['P'-'A']['1'-'0'] = p1;
  fvt['Q'-'A']['0'-'0'] = q0;

  // A példaprogram feldolgozása
  n=sizeof(prog)/sizeof(prog[0]);
  for ( i = 0; i < n; i++ )
   (*fvt[ prog[i][0]-'A' ][ prog[i][1]-'0' ])(&prog[i][3]);
  cin.get();
}

// A parancsfüggvények definíciója

void ures (char * p) {
  cout<<"**** Ervenytelen parancs ****"<<endl;
}

void p0 (char * p) {
  int a,b;
  char ch;
  istrstream ist(p);
  ist>>a>>ch>>b;
  cout<<a<<" + "<<b<<" = "<<a+b<<endl;
}
```

```
void p1 (char * p) {
  cout<<p<<endl;
}

void q0 (char * p) {
  cout<<"--- A feldolgozas vege ---"<<endl;
}
```

Ugyancsak érdemes egy pillantást vetni a feldolgozás eredményeként megjelenő képernyőtartalomra:

```
100 + 300 = 400
Hello
**** Ervenytelen parancs ****
200 + 500 = 700
**** Ervenytelen parancs ****
Bye
--- A feldolgozas vege ---
```

8.4.6. Alapértelmezés szerinti (default) argumentumok

A C++ függvények prototípusában bizonyos paraméterekhez ún. alapértelmezés szerinti értéket rendelhetünk. A fordító ezeket az értékeket használja fel a függvény hívásakor, ha az adott argumentum nem szerepel a hívási listában:

```
double defargfv(int a, double b=3.14, char c='K');
```

A példából is látható, hogy az alapértelmezés szerinti értékkel ellátott paraméterek jobbról-balra haladva folytonosan helyezkednek el. A fenti függvény lehetséges hívásait felsorolva nézzük meg a paraméterek tényleges értékét!

Hívás	*Paraméterek:*	*A*	*b*	*c*
`defargfv(10);`		*10*	*3.14*	*'K'*
`defargfv(10,2.5);`		*10*	*2.5*	*'K'*
`defargfv(10,2.5,'X');`		*10*	*2.5*	*'X'*

Nem megengedett hívási forma például a *defargfv(10,,'X');*. Ha valamely alapértelmezett argumentumot elhagyjuk, akkor az azt követő alapértelmezés szerinti értékkel ellátott argumentumokat is el kell hagynunk.

A következő példában használt *terulet()* függvény háromszög területét határozza meg az oldalak ismeretében. (Derékszögű háromszög esetén a két befogó felhasználásával a terület egyszerűbben kiszámítható.) Amennyiben prototípust is használunk, akkor az alapértelmezés szerinti értékeket csak a prototípusban adhatjuk meg.

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

double terulet(double a, double b, double c=0);
```

```
void main() {
   // Általános háromszög területe:
   cout <<"\nÁltalános : " << terulet(3,4,5);

   // Derékszögű háromszög területe:
   cout <<"\nDerékszögű: " << terulet(3,4);
}

double terulet(double a, double b, double c) {
  if (c) {
     double s=(a+b+c)/2;
     return sqrt(s*(s-a)*(s-b)*(s-c));
  }
  else
    return(a*b/2);
}
```

Az alapértelmezés szerinti argumentumokkal ellátott prototípusok rugalmasabbá teszik a függvények használatát. Például, ha valamely függvényt sokszor hívunk egyazon argumentumlistával, érdemes a gyakran használt argumentumokat alapértelmezés szerintivé tenni, és a hívást argumentumok megadása nélkül elvégezni.

8.4.7. Változó hosszúságú argumentumlista

Bizonyos függvények esetén nem lehet pontosan megadni az argumentumok számát és típusát. Az ilyen függvények deklarációjában a paraméterlistát három pont zárja:

```
int printf(const char * formatumsztring, ... );
```

A három pont azt jelenti a fordítóprogram számára, hogy „még lehetnek további argumentumok". A ***printf()*** függvény esetén legalább egy argumentumnak kell szerepelnie, amelyet tetszőleges számú további argumentum követhet:

```
printf("Hello C++ !\n");
printf("A nevem: %s \n", nev);
printf("Az összeg: %d + %d = %d\n", a, b, c);
```

Felvetődik a kérdés, honnan tudja a ***printf()***, hogy hány argumentumot kell feldolgoznia? A választ a *formatumsztring* adja, melyen végiglépkedve a formátum alapján dolgozza fel a ***printf()*** függvény a soron következő argumentumot.

Mivel az ilyen deklarációjú függvények hívásakor a fordító csak a „. . ." listaelemig képes az argumentumok típusát egyeztetni, ezért a további argumentumok esetén a hagyományos konverziókat hajtja végre. Más szavakkal, ebben az esetben a megadott argumentumok (esetleg konvertált) típusa szerint megy végbe az argumentumok átadása a függvénynek.

A C++ nyelv lehetővé teszi, hogy saját függvényeinkben is használjuk a három pontot - az ún. változó hosszúságú argumentumlistát. Ahhoz, hogy a paramétereket tartalmazó memóriaterületen megtaláljuk az átadott argumentumok értékét, legalább az első paramétert mindig meg kell adnunk.

A C++ szabvány tartalmaz néhány olyan makrót, amelyek segítségével a változó hosszúságú argumentumlista feldolgozásához nem kell ismernünk az adott számítógépes környezet „lelkivilágát". Az *cstdarg* fejállományban deklarált, illetve definiált makrók a következők:

`type va_arg(va_list ap, type);`	Az argumentumlista következő elemét adja vissza.
`void va_end(va_list ap);`	„Nagytakarítás" az argumentumok feldolgozása után.
`void va_start(va_list ap, lastfix);`	Inicializálja az argumentumok eléréséhez használt mutatót.

A fenti makrók a ***va_list*** típusú mutatót használják az argumentumok eléréséhez.

Példaként tekintsük a tetszőleges számú **int** érték összegét kiszámító *osszeg()* függvényt, melynek hívásakor a számsort 0-val kell zárni! Az *atlag()* függvény segítségével adott darabszámú **double** érték átlagát számoljuk. A darabszámot a függvény első argumentumaként kell megadni.

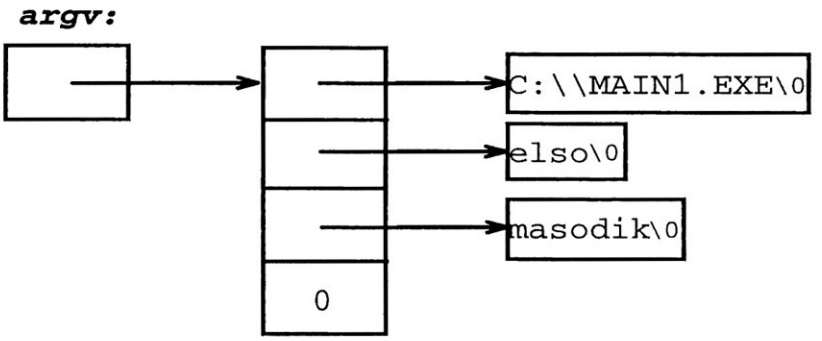
```
// Egész számok összegzése 0-ig
int osszeg(int elso, ...) {
  va_list  ap;
  int      s = elso, ertek;
  va_start( ap, elso);              // az első argumentum átlépése
  while (ertek = va_arg(ap, int))   // int argumentumok
        s += ertek;
  va_end(ap);
  return s;
}

// Adott számú double érték átlagolása
double atlag(int n, ...) {
  va_list  ap;
  double   s = 0;
  va_start(ap, n);                  // az első argumentum átlépése
  for (int i=0; i<n; i++)
       s += va_arg(ap,double);      // double argumentumok
  va_end(ap);
  return s / n;
}
```

8.4.8. A main() függvény paraméterei és visszatérési értéke

A *main()* függvény különlegességét nemcsak az adja, hogy a program végrehajtása vele kezdődik, hanem az is, hogy nulla, egy vagy két paramétere lehet:

```
int main( )

int main( int argc) ( )

int main( int argc, char *argv[]) ( )
```

A paraméterek neveit tetszőlegesen megválaszthatjuk, azonban a program olvashatóságát jelentősen javítja a szabványos elnevezések használata. Az *argv* egy karaktermutatókat tartalmazó tömbre (vektorra) mutat, az *argc* pedig a tömbben található sztringek számát adja meg. (Az *argc* értéke legalább 1, mivel az *argv[0]* mindig a program nevét tartalmazó sztringre hivatkozik.)



A *main()* visszatérési értékét, amely általában **int** típusú, a *main()* függvényen belüli **return** utasításban, vagy a program tetszőleges pontján az *exit()* könyvtári függvény argumentumában adhatjuk meg. Az *cstdlib* fejállomány szabványos konstansokat is tartalmaz,

```
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

amelyeket kilépési kódként használva, a program sikeres, illetve sikertelen futását jelölhetjük. A *main()* függvényt **void** típusúnak definiálva a programunknak határozatlan lesz a kilépési kódja.

Az alábbi példaprogram (*MAIN1.cpp*) a segédprogramok írásánál jól felhasználható megoldást mutat be az indítási argumentumok helyes megadásának tesztelésére. A program csak akkor indul el, ha pontosan két argumentummal indítjuk. Ellenkező esetben hibajelzést és az indítási parancssor formáját bemutató üzenetet ír ki a képernyőre.

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

int main(int argc, char *argv[]) {
  int i;
  if (argc !=3 ) {
      cerr<<"Hibas parameterezes!"<<endl;
      cerr<<"A program inditasa: MAIN1 arg1 arg2"<<endl;
      return EXIT_FAILURE;
  }
  cout<<"Helyes parameterezes:"<<endl;
  cout<<"1. argumentum: "<<argv[1]<<endl;
  cout<<"2. argumentum: "<<argv[2]<<endl;
  cin.get();
  return EXIT_SUCCESS;
}
```

A ***main()*** függvény argumentumainak megadása, illetve a visszatérési érték feldolgozása függ az operációs rendszertől. Az MS-DOS-ban az argumentumokat a programot indító parancssorból másolja be a futtató rendszer az *argv* által kijelölt tömbbe.

Ha a program neve *MAIN2*, akkor a hívási argumentumok megadására a

```
C:\CPP\MAIN2 elso masodik 3. 4.
```

parancssor használható. Az argumentumok elválasztására a szóközt használjuk.

Az alábbi *MAIN2.cpp* program kiírja a parancssor-argumentumok számát, megjeleníti az argumentum-karaktersorozatokat és az *envp* által mutatott sztringtömb tartalmát. Az *envp* a DOS-ban a környezeti változókat tartalmazó sztringtömböt jelöli, melyben az érvényes elemeket egy ***NULL*** érték zárja.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(int argc, char *argv[], char **envp) {
  int i;
  char **p;

  cout<<"Az argumentumok szama: "<<argc<<endl<<endl;
  for (i=0; i<argc; i++)
      cout<<i<<". argumentum: \""<<argv[i]<<"\""<<endl;

  p = envp;
  while (*p)
      cout<<"Kornyezeti valtozo: \""<<*p++<<"\""<<endl;
  cin.get();
  return (EXIT_SUCCESS);
}
```

A fenti parancssort használva, a program futásának eredménye:

```
Az argumentumok szama: 5

0. argumentum: "C:\MAIN2.EXE"
1. argumentum: "elso"
2. argumentum: "masodik"
3. argumentum: "3."
4. argumentum: "4."

Kornyezeti valtozo:"COMSPEC=C:\COMMAND.COM"
Kornyezeti valtozo: "PATH=C:\DOS;C:\DOS\HELP;
Kornyezeti valtozo: "PROMPT=$P$G"
```

8.5. Rekurzív függvények használata

A matematikában lehetőség van bizonyos adatok és műveletek *rekurzív* definiálására. Minden *rekurzív* problémának létezik *iteratív* (ciklust használó) megoldása, amely általában sokkal nehezebben programozható, de hatékonysága miatt mégsem szabad megfeledkezni róla! Klasszikus példaként tekintsük először a *Fibonacci* néven is ismert *Leonardo de Pisa* nyúl feladatát:

> "Hány nyúlpárunk lesz 3,4,5,...,n hónap múlva, ha egy nyúlpár kéthónapos kortól kezdve havonta egy-egy új párt hoz világra, feltéve, hogy az új párok is e törvény alapján szaporodnak, és mind életben maradnak."

A megoldást a *Fibonacci* számok sora tartalmazza (ha a 0 kezdőelemet figyelmen kívül hagyjuk):

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

A sor *n*-dik elemének meghatározására az alábbi rekurziós szabály szolgál:

$$a_0 = 0$$
$$a_1 = 1$$
$$a_n = a_{n-1} + a_{n-2}, \qquad n = 2, 3, 4, \ldots$$

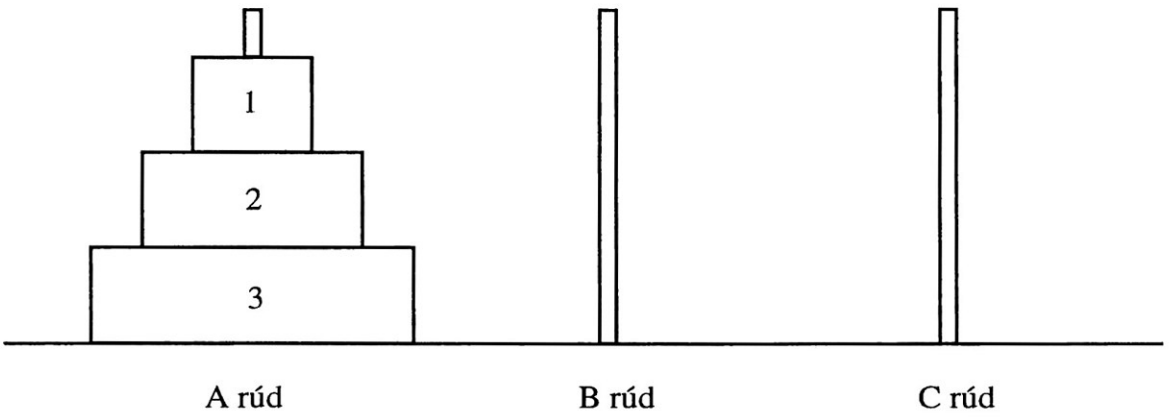
A rekurziós szabály alapján elegáns megoldást kapunk, ha önmagát hívó rekurzív függvényt használunk. Ekkor gyakorlatilag a fenti összefüggést fogalmazzuk át C++ programmá:

```
unsigned long fibr( int n ) {
   if (n<2)
      return n;
   else
      return fibr(n-1) + fibr(n-2);
}
```

A rekurzív megoldás általában rövidebb és áttekinthetőbb, mint az iteratív megoldás, azonban a számítási idő és a memóriaigény jelentős növekedése miatt az esetek többségében mégis az iteratív megoldás használatát javasoljuk:

```
unsigned long fib( int n ) {
  unsigned long f0 = 0, f1 = 1, f2 = n;

  while (n-- > 1) {
        f2 = f0 + f1;
        f0 = f1;
        f1 = f2;
  }
  return f2;
}
```

A következő feladat megoldása a rekurzió alkalmazása nélkül igen bonyolult. A megoldandó probléma a Hanoi tornyai nevet viseli.



Adott három rúd: *A*, *B* és *C*. Az *A* rúdon induláskor *N* darab különböző átmérőjű lyukas korong helyezkedik el, az átmérők csökkenő sorrendjének megfelelően. A feladat a korongok átrakása a *C* rúdra, az alábbi szabályok figyelembevételével:

- a *B* rúd közbenső tárolásra használható,
- minden lépésben csak egy korong mozgatható,
- minden korong csak nálánál nagyobb átmérőjű korongra helyezhető.

(A feladat egy legendán alapul, amely szerint Hanoi közelében található kolostorban 64 aranykorongból álló tornyot raknak át a szerzetesek, a fenti szabályok betartásával, minden nap egyetlen korongot mozgatva. A legenda szerint akkor lesz vége a világnak, ha az átrakást befejezik.)

A feladat lehetséges megoldását az alábbi példaprogram tartalmazza:

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

void Honnan_Hova_Mozgatas(char, char);
void Hanoi(int, char, char, char);
```

```
void main() {
  int korongszam;

  cout<<"A korongok szama : ";
  cin>>korongszam; cin.get();
  cout<<endl;
  Hanoi(korongszam, 'A', 'B', 'C');
  cin.get();
}

void Hanoi(int n, char honnan, char mivel, char hova) {
    if (n == 1)
       Honnan_Hova_Mozgatas(honnan, hova);
    else {
       Hanoi(n-1, honnan, hova, mivel);
       Honnan_Hova_Mozgatas(honnan, hova);
       Hanoi(n-1, mivel, honnan, hova);
    }
}

void Honnan_Hova_Mozgatas(char innen, char ide) {
    cout<<"Tedd a korongot a(z)\t"<<innen<<"\t";
    cout<<" rudrol a(z) \t"<<ide<<"\t rudra.\n";
}
```

Három korong megadása esetén a program által javasolt megoldás:

```
A korongok szama : 3

Tedd a korongot a(z)    A       rudrol a(z)    C      rudra.
Tedd a korongot a(z)    A       rudrol a(z)    B      rudra.
Tedd a korongot a(z)    C       rudrol a(z)    B      rudra.
Tedd a korongot a(z)    A       rudrol a(z)    C      rudra.
Tedd a korongot a(z)    B       rudrol a(z)    A      rudra.
Tedd a korongot a(z)    B       rudrol a(z)    C      rudra.
Tedd a korongot a(z)    A       rudrol a(z)    C      rudra.
```

8.5.1. A rekurzív függvények csoportosítása

A rekurzív algoritmusok általában önrekurzióra vagy kölcsönös rekurzióra épülnek.

8.5.1.1. Önrekurzió

Önrekurzióról akkor beszélünk, amikor egy függvény közvetlenül hívja önmagát. A fenti két példában ezt a megoldást használtuk. A rekurzív működés megértéséhez tekintsük a faktoriális számítását, amely rekurzív megoldással szintén hosszasan működik! Ebben az esetben is az iteratív (ciklusos) megoldás használata javasolt. A faktoriális számítás rekurzív definíciója:

$$n! = \begin{cases} 1, \text{ha } n = 0 \\ n \cdot (n-1)!, \text{ha } n > 0 \end{cases}$$

Ennek alapján *4!* kiszámításának lépései az alábbi szemléletes formában ábrázolhatók:

```
4! = 4 * 3!
          3 * 2!
              2 * 1!
                  1 * 0!
                  1  = 4 * 3 * 2 * 1 * 1 = 24
```

A fenti számítási menetet megvalósító C++ függvény:

```
unsigned long factr (int n)
{
    if (n == 0)
       return 1;
     else
       return n * factr (n-1);
}
```

A faktoriálissal összefüggő, rekurzióval megoldható probléma egy szó betűinek összes lehetséges módon történő összerakása (ismétlés nélküli permutáció). A megoldásban szereplő függvények (*main()*, *swapnm()* és *permut()*) közös globális változókat használnak.

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

// Globális változók a függvények közötti kommunikációhoz
char szo[8];
int nbetu, nperm;

void swapnm(char, char);
void permut(int);

void main()
{
   cout<<"Kerek egy max. 7 betus szot: ";
   cin.getline(szo,8);
   nbetu = strlen(szo);
   nperm = 1;
   if (nbetu > 0 && nbetu <8)
      permut(nbetu-1);
   cin.get();
}
```

```
// A permutációt előállító rekurzív függvény
void permut(int k) {
   for (int i=0; i<=k; i++) {
        swapnm(i, k);
        if (k)
           permut(k-1);
        else
           cout<<"\n\t"<<nperm++<<" - "<<szo;
       swapnm(k, i);
   }
}
// Két karakter felcserélése a szóban
void swapnm( char n, char m) {
    char z;
    z = szo[n];
    szo[n] = szo[m];
    szo[m] = z;
}
```

A program működése során sorszámozza az előállított permutációkat:

```
Kerek egy max. 7 betus szot: C++

        1 - ++C
        2 - ++C
        3 - +C+
        4 - C++
        5 - +C+
        6 - C++
```

8.5.1.2. Kölcsönös rekurzió

Kölcsönös rekurzióról akkor beszélünk, amikor egy függvény az általa hívott másik függvényből hívódik meg újra. Kölcsönös rekurzió esetén különösen oda kell figyelni a rekurziót leállító feltétel megadására. Az ilyen megoldásokban szükséges a függvények prototípusának megadása:

```
typedef void fv(int);          // A függvények típusa
fv fv_a, fv_b;                 // A függvények prototípusa
void fv_a(int i) {
   //. . .
   if (i > 0) fv_b(i / 2);     // Az fv_b hívása
   //. . .
}
void fv_b(int j) {
   //. . .
   if (j > 0) fv_a(j - 1);     // Az fv_a hívása
   //. . .
}
```

8.6. Inline függvények

A C++ előfordító (preprocesszor) ***#define*** direktívájának használata során olyan hibákat vihetünk be a programunkba, amelyek a forráskód áttanulmányozásával nem derülnek ki (kifejezés megadása makróban, vagy a léptető operátorok használata a makró argumentumában). A C++-ban javasolt a ***#define*** használatának korlátozása, hiszen az esetek többségében a **const** és **inline** definíciók kiváltják azt.

Az **inline** („soron belüli") függvényekkel a ***#define*** makrókat helyettesíthetjük. Az **inline** függvény esetén a függvény törzsét képező kód helyettesítődik be a függvényhívás helyére („kódmakró"). Például a *MAX* makró helyett egész értékek esetén *a max()* **inline** függvény használata javasolt.

```
#define MAX(x,y) (x)>(y)?(x):(y)

inline int max(int x, int y)
{
   return (x>y?x:y);
}
```

Nézzük a hivatkozást a makróra és az **inline** függvényre!

```
void main()
{
   int a;
   a=MAX(4,26); // az előfordító a=(4)>(26)?(4):(26); utasítássá
                // alakítja át.

   a=max(4,26); // a fordító a függvény törzsét képező kódot
                // helyettesíti be az utasításba: a=x>y?x:y;
}
```

Tudnunk kell azonban, hogy az **inline** definíció csak javaslat a fordító számára, amelyet az bizonyos feltételek esetén (de nem mindig!) figyelembe is vesz. Általában kisméretű, gyakran hívott függvények esetén ajánlott alkalmazni ezt a megoldást.

Az **inline** függvények használatának előnye, hogy a függvényhíváskor az argumentumok feldolgozása teljes körű típusellenőrzés mellett megy végbe. Talán ez egyben a hátránya is, hiszen a *MAX* makró tetszőleges aritmetikai típus esetén használható, míg a *max()* függvény csak **int** típusú argumentummal hívható. Ezen probléma kiküszöbölésében a C++ egy nagyon fontos mechanizmusa, a függvénynevek átdefiniálása (túlterhelése, *overloading*) segít.

8.7. Függvénynevek átdefiniálása (overloading)

A C++-ban több függvényt is definiálhatunk ugyanazzal a névvel, ha a definiált függvények paraméterlistája („kézjegye") eltér egymástól. Az így definiált függvények közül az éppen szükségeset a fordító választja ki a hívási argumentumok száma és típusa alapján.

Az alábbi példában a *sumall()* függvénynek két átdefiniált formája létezik, az **int** és a **double** típusú tömbök elemösszegének meghatározására:

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

int sumall(int a[], int n)
{
   int sum=0;
   for (int i=0; i<n; i++)
      sum+=a[i];
   return sum;
}

double sumall(double a[], int n)
{
   double sum=0;
   for (int i=0; i<n; i++)
     sum+=a[i];
   return sum;
}

void main()
{
   int        ai[]={1,1,2,3,5,8,13};
   const int ni=sizeof(ai) / sizeof(ai[0]);
   cout << "\nAz int tömb elemösszege: "<<sumall(ai,ni);

   double     ad[]={1.2,2.3,3.4,4.5,5.6};
   const int nd=sizeof(ad) / sizeof(ad[0]);
   cout << "\nA double tömb elemösszege: "<<sumall(ad,nd);
}
```

A fordító az első híváskor *sumall(int *, const int)*, míg a második esetében *sumall(double *, const int)* szignatúrát talál. Ezért például **unsigned** és **float** tömbök esetén fordítási hibát kapunk, hiszen C++-ban a mutatók automatikus konverziója erősen korlátozott.

Láthatjuk, hogy a fenti két függvény csak a tömbök típusában tér el. További típusok bevezetése esetén szövegszerkesztési feladattá válik az újabb átdefiniált függvények megírása (blokkmásolás, szövegcsere). Ennek elkerülése érdekében az átdefiniált függvényváltozatok előállítása is rábízható a fordítóprogramra függvénysablon (*template*) definiálásával. A megfelelő függvényváltozat kiválasztásában a függvény visszatérési típusa nem játszik szerepet.

Átdefiniált nevű függvények esetén a megfelelő függvény kiválasztása több lépésben megy végbe:

1. Keresés a teljes (egzakt) típusegyezőség feltételének felhasználásával. (A teljes típusegyezőség esetén a **float** nem egyezik meg a **double**, és a **char** nem egyezik meg az **int** típussal.)
2. Keresés az ún. triviális típuskonverziók végrehajtásával. A C++ nyelvben az alábbi konverziókat tekintjük triviálisnak. (Az utolsó konverzió a függvény neve és függvényre mutató pointer közötti automatikus konverziót jelöli.)

Típusról	*Típusra*
típus	típus&
típus&	típus
típus[]	típus*
típus	**const** típus
típus	**volatile** típus
fv(argumetumok)	(*fv)(argumentumok)

3. Keresés az egész típusok közötti konverziók (**bool** → **int**, **char** → **int**, **short** → **int**, **unsigned char** → **int**, **unsigned short** → **int**), illetve a **float** → **double** átalakítás felhasználásával.
4. Keresés a szabványos típuskonverziók alkalmazásával. Ezek a konverziók alapvetően a szokásos aritmetikai (**int** → **double**, **signed** → **unsigned**), és a mutatókonverziókat jelentik. A pointerkonverziókat az alábbiakban foglaltuk össze. (A referenciatípusok konverziója a mutatótípusok konverziójának megfelelően megy végbe.)

Típusról	*Típusra*
tetszőleges mutatótípus	**void** *
leszármaztatott osztály mutatója	az alaposztályra mutató pointer
0 konstans	***NULL*** pointer.

5. A típusegyezőség keresése ideiglenes objektum létrehozásával.
6. Keresés a felhasználó által definiált konverziók alkalmazásával.

A függvényekhez hasonló módon használható a műveletek átdefiniálásának mechanizmusa (*operator overloading*). Az operátorokat azonban csak felhasználó által definiált típusokkal lehet átdefiniálni (**struct**, **class**), ezért ezzel a lehetőséggel könyvünk következő részében foglalkozunk részletesen.

8.8. Általánosított függvények (template)

A függvények átdefiniálásánál láttuk, hogy sok esetben ugyanazt a függvényfelépítést használjuk a túlterhelt függvényváltozatokban, csupán néhány típust cserélnünk le. Ilyen esetekben a fordító segíthet nekünk az átdefiniált változatok elkészítésében. Egyetlen dolgunk van, meg kell mondanunk a fordítónak, hogy mely függvények esetén mely típusokat kell lecserélnie, vagyis el kell készítenünk egy általánosított függvénymintát (sablont, *template*-t).

A függvénysablon előállításakor érdemes kiindulnunk egy működő függvényből, amelyben a lecserélendő típusokat általános típusokkal (*TIPUS*) helyettesítjük. Ezek után közölnünk kell a fordítóval, hogy mely típusokat kell lecserélni a függvénymintában (*template<class TIPUS>*). (Az itt szereplő **class** kulcsszónak nincs semmi köze az osztályokhoz.)

```
template<class TIPUS>
TIPUS sumall(TIPUS a[], int n)
{
   TIPUS sum=0;
   for (int i=0; i<n; i++)
      sum+=a[i];
   return sum;
}
```

Az elkészült függvénysablonból a fordító állítja elő a szükséges függvényváltozatokat, amikor először egy hívással találkozik. Az előző alfejezet példája sokkal áttekinthetőbbé válik *template* felhasználásával:

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

template<class TIPUS>
TIPUS sumall(TIPUS a[], int n)
{
   TIPUS sum=0;
   for (int i=0; i<n; i++)
      sum+=a[i];
   return sum;
}
```

```
void main() {
   int        ai[]={1,1,2,3,5,8,13};
   const int ni=sizeof(ai) / sizeof(ai[0]);
   cout << "\nAz int tömb elemösszege: "<<sumall(ai,ni);

   double     ad[]={1.2,2.3,3.4,4.5,5.6};
   const int nd=sizeof(ad) / sizeof(ad[0]);
   cout << "\nA double tömb elemösszege: "<<sumall(ad,nd);

   float      af[]={3,2,4,5};
   const int nf=sizeof(af) / sizeof(af[0]);
   cout << "\nA char   tömb elemösszege: "<<sumall(af,nf);
}
```

A függvénysablonban természetesen több típus helyettesítése is megoldható, mint ahogy az a következő példában látható:

```
template <class T1, class T2>
inline T1 max(T1 a, T2 b)
{
   return (a>b? a : b);
}

void main() {
    cout<<max(5,4)<<endl;          // int max(int,int);
    cout<<max(5.6,4)<<endl;        // int max(double,int);
    cout<<max('A',66.5F)<<endl;    // char max(char, float);
}
```

Az ANSI/ISO C++ nyelvben a függvénysablonokat paraméterezhetjük is. Ebben az esetben a függvény hívásakor a függvény neve mögött meg kell adnunk a *template* argumentumait. Az alábbi példában a *Kiiras()* általánosított függvény hívásakor a típuson túlmenően a mezőszélességet és a pontosságot is beállítjuk.

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

template <class T, int w, int p>
void Kiiras(T a) {
    cout.width(w);
    cout.precision(p);
    cout<<a<<endl;
}

void main() {
    Kiiras<int,10,8>(1234567);
    Kiiras<char *,12,10>("C++ nyelv");
    cin.get();
}
```

8.9. Típusmegőrző szerkesztés (type-safe linking)

A szabványos C++ nyelv támogatja a típusmegőrző szerkesztést, amely segíti a hibás argumentummal történő függvényhívások kiszűrését a szerkesztés folyamán. A típusmegőrző szerkesztés mechanizmusának jobb megértése érdekében nézzünk néhány *Borland C++ Builder* rendszerben használt névképzési példát!

Prototípus	*Tárgykódbeli név*
`double sumall(double a[], int n);`	*@@sumall$qpdi*
`int sumall(int* a, int n);`	*@@sumall$qpii*
`int fv1(int& a, bool n);`	*@@fv1$qrio*
`extern "C" int fv1(int& a, bool n)`	*_fv1*

A C++ fordítók fent bemutatott névképzése teszi lehetővé a függvénynevek átdefiniálását (túlterhelését), hisz ekkor a függvény tényleges nevének kialakítása függ a függvény paraméterezésétől. Megjegyezzük, hogy valamely osztály tagfüggvényeinek nevei az osztály nevét is tartalmazzák.

A függvénynevek ilyen formán történő képzése természetesen megnehezíti más nyelven (például C) írt modulok, könyvtárak C++ nyelvből való hasznosítását. A probléma leküzdésére a C++ nyelv tartalmazza az **extern "C"** típusmódosítót. Ha például C nyelven megírt *sin()* függvényt szeretnénk használni a C++ programból, szükséges az

```
extern "C" double sin(double a);
```

deklaráció megadása. Ennek hatására a C++ fordító a szokásos C-névképzést használja a *sin()* függvényre. Ebből persze az is következik, hogy a nem C++ függvényeket nem lehet átdefiniálni (túlterhelni)! Ha több C függvényt kívánunk meghívni, akkor az **extern** "C" deklaráció csoportos formáját használhatjuk:

```
extern "C" {
   double sin(double a);
   double cos(double a);
}
```

Ha a C-könyvtár függvényeit külön deklarációs állomány tartalmazza, akkor a következő megoldás áll rendelkezésünkre:

```
extern "C" {
   #include <stdio.h>
}
```

8.10. Esettanulmány: C++ deklarációk értelmezése és készítése

A C++ nyelv deklarációinak formája (szintaxisa) jelentősen különbözik más programozási nyelvekben használt deklarációktól. Egy-egy bonyolultabb C++ deklaráció pontos jelentése még a gyakorlott programozóknak sem azonnal érthető. Ezért ebben a fejezetben a C++ típusok összefoglalásaként kitérünk arra, hogyan kell értelmezni, illetve elkészíteni a C++ deklarációkat.

A C++ nyelv ugyanazt az operátor- és szimbólumkészletet használja a deklarációkban, mint a kifejezésekben szereplő azonosítók esetén. Az alábbi példában egy egész típusú változót (*x*), egy mutatót (*px*) és egy referenciát (*rx*) hozunk létre:

```
int x;
int *px;
int &rx=x;
```

A **px* deklarátornak ugyanaz a formája, mint amikor kifejezésben a mutatott egész változót jelöli:

```
x = *px;
```

A példában bemutatott szimmetria jól használható a kifejezések és a deklarátorok típusának megállapításakor: a **px* olyan egész változó, amelyre a *px* pointer mutat.

> A C++ szabvány a deklaráció és a definíció fogalmak mellett a deklarátor fogalmát is használja. Deklarátor alatt a deklarálandó objektumot és a hozzákapcsolt módosítókat (mutató *, referencia &, tömb *[]* vagy függvény *()*) értjük.

8.10.1. C++ deklarációk értelmezése

A fentinél bonyolultabb deklarációk értelmezéséhez azonban további magyarázatok szükségesek. Fontos megjegyeznünk, hogy a C++ deklarátorokban használt szimbólumok valójában C++ operátorok, amelyekre érvényes a precedencia és az asszociativitás szabálya. A precedencia szerint az alábbi két csoportból kerülnek ki a deklarátorokban használt operátorok:

1. A függvényhívás operátora (*()*) és a tömbindexelés operátora (*[]*) azonos precedenciával rendelkeznek. Ezen operátorok csoportosítása balról-jobbra történik.
2. Az indirekt hivatkozás operátora (*) és a címe operátor (&) szintén azonos precedenciával rendelkeznek. Több operátor használata esetén a csoportosítás jobbról-balra történik.

Tekintsük a következő példát!

```
int *x[3];
```

A kérdés, hogy ez most egészre mutató pointerek tömbje, vagy pedig egészeket tartalmazó tömbre mutató pointer? A válaszadás érdekében a kifejezést szétbontjuk, méghozzá a precedencia- és az asszociativitás-szabályokkal ellentétes irányban:

1. `*x[3]` egy egész,
2. `x[3]` egészre mutató pointer,
3. `x` egészre mutató pointerek 3-elemű tömbje.

A legbonyolultabb deklarátorok és kifejezések értelmezése gyorsan elvégezhető ezzel a szétbontásos módszerrel. (A csillagot azért vettük le először, mivel kisebb a precedenciája az indexelés operátorénál.)

A szétbontás általános szabályát is megfogalmazhatjuk. Először sorban levesszük a legkisebb precedenciájú operátorokat. Ezt követően, ha a megmaradt operátorok precedenciája megegyezik, akkor nézzük az asszociativitást. A balról-jobbra csoportosított operátorok esetén először a jobbszélső, míg a jobbról-balra csoportosított operátorok esetén először a balszélső operátort vesszük le. Minden egyes lépés után megfogalmazzuk, hogy mi az, amit kaptunk. A lebontásnál természetesen a zárójeleket is figyelembe kell venni.

Ezzel a szabállyal könnyedén értelmezhetjük a leggyakrabban használt deklarációkat. Sajnos a magyar nyelv nem teszi lehetővé, hogy a típusokat az angol nyelvben használt egyszerűséggel írjuk fel. Nézzünk néhány példát!

```
char **argv
```
karakterre mutató pointerre mutató pointer,

```
int *p;
int * & q = p ;
```
q referencia egy egészre mutató mutatóra,

```
int a[5]
```
egészeket tartalmazó 5-elemű tömb,

```
int (*pa)[5]
```
egészeket tartalmazó 5-elemű tömbre mutató pointer,

```
int *pv[5]
```
egészre mutató pointereket tartalmazó 5-elemű vektor,

```
char *fv(int)
```
függvény, amely **int** argumentumot kap, és karakterre mutató pointerrel tér vissza,

```
char & rfv(int)
```
függvény, amely **int** argumentumot kap, és karakter-referenciával tér vissza,

```
char (*pfv)(int)
```
egy olyan függvényre mutató pointer, amely **int** argumentumot kap karaktert ad vissza.

Az Olvasó megnyugtatására közöljük, hogy ezeknél bonyolultabb deklarációk nem túl gyakran fordulnak elő a C++ programokban.

Tekintsük az alábbi egyszerűnek nem nevezhető deklarációkat, és értelmezzük azokat. A deklarációk értelmezéséhez a lebontás elvét használjuk!

```
int (* (*x()) [6]) ();
```

1. `(*(*x())[6])()` egy egész,
2. `*(*x())[6]` függvény, amely egészet ad vissza,
3. `(*x())[6]` egészet visszaadó függvényre mutató pointer,
4. `*x()` egészet visszaadó függvényre mutató pointerek 6-elemű tömbje,
5. `x()` mutató az egészet visszaadó függvényre mutató pointerek 6 elemű tömbjére,
6. `x` függvény, amely egészet visszaadó függvényre mutató pointerek 6-elemű tömbjére mutató pointerrel tér vissza.

Az előzőhöz formailag hasonló, azonban lényegileg különböző az alábbi deklaráció:

```
int *(* (*y) [6]) ();
```

1. `*(*(*y)[6])()` egy egész,
2. `(*(*y)[6])()` egészre mutató pointer,
3. `*(*y)[6]` egészre mutató pointert visszaadó függvény,
4. `(*y)[6]` egészre mutató pointert visszaadó függvényre mutató pointer,
5. `*y` egészre mutató pointert visszaadó függvényre mutató pointerek 6-elemű tömbje,
6. `y` egészre mutató pointert visszaadó függvényre mutató pointerek 6-elemű tömbjére mutató pointer.

A fenti két függvénydeklaráció után nézzünk egy tömbdeklarációt:

```
char (* (*z[2])() )[5];
```

1. `(*(*z[2])())[5]` egy karakter,
2. `*(*z[2])()` 5-elemű karaktertömb,
3. `(*z[2])()` 5-elemű karaktertömbre mutató pointer,
4. `*z[2]` 5-elemű karaktertömbre mutató pointert visszaadó függvény,
5. `z[2]` 5-elemű karaktertömbre mutató pointert visszaadó függvényre mutató pointer,
6. `z` 5-elemű karaktertömbre mutató pointert visszaadó függvényre mutató pointereket tartalmazó 2-elemű tömb.

8.10.2. C++ deklarációk készítése

A program írása szempontjából a saját deklarációk megfelelő kialakítására kell a hangsúlyt helyezni. A deklaráció felépítése az előző alfejezetben alkalmazott lebontási módszerrel ellentétes művelet, így az ott alkalmazott szabályokat is fordítva kell alkalmaznunk. Azonban az igazán megbízható és olvasható deklarációk csak a **typedef** típusdeklaráció segítségével készíthetők.

A példaként elkészített deklaráció elég bonyolult, azonban ennek megértése után az egyszerűbb (és az összetettebb) deklarációk írása már nem jelenthet gondot.

Készítsük el annak a függvénynek a deklarációját, amely karakterre mutató pointerek 5 elemű tömbjére mutató pointerrel tér vissza!

`F`	függvény, amely karakterre mutató pointerek 5 elemű tömbjére mutató pointerrel tér vissza,
`f()`	karakterre mutató pointerek 5-elemű tömbjére mutató pointer,
`*f()`	karakterre mutató pointerek 5-elemű tömbje,
`(*f())[5]`	karakterre mutató pointer,
`*(*f())[5]`	karakter,
`char` `*(*f())[5];`	a kész deklaráció!

A megoldást a **typedef** felhasználásával, a deklarációk elemi lépéseiből állítjuk elő:

```
// karakterre mutató pointer típusa
typedef char *cp;

// karakterre mutató pointerek 5-elemű tömbjének típusa
typedef cp v5cp[5];

// karakterre mutató pointerek 5 elemű tömbjére mutató
// pointer típusa
typedef v5cp *pv5cp;

// olyan függvény típusa, amely karakterre mutató pointerek
// 5-elemű tömbjére mutató pointerrel tér vissza
typedef pv5cp fpv5pc();

// a g olyan függvény, amely karakterre mutató pointerek
// 5-elemű tömbjére mutató pointerrel tér vissza
pv5cp g();

// az f olyan függvény, amely karakterre mutató pointerek
// 5-elemű tömbjére mutató pointerrel tér vissza
char *(*f())[5] ;

// A deklarációk ekvivalenciájának ellenőrzése
fpv5pc *pf=f;
fpv5pc *pg=g;
```

9. Névterek és tárolási osztályok

Ahhoz, hogy igazán megértsük a C++ programok működését, fontos ismernünk azokat a szabályokat, amelyek meghatározzák, hogy a programon belül miként lehet használni a különböző változókat és függvényeket. Ahhoz, hogy a C++ fordító korrekt kapcsolatot tudjon kialakítani az azonosítók és a tárolók között, szükséges, hogy minden azonosító rendelkezzen legalább két jellemzővel - típussal és tárolási osztállyal.

A *típus*, amellyel a 4. fejezetben részletesen foglalkoztunk, kijelöli a változó számára lefoglalt memória-terület méretét, illetve a benne tárolt adat értelmezését.

A *tárolási osztály* egyrészt meghatározza, hogy a változó hol jön létre a memóriában (regiszterben, statikus vagy dinamikus tárterületen), másrészt pedig definiálja a változó élettartamát. A tárolási osztályt (**auto**, **register**, **static**, **extern**) megadhatjuk a változó-definíciókban, illetve ha onnan hiányzik, akkor maga a fordító határozza meg azt, a definíció a programszövegben való elhelyezkedése alapján. Mielőtt rátérnénk a tárolási osztályok részletes tárgyalására, szükséges tisztáznunk néhány fogalom jelentését:

- élettartam (*lifetime*),
- láthatóság (*visibility*),
- érvényességi tartomány (hatókör, *scope*),
- kapcsolódás (*linkage*),
- névterület (*namespace*).

9.1. Az azonosítók élettartama

Az élettartam (*lifetime*) a program végrehajtásának olyan időszaka, amelyben az adott változó vagy függvény létezik. Az élettartam és a tárolási osztály szoros kapcsolatban állnak. Az élettartam szempontjából az azonosítókat három csoportra oszthatjuk: globális (statikus), lokális (automatikus) és dinamikus élettartamú nevek.

Statikus (globális) élettartam
Azok az azonosítók, amelyeket a **static** vagy **extern** tárolási osztállyal rendelkeznek, statikus élettartamúak. Például minden függvény és minden külső (a függvényekkel azonos) szinten definiált azonosító globális élettartamú.

A statikus élettartamú (globális) azonosító számára kijelölt memória-terület (és a benne tárolt adatok) a program futásának teljes időtartama alatt megmarad. A globális változók inicializálása egyetlen egyszer - a program indításakor - megy végbe.

Felhívjuk a figyelmet arra, hogy az azonosító élettartama és elérhetősége nem azonos fogalmak. Vannak esetek, amikor a program végrehajtásának teljes ideje alatt létező globális azonosító nem érhető el a program tetszőleges pontjáról (ilyenek például a statikus lokális vagy a statikus globális változók.) Az azonosítók elérhetőségét az élettartam és a láthatóság együtt határozzák meg.

Automatikus (lokális) élettartam
A függvényen (blokkon) belül a **static** tárolási osztály nélkül definiált azonosítók automatikus élettartammal rendelkeznek. Szintén automatikus élettartammal rendelkeznek a függvényen belül deklarált, kapcsolódás nélküli azonosítók és a függvények paraméterei.

Az automatikus élettartamú (lokális) azonosító memória-területe (és a benne tárolt adatok) csak abban a blokkban létezik, amelyben az azonosítót definiáltuk. A lokális azonosítóhoz a blokkba történő minden egyes belépéskor új terület kerül lefoglalásra, ami blokkból való kilépés során megszűnik (tartalma elvész). Következésképpen, ha a lokális változót kezdőértékkel látjuk el, akkor az inicializálás mindig újra megtörténik, amikor a változó létrejön.

Dinamikus élettartam
Dinamikus élettartammal rendelkeznek azok a memória-területek, amelyeket a **new** operátorral lefoglalunk, illetve a **delete** operátorral felszabadítunk.

9.2. Érvényességi tartomány és a láthatóság

A tárolási osztállyal ugyancsak szoros kapcsolatban álló fogalom az azonosítók láthatósága (*visibility*), illetve érvényességi tartománya (hatóköre, *scope*). Az azonosító csak a hatókörén belül látható. Az érvényességi tartomány a program azon részét jelöli ki, amelynek határain belül az adott azonosítót felhasználhatjuk a tároló elérésére. Az érvényességi tartományok több fajtáját különböztetjük meg: blokkszintű (lokális), fájlszintű (globális), függvény-, prototípus- vagy osztályszintű.

- A blokkszintű érvényességi tartománnyal rendelkező azonosító csak abban a blokkban látható, amelyikben deklaráltuk. Amikor a program eléri a blokkot záró '}' zárójelet, az azonosító többé nem lesz elérhető.
- A fájlszintű érvényességi tartománnyal rendelkező azonosító abban a fordítási egységben látható, amely a deklarációját tartalmazza. Csak azok az azonosítók rendelkeznek fájlszintű érvényességi tartománnyal, amelyeket globálisan, vagyis minden függvényen kívül deklarálunk. Amennyiben valamely függvényből olyan globális változót kívánunk használni, amelynek definícióját egy másik állomány tartalmazza, akkor az **extern** tárolási osztály felhasználásával kell az azonosítót deklarálni (külső azonosító).
- Az egyetlen függvény szintű érvényességi tartománnyal rendelkező C++ nyelvi egység az utasításcímke, amely csak a függvényen belül látható.

- Azok a paraméterazonosítók, amelyeket a függvények prototípusában megadunk (használatuk nem kötelező), csak a prototípust lezáró pontosvesszőig - a prototípusszintű hatókörben - láthatók.
- Az osztályokon belül definiált **private** és **protected** adattagok, illetve tagfüggvények csak az osztály más tagfüggvényeiből (valamint a „barát" függvényekből) érhetők el.

9.3. A kapcsolódás

Az érvényességi tartomány fogalma hasonló a kapcsolódás (*linkage*) fogalmához, azonban nem teljesen egyezik meg azzal. Azonos nevekkel különböző érvényességi tartományokban különböző azonosítókat jelölhetünk. Azonban a különböző érvényességi tartományban deklarált, illetve az azonos érvényességi tartományban egynél többször deklarált azonosítók a kapcsolódás mechanizmusának felhasználásával ugyanarra a változóra vagy függvényre hivatkozhatnak. A kapcsolódás kijelöli azt a programrészt, amelyben az adott azonosítóra hivatkozhatunk (láthatóság). Háromféle kapcsolódást különböztetünk meg: belső, külső, és amikor nincs kapcsolódás.

- A belső kapcsolódású azonosítók csak egyetlen fordítási egységen (modulban) belül ismertek. Ha a fájlszintű érvényességi tartománnyal rendelkező változó- vagy függvényazonosítók deklarációja tartalmazza a **static** kulcsszót, akkor belső kapcsolódású azonosító jön létre, amely más fordítási egységből nem érhető el. (Ellenkező esetben külső kapcsolódással rendelkeznek a globális azonosítók.) A C++ szabvány a **static** kulcsszó ilyen módon történő felhasználását *elavultnak* nyilvánította, helyette a *névtelen névterületek* használatát javasolja belső kapcsolódású azonosítók kialakítására.
- A külső kapcsolódású azonosítók több fordítási egységben (modulban) is ismertek. Azok a globális változó- vagy függvényazonosítók, amelyek deklarációjában nem szerepel tárolási osztály, vagy az **extern** tárolási osztály szerepel, külső kapcsolódásúak.
- A kapcsolódás nélküli azonosítók - valamely függvény vagy blokk helyi (lokális) azonosítói - nem rendelkeznek állandó memória-területtel. C++ nyelven az alábbi azonosítók nem rendelkeznek kapcsolódással:
    - minden olyan azonosító, amely nem jelöl változót vagy függvényt (például az **enum** konstansok, a címkék stb.),
    - a függvényparaméterek,
    - olyan blokkszintű hatókörbeli változó-azonosítók, melyek deklarációjában nem szerepel az **extern** kulcsszó.

9.4. Névterületek

A fordító a programban használt neveket (azok felhasználási módjától függően) különböző területeken (névterület - *namespace*) tárolja. Valamely névterületen belül tárolt neveknek egyedieknek kell lenniük, azonban a különböző névterületeken azonos nevek is szerepelhetnek. Két azonos név, amelyek azonos érvényességi tartományban helyezkednek el, de nincsenek azonos névterületen, különböző azonosítókat jelölnek. A C++ fordító az alábbi névterületeket különbözteti meg:

Utasításcímkék:
A névvel ellátott utasításcímke, amelyet mindig kettőspont ':' követ, része az utasításnak. Nem szükséges, hogy a különböző függvényekben használt címkék eltérőek legyenek.

Struktúrák, osztályok és uniók tagjai
Az adattagok és a tagfüggvények nevei az adott felhasználói típushoz tartozó névterületen helyezkednek el. Ezért valamely tag nevét egyidejűleg több felhasználói típusban is felhasználhatjuk. A tagok elérése a pont (.) vagy a nyíl (->) operátor megadásával lehetséges.

Közönséges nevek
Minden más név - a változók, a függvények, a paraméterek, a lokális változók és az **enum** konstansok nevei - közös névterületen tárolódnak. A változó-azonosítók egymásba ágyazott láthatósággal rendelkeznek, ami azt jelenti, hogy egy új blokkban újradefiniálhatók.

Típusnevek
A típusnevet azonos érvényességi tartományon belül nem lehet azonosító neveként használni.

Globális névtér
A globális névtér tartalmazza a fájlszinten definiált változóinkat és függvényeinket. A ISO/ANSI szabvány előtti C++ változatokban ugyancsak a globális névtérben helyezkedtek el a könyvtári függvények és osztályok. (A globális névterület azonosítóira a hatókör operátorral hivatkozhatunk, például *::nata*.)

9.4.1. Saját névterületek kialakítása és használata

A szabványos C++ lehetővé teszi, hogy a névterek kezelését saját kezünkbe vegyük. A névterületek kialakításával a fájlszintű láthatóságot programszintűvé terjeszthetjük ki azzal, hogy a globális névtér mellett saját, névvel ellátott névterületeket alakítunk ki.

A névterület kijelölése tetszőleges forrásállományban (*.cpp*, *.h*) megadható, a fordító az azonos néven szereplő definíciókat egyesíti. A kijelölést a **namespace** kulcsszó felhasználásával végezzük:

```
namespace nevter {

   deklarációk és definíciók

}
```

Az alábbi példában szereplő névtér különböző definíciókat és deklarációkat tartalmaz:

```
namespace Pelda {
   int a=12;                          // változó-definíció
   extern double d;                   // változó-deklaráció
   double sqr(double x) {             // függvény-definíció
     return x*x;
   }
   int mod(int a, int b);             // függvény-deklaráció
}
```

Amennyiben a névterületen belül deklarációk szerepelnek, az azokhoz tartozó definíciókat a névtéren kívül is megadhatjuk, használva a hatókör operátort:

```
int Pelda::mod(int a, int b)
{
   return a%b;
}

double Pelda::d=100;
```

A névterületen megadott azonosítók minden olyan modulból elérhetők, amelybe beépítjük a névtér deklarációkat tartalmazó változatát (például fejállományból), esetünkben:

```
namespace Pelda{
   extern int a;
   extern double d;
   double sqr(const double x);
   int mod(int a, int b);
}
```

Ezek után a hatókör (*::*) operátor segítségével egyenként elérhetjük a neveket:

```
Pelda::d = Pelda::sqr(13.26);
Pelda::a=7430;
int x = Pelda::mod(Pelda::a, 12);
```

Természetesen sokkal kényelmesebb megoldáshoz jutunk, ha az összes nevet elérhetővé tesszük a magunk számára a **using namespace** direktívával:

```
using namespace Pelda;
d = sqr(13.26);
a=7430;
int x = mod(a, 12);
```

A **using** direktíva felhasználásával a névtér elemei közül csak a számunkra szükségeseket érjük el:

```
using Pelda::d;
using Pelda::sqr;
d = sqr(13.26);
using Pelda::a;
a=7430;
using Pelda::mod;
int x = mod(a, 12);
```

A névterek további lehetőségeinek bemutatása messze meghaladja fejezetünk méretét. A névterületeket egymásba lehet skatulyázni, a **static** tárolási osztály (**extern** helyén való) használatát ún. névtelen névterekkel ajánlott felváltani, a névterekhez álneveket definiálhatunk stb. Az elmondottak bemutatására nézzünk egy saját könyvtár kialakítását bemutató példaprogramot!

```
// ------------------------------------------------------------
// lib.h állomány a szükséges deklarációkat teszi a névtérbe

namespace lib {
  #include <math.h>     // C függvények elérése
  #include <stdlib.h>
}

namespace lib {
  void rendez(int *v, int n, bool nov=true);
  void kiir(const int *v, int n);
}

// ------------------------------------------------------------
// lib.cpp fájl a könytár elemeit deinifálja

#include <iostream>
namespace rendszer = std;    // álnév létrehozása

#include "lib.h"
using namespace lib;
```

```
// Névtelen névtér a modulszintű definíciókhoz
namespace {
   void csere(int & a, int & b){
       int sv=a;
       a=b, b=sv;
    }
    const char TAB='\t';
}

void lib::rendez(int *v, int n, bool nov) {
    for (int i=0; i<n-1; i++)
      for (int j=i+1; j<n; j++)
         if (nov==(v[j]<v[i]))
             csere(v[j], v[i]);
}

void lib::kiir(const int *v, int n) {
   for (int i=0; i<n; i++)
     rendszer::cout<<v[i]<<TAB;
   rendszer::cout<<rendszer::endl;
}

// ------------------------------------------------------------
// main.cpp a könyvtár használatát bemutató példaprogram
#include <iostream>
#include <ctime>

#include "lib.h"
using namespace lib;

void main()
{
   int a[6];
   srand(unsigned(std::time(NULL)));
   for (int i=0; i<6; i++)
      a[i]=pow(rand()%10,2);

   kiir(a,6);
   rendez(a,6);          // növekvő sorrend
   kiir(a,6);
   rendez(a,6, false);   // csökkenő sorrend
   kiir(a,6);
   std::cin.get();
}
```

9.5. A tárolási osztályok használata

A tárolási osztály megadásával lehetőségünk van az alapértelmezéstől eltérő élettartam és érvényességi tartomány (hatókör) kialakítására. Azok az azonosítók, amelyek tárolási osztálya **auto** vagy **register**, lokális élettartammal, míg a **static**, illetve **extern** azonosítók globális élettartammal rendelkeznek. A **typedef** szintén tárolási osztályt jelöl a C++ nyelvben, azonban csak formai szempontból soroljuk ebbe a csoportba.

A változó- és a függvény-deklarációk forrásállományon belüli elhelyezkedése szintén hatással van a tárolási osztályra és a láthatóságra. Azok a deklarációk, amelyek minden függvényen kívül helyezkednek el, a „külső szintű" deklarációk, míg a függvényen belül megadott deklarációk - „belső szintűek" . A tárolási osztály azonosítójának pontos jelentése függ attól, hogy a deklaráció a külső vagy a belső szinten szerepel, illetve attól is, hogy változót vagy függvényt deklarálunk. Az alábbi táblázatban összefoglaltuk az azonosítók élettartamára és láthatóságára vonatkozó megállapításokat:

Jellemzők			**Eredmény**	
Szint	***Tétel***	***Tárolási osztály***	***Élettartam***	***Láthatóság***
fájlszintű hatókör	változó-definíció	**static**	globális	korlátozva az adott állományra, a definíció helyétől a fájl végéig
	változó-deklaráció	**extern**	globális	a definíció helyétől a fájl végéig
	Prototípus vagy függvénydefiníció	**static**	globális	korlátozva az adott fájlra
	Prototípus	**extern**	globális	a definíció helyétől a fájl végéig
blokk-szintű hatókör	változó-deklaráció	**extern**	globális	blokk
	változó-definíció	**static**	globális	blokk
	változó-definíció	**auto** vagy **register**	lokális	blokk

Minden tárolási osztály esetén tisztáznunk kell a deklarált változó vagy függvény élettartamát és láthatóságát, illetve változók esetén az inicializálás kérdését is.

9.5.1. Az auto tárolási osztály

Azok a változók, amelyeket blokkon belül definiálunk alapértelmezés szerint automatikus (**auto**) változók. Az automatikus változók a függvények belső változói, amelyek akkor kezdenek el létezni, amikor a függvényt meghívjuk. A függvényből való kilépés után pedig megszűnnek. (A függvény paramétereit is hasonló módon kezeli a rendszer.)

Mint ismert, az összetett utasítások (a blokkok) egymásba ágyazhatók. Mindegyik blokk egyaránt tartalmazhat definíciókat (deklarációkat) és utasításokat:

```
{
  Definíciók, deklarációk és
  utasítások
}
```

Valamely automatikus azonosító ideiglenesen láthatatlanná válik, ha egy belső blokkban ugyanolyan névvel egy másik változót definiálunk. Ez az elfedés a belső blokk végéig terjed, és nincs mód arra, hogy az elfedett (létező) objektumra hivatkozzunk:

```
void main()
{
   int sum = 10;
   // az int típusú sum értéke 10 */
   {
      float sum=3.1416;
      // a float típusú sum értéke 3.1416 */
   }
   // az int típusú sum értéke 10
}
```

Mivel az automatikus tárolók a blokkból kilépve megszűnnek, ezért alapvetően hibás elgondolás olyan függvényt írni, amely egy automatikus változó címével (vagy referenciájával) tér vissza. Ha a függvényen belül kívánunk helyet foglalni olyan tároló számára, melyet a függvényen kívül használunk, akkor a dinamikus memóriafoglalás eszközeit kell alkalmaznunk. Az alábbi két függvény közül a *fordit1()* hibás (az elmondottak értelmében), míg a *fordit2()* a helyes megoldást tartalmazza. A függvények a kapott karaktersorozatot megfordítják, és egy másik területen adják vissza:

```
// Hibás megoldás!
char * fordit1(const char * s) {
   char a[80];     // az a tömb automatikus változó!
   int i = strlen(s);
   a[i]=0;
   while (*s)
         a[--i] = *s++;
   return a;
}
```

```
// Helyes megoldás
char * fordit2(const char * s) {
   // az a mutató ugyan automatikus, de az általa kijelölt
   // terület dinamikus helyfoglalású!
   char *a = new (nothrow) char[strlen(s)+1];
   int i = strlen(s);
   if (!a)
      return NULL; // ha nincs hely
   a[i]=0;
   while (*s)
         a[--i] = *s++;
   return a;
}
```

Az **auto** változók inicializálása minden esetben végbemegy, amikor a vezérlés a blokkhoz kerül. Azonban csak azok a változók kapnak kezdőértéket, amelyek definíciójában szerepel kezdőértékadás. (A többi változó értéke határozatlan!). Mivel az automatikus változók esetén az inicializáló kifejezés kiértékelése futási időben történik, ezért tetszőleges kifejezés megadható (például függvényhívás is):

```
int fv(void) {
   double pi    = asin(1)*2;
   int    lepes = 20;
   double lrad  = 2*pi/lepes;
   int    a;
   int  * pa = &a;
}
```

Meg kell jegyeznünk, hogy az ANSI C++ az automatikus tömb- és struktúratípusú változók tetszőleges kifejezéssel történő inicializálását is lehetővé teszi:

```
double b[3]={sin(12), cos(12), sqrt(12)};
```

9.5.2. Az extern tárolási osztály

A C++ program általában egy sor külső azonosítót használ. A *külső* kifejezést a függvények paramétereit és automatikus változóit jellemző *belső* kifejezéssel ellentétes értelemben használjuk. C++ nyelv külső azonosítói a függvényeken kívül definiált változók és függvények nevei. Azok a külső változók és függvények, amelyek definíciójában nem adunk meg tárolási osztályt, alapértelmezés szerint **extern** tárolási osztállyal rendelkeznek. (Természetesen az **extern** közvetlenül is megadható.)

Az **extern** változók és függvények élettartama a programba való belépéstől a program befejezésig terjed. Azonban a láthatósággal lehetnek problémák. Egy adott modulban definiált függvény csak akkor érhető el egy másik modulból, ha abban szerepeltetjük a függvény prototípusát (például deklarációs állomány beépítésével).

Bonyolultabb a helyzet a változók esetén. Ha az egyik modulban egy változót tárolási osztály nélkül szerepeltetünk a függvényeken kívül, akkor ez a változó definícióját jelenti (függetlenül attól, hogy adunk-e kezdőértéket vagy sem). A változó másik modulból való eléréséhez meg kell adnunk a változó deklarációját az **extern** tárolási osztályt követően. Az elmondottakat jól szemlélteti az alábbi példa:

```
// modul1.cpp
// prototípus - deklaráció
double KorKerulet(double r);

// definíció
double KorTerulet1(double r)
{
   return r*KorKerulet(r)/2;
}

// deklaráció
extern double pi;

// definíció
double KorTerulet2(double r)
{
   return r*r*pi;
}
```

```
// modul2.cpp
// definíció
double pi=asin(1)*2;

// definíció
double KorKerulet(double r)
{
  return 2*r*pi;
}
```

A szabványos C++ nyelv (a C-vel ellentétben) lehetővé teszi, hogy a statikus élettartamú változóknak futásidejű kifejezést adjunk kezdőértékként.

9.5.3. A static tárolási osztály

A **static** tárolási osztály mind külső, mind pedig belső szintű azonosítókkal együtt használható. Ha külső szintű azonosítók előtt szerepel, akkor az azonosítók láthatóságát a forrásállományra korlátozza. Ha pedig belső szintű nevek előtt adjuk meg, a nevek élettartamát automatikusról globálisra módosítja.

Amikor több modulból álló programot fejlesztünk, a moduláris programozás elvének megvalósításához nem elegendő, hogy vannak közös változóink. Szükségünk lehet olyan modul szinten definiált változókra és függvényekre is, amelyek elérését a modulra korlátozhatjuk (információrejtés). Ezért **extern** és **static** tárolási osztályú azonosítókat egyaránt használunk a megfelelő modulszerkezet kialakításához. A C++ szabvány a statikus külső váltózók helyett a névtelen névterület használatát javasolja.

Példaként készítsünk olyan modult, amely pszeudovéletlen szám előállítására használható! Több modulból álló program esetén feltétlenül szükség van arra, hogy az egyes modulok elején információkat helyezzünk el a fájlok tartalmáról. A modulban definiált **extern** függvények deklarációját a *random.h* deklarációs állomány tartalmazza.

```
// random.h
// Pszeudovéletlen számok sorozatának előállítására
// szolgáló modul globális deklarációi.
#ifndef randomH
#define randomH
namespace nsrandom
{
  void set_random(unsigned short int);
  unsigned short int random(void);
}
#endif
```

A *random.cpp* állomány két kívülről is elérhető függvényt és egy modulszintű változót tartalmaz:

```
// random.cpp
// Pszeudovéletlen számok sorozatának előállítása.
//    set_random - a kiindulási érték beállítása,
//    random     - a következő véletlen szám lekérdezése.
#include "random.h"
using namespace nsrandom;

// modulszintű változók és konstansok
// definiálása (static helyett!)
namespace {
   const short SZORZO = 97;
   const int OSZTO = 0x10000;
   const short NOVELES = 59;
   unsigned short int pseudo=0;
}

// A kiindulási érték beállítása
void nsrandom::set_random(unsigned short int init) {
  pseudo = init;
}

// A következő véletlen szám előállítása
unsigned short int nsrandom::random(void) {
  pseudo = (SZORZO * pseudo + NOVELES) % OSZTO;
  return pseudo;
}
```

Végezetül tekintsük a *randmain.cpp* programot, amelyben kockadobás szimulációjához használjuk a fenti *random()* függvényt!

```
// randmain.cpp
// A hatlapú kocka dobásának szimulációja
// - a példában a kockát 5-ször dobjuk
#include <iostream>
using namespace std;
#include "random.h"
using namespace nsrandom;
```

```
void main() {
  set_random(2003);     // a generátor inicializálása
  cout<<"Otszor dobunk a kockaval: "<<endl;
  for (int i=0; i<5; i++)
   cout<<i<<". dobas \t"<<random()%6+1<<endl;
  cin.get();
}
```

A **static** tárolási osztály másik, a C++ szabvány által is támogatott alkalmazási területe a statikus belső szintű változók definiálása. Az ilyen változók láthatósága a definíciót tartalmazó blokkra korlátozódik, azonban a változó a program indításától a programból való kilépésig él. A statikus változók inicializálása szintén egyetlen egyszer, a változó létrehozásakor megy végbe. Ezért ezek a statikus lokális változók a blokkból való kilépés után (a függvényhívások között) is megőrzik értéküket.

Az előző példánk véletlenszám-generátorát egyetlen függvény felhasználásával is megvalósíthatjuk:

```
unsigned short int srandom(unsigned short int init=0)
{
  static const short SZORZO = 97;
  static const int OSZTO = 0x10000;
  static const short NOVELES = 59;
  // statikus lokális változó definiálása 0 kezdőértékkel
  static unsigned int pseudo=0;
  if (init)  // Ha az init nem 0
     pseudo = init;
  else
     pseudo = (SZORZO * pseudo + NOVELES) % OSZTO;
  return pseudo;
}
```

Az *srandom()* függvényt kétféleképpen kell hívni. Ha 0-tól különböző argumentummal hívjuk, akkor a véletlenszám-sorozat kezdőértékét állítjuk be. A következő véletlen szám lekérdezéséhez 0 értékű argumentummal, vagy argumentum nélkül kell az *srandom()* függvényt aktivizálnunk.

A statikus változók kezdőértékkel való ellátására ugyanazok a szabályok vonatkoznak, mint az **extern** változókéra. A **static** változók inicializálása a programba való belépés során egyszer megy végbe. Amennyiben nem adunk meg kezdőértéket a definícióban, úgy a fordító automatikusan 0-val inicializálja a változót (feltöltve annak területét nullás bájtokkal). C++-ban a statikus változók kezdőértékeként szintén tetszőleges kifejezést használhatunk.

9.5.4. A register tárolási osztály

A **register** tárolási osztály csak belső szintű azonosítókhoz (automatikus lokális változókhoz és paraméterekhez) használható. A **register** kulcsszó megadásával közöljük a fordítóval, hogy az adott változót gyors eléréssel szeretnénk kezelni. Ennek érdekében a fordító (amennyiben módjában áll kérésünket teljesíteni) a processzor regiszterében hozza létre a változót. Ha nincs szabad felhasználható regiszter, akkor a tároló **auto** változóként jön létre.

A fenti megoldásból következik, hogy nincs mód annak lekérdezésére, hogy a tárolást végül is hol valósította meg a fordító (memóriában vagy regiszterben). A regiszterek adattárolási kapacitása processzoronként eltérő, ezért implementációnként különböznek azok a típusok, amelyeket regiszterben tárolhatunk. A legtöbb C++ fordító az egész jellegű, a mutató- és a referenciatípusokat tárolja regiszterben.

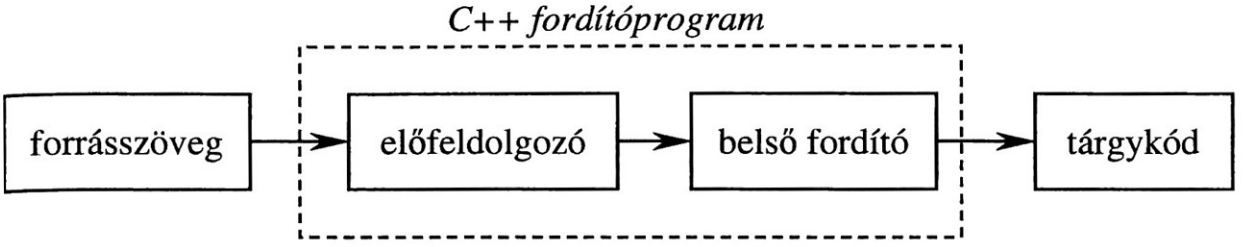
Az alábbi *csere* függvény a korábban bemutatott megoldásnál potenciálisan gyorsabb működésű. (A potenciális szót azért használjuk, mivel nem ismert, hogy a **register** előírások közül hányat teljesít a fordító.)

```
void csere(register int &a, register int &b) {
   register int c = a;
                a = b;
                b = c;
}
```

A **register** változók élettartama, láthatósága és inicializálásának módja megegyezik az **auto** tárolási osztályú változókéval.

10. Az előfeldolgozó (preprocesszor)

Minden C++ fordítóprogram szerves részét képezi az ún. előfeldolgozó (*preprocesszor*). A fordítóprogram gyakorlatilag nem azt a forráskódot fordítja, amit mi begépelünk, hanem az előfeldolgozó által előállított szöveget dolgozza fel.



A C++ fordítók többségében ez a két jól elkülöníthető fordítási szakasz nem válik külön, azaz nem jelenik meg az előfordító kimenete valamilyen szöveges állományban. A fentiekben bemutatott működés egyben az előfordító használatának előnyét és hátrányát is jelenti. A fejezetben ismertetésre kerülő megoldások a C++ program olvasható formában történő előállítását támogatják. A hátrány pedig abban áll, hogy a programozó általában nem látja azt az előfordított kódot, amiből a futó program előáll. Ebből következik, hogy bizonyos előfordításból származó programhibák kiderítése nem egyszerű feladat.

Nem kell tehát csodálkoznunk azon, hogy a szabványos C++ nyelv tartalmaz olyan eszközöket, amelyek segítségével az előfordító bizonyos elemeit igazi C++ programelemekkel helyettesíthetjük. Nevezetesen, a ***#define*** konstansok helyett **const** konstansokat használhatunk, a ***#define*** makrók helyett pedig **inline** függvénysablonokat készíthetünk.

Az előfeldolgozó olyan sororientált szövegfeldolgozó (makrónyelv), amely semmit sem „tud" a C++ nyelvről. Ez két fontos következménnyel jár:

- az előfeldolgozónak szóló utasításokat nem írhatjuk olyan kötetlen formában, mint az egyéb C++ utasításokat (tehát egy sorba csak egy utasítás kerülhet, és a parancsok nem csúszhatnak át másik sorba, hacsak nem jelölünk ki folytatósort),
- az előfeldolgozó által elvégzett minden művelet egyszerű szövegkezelés (függetlenül attól, hogy a C++ nyelv kulcsszavai, kifejezései vagy változói szerepelnek benne).

A preprocesszornak szóló parancsokat a sor elején (esetleg szóközök és/vagy tabulátorok után) álló # karakter jelzi. Az előfeldolgozót leggyakrabban szöveghelyettesítésre (***#define***) és szöveges fájl beépítésére (***#include***) használjuk. Ugyancsak jól alkalmazható a program részeinek feltételtől függő fordítására is (***#if***). A preprocesszor-parancsokat szokás direktíváknak is nevezni.

10.1. Állományok beépítése a forrásprogramba

Az ***#include*** direktíva utasítja az előfeldolgozót, hogy az utasításban szereplő szöveges állomány tartalmát építse be a programunkba. (A beépítés helyét a direktíva elhelyezkedése határozza meg.) Általában a deklarációkat és makrókat tartalmazó ún. fejállományokat (*header file*-okat) szoktuk beépíteni a programunk elején, azonban tetszőleges szöveges állományra használható a művelet. A beépítési utasítás általános alakja:

```
#include <fájlnév>
```

illetve

```
#include "fájlnév"
```

ahol a *fájlnév* a beépítendő állomány neve. Az első formát általában a C++ rendszer szabványos *fejállományainak* (*iostream*, *cmath*, *ctime*, *cstdlib* stb.) beépítésére használjuk. A második pedig a saját magunk készített fájlok beépítésére szolgál.

Mindig problémát jelent annak eldöntése, hogy mi is szerepeljen a beépített állományokban. Az alábbi táblázat segíthet e kérdés megválaszolásában:

Mi lehet a fejállományokban?	***Példa***
Megjegyzések	`/* megjegyzés */`
Feltételes fordítási direktívák	`#ifdef DEBUG`
Makró-definíciók	`#define UNIX`
Include direktívák	`#include <iostream>`
Felsorolások	`enum Valasz {nem, igen, talan};`
Konstansdefiníciók	`const double pi=3.1415265;`
Névvel ellátott névterületek	`namespace nsrandom { /* ... */ }`
Névdeklarációk	`class Vektor;`
Típusdefiníciók	`struct Complex { double re, im; };`
Változó-deklarációk	`extern int x[];`
Függvény-deklarációk (prototípusok)	`void csere(int &, int &);`
Inline függvény-definíciók	`inline int sqr(int a) {return a*a;}`
Template deklarációk	`template <class Tipus> class Vekor;`
Template definíciók	`template<class Tipus> class Vekor {/*...*/};`

Néhány olyan elemet külön is megemlítünk, amit nem szabad *include* fájlba helyezni:

- nem *inline* függvények definícióját,
- változó-definíciót,
- névtelen névtér definícióját.

10.2. Feltételes fordítás

A feltételes fordítás lehetőségeinek használatával elérhető, hogy a forrásprogram bizonyos részei csak adott feltételek teljesülése esetén kerüljenek be az előfeldolgozó által előállított programba. A feltételesen fordítandó programrészek kijelölésére többféle szerkezet közül választhatunk. A különböző megoldásokat az ***#if***, ***#ifdef***, ***#ifndef***, ***#elif***, ***#else*** és ***#endif*** preprocesszor-direktívákkal állíthatjuk elő. Az ***#if*** utasításban a feltétel megadására konstans kifejezéseket használhatunk, melyek 0 és nem nulla értéke jelöli az igaz, illetve a hamis feltételt, például:

```
#if 'z' - 'a' == 25
```

A feltételek másik fajtájával azt ellenőrizhetjük, hogy a megadott szimbólum definiált-e vagy sem. Ehhez a feltételben a ***defined*** operátort használjuk, mely 1 értékkel tér vissza, ha az operandusa létező szimbólum:

```
#if defined szimbolum
```

vagy

```
#if defined(szimbolum)
```

Mivel gyakran használunk ehhez hasonló vizsgálatokat, ezért valamely szimbólum létezésének tesztelésre külön előfeldolgozó utasítás áll a rendelkezésünkre:

```
#ifdef szimbolum
```

Az ***#if*** segítségével azonban bonyolultabb feltételek is megfogalmazhatók:

```
#if defined szimbolum && ('z' - 'a' == 25)
```

Nézzük először az egyszerűbb szerkezetet, amellyel két programrész közül választhatunk:

```
#if konstans kifejezés
      programrész1
#else
     programrész2
#endif
```

Ez a szerkezet jól használható ún. hibakeresési (*debug*) információk beépítésére a programba. Ekkor egyetlen szimbolikus konstans 1 vagy 0 értékével jelölhetjük ki a lefordítandó programrészeket. Az alábbi példában szereplő osztást végző függvény a program fejlesztési szakaszában (***#define*** *DEBUG 1*) kiírja a paraméterek értékét, és 0-val való osztás esetén hagyja hibajelzéssel kilépni a programot. Az „éles" verzióban (***#define*** *DEBUG 0*) azonban valahogy kivédi a 0-val való osztást. A DEBUG szimbólumot a program elején látjuk el a megfelelő értékkel:

```
#define DEBUG 1

double osztas(int a, int b){
  #if DEBUG
      cout<<"osztas - a ="<<a<<endl;
      cout<<"osztas - b ="<<b<<endl;
  #else
      if (b == 0) {
          a = MAXINT;
          b = 1;
      }
  #endif
  return (double)a/b;
}
```

A fenti szerkezet helyett jobb a *DEBUG* szimbólum definiáltságát vizsgáló megoldást használni, melyhez a *DEBUG* érték nélkül is definiálható:

```
#define DEBUG
. . .
#if defined(DEBUG)
      cout<<"osztas - a ="<<a<<endl;
      cout<<"osztas - b ="<<b<<endl;
  #else
      if (b == 0){
          a = MAXINT;
          b = 1;
      }
  #endif
```

Az alábbi két-két vizsgálat ugyanazt az eredményt szolgáltatja:

```
#if defined(DEBUG)
. . .
#endif
```

```
#ifdef DEBUG
. . .
#endif
```

illetve a fordított megoldás:

```
#if !defined(DEBUG)
. . .
#endif
```

```
#ifndef DEBUG
. . .
#endif
```

Bonyolultabb struktúrák kialakításához egy másik preprocesszor-utasítást ajánlott használni, ami többirányú elágaztatás megvalósítását teszi lehetővé:

```
#if konstans_kifejezés1
      programrész1
#elif konstans_kifejezés2
      programrész2
#elif konstans_kifejezés3
      programrész3
#else
     programrész4
#endif
```

Az ***#include*** utasítások használata során előfordul, hogy ugyanazt az állományt többször építjük be a programunkba, ami programhibát okozhat. Nézzük meg, hogyan oldhatjuk meg ezt a problémát! A következő megoldást minden makrókat definiáló fejállományban ajánlott alkalmazni. A deklarációs fájl első beépítésekor létrejön egy szimbólum (például *fajlnevH*), melynek létezését vizsgálva elkerülhető az ismételt beépítés:

```
#ifndef fajlnevH
#define fajlnevH

   makró-definíciók és egyéb deklarációk

#endif
```

10.3. Makrók használata

A ***#define*** direktívát arra használjuk, hogy „beszédes" azonosítókkal lássunk el C++ konstansokat, kulcsszavakat, illetve gyakran használt utasításokat és kifejezéseket. Az így definiált konstansokat reprezentáló makrókat szimbolikus konstansnak nevezzük. Általában a kifejezéseket és utasításokat megvalósító definíciókat hívjuk makrónak (függvényszerű makró). A makrónevekre ugyanaz a képzési szabály vonatkozik, mint más azonosítókra. Azért, hogy a preprocesszor számára definiált szimbólumok a C++ forrásnyelvi szövegben jól elkülönüljenek a programban használt azonosítóktól, a makróneveket csupa nagybetűvel ajánlott írni.

Az előfeldolgozó minden programsort átvizsgál, hogy az tartalmaz-e valamilyen korábban definiált makrónevet. Ha igen, akkor azt lecseréli a megfelelő helyettesítő szövegre, majd újból átvizsgálja a sort további makrókat keresve, amit új helyettesítés követhet. Mindaddig folytatódik ez a folyamat, amíg vagy nem talál a preprocesszor újabb makrónevet a sorban, vagy csak olyat talál, amit már egyszer helyettesített (a végtelen rekurziók elkerülése). Ezt a folyamatot makróhelyettesítésnek, vagy makrókifejtésnek nevezzük.

A makrókat a ***#define*** utasítással hozzuk létre, melynek két formája használható. Ha a makróra többé nincs szükségünk, akkor az ***#undef*** direktíva segítségével megsemmisítjük azt.

10.3.1. Szimbolikus konstans makrók készítése

Ebben az esetben a ***#define*** direktíva egyszerűbb alakját használjuk:

```
#define  azonosító  helyettesítő szöveg
```

Nézzünk néhány példát a szimbolikus konstansok definiálására!

```
#define EOS         '\0'
#define TRUE 1
#define FALSE0
#define NOT  !
#define BOOL int
#define IGEN 1
#define NEM  !IGEN
#define URES
#define STR  ""
#define UDV  "Üdvözöllek dicső lovag ...!"
#define PMERET       1024
#define HA   if
#define KIIR cout
```

A fenti makrókat használva eléggé furcsa kinézetű (működő) C++ programot írhatunk. A kifejtés menetének érzékeltetés érdekében nézzük meg a

```
HA (NEM)
```

sor feldolgozásának lépéseit!

```
if (NEM)      →   if (!IGEN)      →   if (!1)
```

10.3.2. Függvényszerű makrók készítése

A makrók felhasználási lehetőségeit lényegesen megnövelik a paraméterezett makrók, melyek definíciójának általános formája:

```
#define  azonosító(paraméterlista)  helyettesítő szöveg
```

A makró hívása:

```
azonosító(argumentumlista)
```

A makróhívásban szereplő argumentumok számának meg kell egyeznie a definícióban szereplő paraméterek számával. (Lehet paraméterek nélküli makrót is készíteni.) Az alábbi példaprogramban bemutatjuk néhány gyakran használt makró definícióját:

```
// x abszolút értékének meghatározása
#define abs(x)    ( (x) < 0 ? (-(x)) : (x) )

// a és b maximumának kiszámítása
#define max(a,b)  ( (a) > (b) ? (a) : (b) )

// A és B minimumának kiszámítása
#define min(A,B)  ( (A) < (B) ? (A) : (B) )

// Négyzetre emelés
#define sqr(x)    ( (x) * (x) )

// Két egész szám tartalmának felcserélése
// (csak egész balérték argumentummal működik helyesen)
#define csere(a,b) { int c; c = a, a=b, b=c; }
```

A makrókra jellemző, hogy általában tetszőleges típusú argumentummal meghívhatók. (A fenti *csere()* makró kivételnek számít.) A makró törzse az argumentumok aktuális szövegértékének behelyettesítése után bemásolódik a hivatkozás helyére. Nézzük meg közelebbről az *sqr(a)* hivatkozást, melyet az előfordító az alábbi kifejezéssel helyettesít!

```
( (a) * (a) )
```

A makró törzsében a paramétereket általában zárójelben kell használnunk, ellenkező esetben bizonyos argumentumokkal aktivizálva hibás eredményt kapunk. Példaként nézzük meg az *sqr()* makrót úgy is, hogy ne legyenek zárójelben a paraméterek:

```
#define sqr2(x)    ( x * x )                    // hibás!
```

Hasonlítsuk össze az *sqr* és az *sqr2* hívását *a* és *a+1* argumentumokkal!

	a	*a+1*
Sqr	`( (a) * (a) )`	`( (a+1) * (a+1) )`
Sqr2	`(a * a)`	~~`(a+1 * a+1)`~~

A zárójelezésnek azonban kellemetlen következményei is lehetnek, mint például amikor valamilyen léptető operátor szerepel a makró argumentumában:

```
int a=5;
cout<<sqr(a++);        // 30
cout<<a;               //  7
```

A programrészlet lefutásakor a kiírt értékek *30* és *7*. A hiba a kifejtett makró kiértékelésében keresendő, amely mellékhatásokat tartalmaz:

```
(a++) * (a++)
```

A mellékhatás miatt a kifejezés kiértékelése implementációfüggő. Sajnos általánosan is elmondható, hogy nem szabad léptető operátort tartalmazó kifejezést makrónak átadni. A fenti programrészlet kétféle módon javítható. Az egyszerűbb első megoldást akkor kapjuk, ha a léptetés műveletét külön hajtjuk végre:

```
int a=5;
cout<<sqr(a);          // 25
a++;
cout<<a;               //  6
```

A C++ nyelvben ajánlott másik megoldás, ha a makró helyett **inline** függvényt, illetve függvénysablont használunk a négyzetre emelés elvégzésére:

```
inline int square(int x) {
  return x * x;
}

template <class tip>
inline tip sqr(tip x) {
  return x * x;
}
```

Az eddigi példáinkban a helyettesítést csak különálló paraméterek esetén végzi el az előfeldolgozó. Vannak esetek, amikor a paraméter valamely azonosítónak része, vagy az argumentum értékét sztringként kívánjuk felhasználni. Ha a helyettesítést ekkor is el kívánjuk végezni, akkor a ##, illetve a # makróoperátorokat kell használnunk.

A ## operátor megadásával két szintaktikai egységet (*tokent*) lehet összeforrasztani oly módon, hogy a makró törzsében a ## operátort helyezzük a paraméterek közé.

A alábbi példában szereplő *show()* makró tetszőleges *x*-szel kezdődő nevű numerikus változó értékét ***double*** típusúvá konvertálva jeleníti meg:

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

#define show(a) cout<<(double)(x##a)

void main()
{
  double x1  = 10;
  int    xyz =20;
  show(yz);    // cout<<(double)(xyz);
  show(1);     // cout<<(double)(x1);
  cin.get();
}
```

A # karaktert a makró paramétere elé helyezve, a paraméter értéke idézőjelek között (sztringként) helyettesítődik be. Ezzel a megoldással sztringben is lehetséges a behelyettesítés, hisz a fordító az egymás mellett álló sztringliterálokat egyetlen sztringkonstanssá kapcsolja össze. Az alábbi példában a *megjelenit()* makró segítségével tetszőleges változó neve és értéke megjeleníthető:

```
#include <iostream>
using namespace std;
#define megjelenit(n) \
                cout<<#n" valtozo erteke: "<<n<<endl;
void main() {
   int     a = 13;
   char   *p = "C++ nyelv";
   double pi = 3.14259265;
   megjelenit (a);
   megjelenit (p);
   megjelenit (pi);
   cin.get();
}
```

```
a valtozo erteke: 13
p valtozo erteke: C++ nyelv
pi valtozo erteke: 3.14259
```

Ha makró által lefoglalt azonosítót meg szeretnénk szüntetni, akkor az ***#undef*** direktívát kell használnunk. A makró új tartalommal történő átdefiniálása előtt mindig meg kell szüntetnünk a régi definíciót. Az ***#undef*** utasítás nem jelez hibát, ha az azonosító nincs definiálva. Az ***#undef*** használata:

```
#undef azonosító
```

10.3.3. Előre definiált makrók

Az ANSI C++ szabvány az alábbi, előre definiált makrókat tartalmazza. (Mindegyik azonosító két-két aláhúzás karakterrel kezdődik és végződik.) Az előre definiált makrók nevét nem lehet sem a ***#define*** sem pedig az ***#undef*** utasításokban szerepeltetni. Az előre definiált makrók értékét beépíthetjük a program szövegébe, de feltételes fordítás feltételeként is felhasználhatjuk.

Makró	*Leírás*	*Példa*
__DATE__	A fordítás dátumát tartalmazó sztringliterál.	*"Dec 23 2003"*
__TIME__	A fordítás időpontját tartalmazó sztringliterál.	*"12:23:07"*
__FILE__	A forrásfájl nevét tartalmazó sztringliterál.	*"preproc.cpp"*
__LINE__	A forrásállomány aktuális sorának sorszámát tartalmazó számkonstans (1-től sorszámoz).	*23*
__STDC__	A értéke 1, ha a fordító ANSI C++ fordítóként működik, különben nem definiált.	
__cplusplus	A értéke 1, ha C++ forrásállományban teszteljük az értékét, különben nem definiált.	

10.4. A #line, az #error és a #pragma direktívák

Több olyan segédprogram is létezik, amely valamilyen speciális nyelven megírt programot C++ forrásprogrammá alakít át (programgenerátor). A ***#line*** direktíva segítségével elérhető, hogy a C++ fordítóprogram ne a C++ forrásszövegben jelezze a hiba sorszámát, hanem az eredeti speciális nyelven megírt forrásfájlban. (A ***#line*** utasítással beállított sorszám és állománynév a *__LINE__* és a *__FILE__* szimbólumok értékében is megjelennek.) A direktíva használata:

```
#line kezdősorszám
```

vagy

```
#line kezdősorszám "fájlnév"
```

Az ***#error*** direktívát a programba elhelyezve fordítási hibaüzenet jeleníthetünk meg, amely az utasításban megadott szöveget tartalmazza. Az utasítás általános formája:

```
#error hibaüzenet
```

Az alábbi példában a fordítás hibaüzenettel zárul, ha nem C++ módban dolgozunk:

```
#if !defined(__cplusplus)
    #error A fordítás csak C++ módban végezhető el!
#endif
```

A ***#pragma*** direktíva a fordítási folyamat implementációfüggő vezérlésére szolgál. (A direktívához semmilyen szabványos megoldás sem tartozik.) Ha a fordító valamilyen ismeretlen ***#pragma*** utasítást talál a programban, akkor azt figyelmen kívül hagyja. Ennek következtében a programok hordozhatóságát nem veszélyezteti ez a direktíva. Az utasítás általános alakja:

```
#pragma parancs
```

Létezik még egy üres direktíva is (**#**), amelynek semmilyen hatása sincs a fordításra:

```
#
```

II.

Az objektum-orientált C++ nyelv

11. Bevezetés az objektum-orientált C++ nyelvbe

Az objektum-orientált programozás (OOP) a gondolkozást, cselekvést közelítő programozási mód. Egy objektum-orientált programozási nyelv sokkal strukturáltabb, modulárisabb és absztraktabb, mint a hagyományos programozási nyelvek.

Ellentétben a hagyományos programozási nyelvekkel, mint például a C, nem a műveletek (funkciók) megalkotása áll a programozás központjában, hanem az egymással kapcsolatban álló programegységek (objektumok) hierarchiájának megtervezése.

Míg a hagyományos programozási nyelvek használata során az adatok csak másodlagos szerepet töltenek be a rajtuk elvégzendő műveletekkel (függvények) szemben, addig az OOP nyelvben az adatokat és adatokon elvégzendő műveleteket egyenrangúan, zárt egységben kezeljük. Ezeket az egységeket objektumoknak hívjuk. Az adatok és az adatokat kezelő függvények (*metódus*) egységbezárása (*encapsulation*) nagyon fontos sajátossága minden objektum-orientált nyelvnek. A C++-ban az objektumoknak megfelelő tárolási egység típusát osztálynak (*class*) nevezzük. Az objektum tehát valamely osztálytípussal definiált változó, amelyet más szóhasználattal az osztály példányának nevezünk (*instance*).

Az így kialakított osztályok további, a nagyméretű programok kialakításához elengedhetetlen lehetőséggel, az adatrejtés (*data hiding*) képességével rendelkeznek. Ez azt jelenti, hogy az osztály tagjainak elérhetősége szabályozható a **private** és a **public** kulcsszavak felhasználásával.

```
// a Vektor osztály deklarációja
class Vektor {
  private:            // private elérésű adattagok
     int x,y;
  public:             // public elérésű tagfüggvények (metódusok)
     void init(int a,int b);
     int getx();
     int gety();
};

// a Vektor osztály definíciójához meg kell adnunk az összes
// tagfüggvény definícióját is! (Itt nem szerepelnek!)
void main() {
  Vektor v1;  //a Vektor típusú objektumpéldány létrehozása
  // hivatkozás az objektum tagjaira
  // a public elérésű init() tagfüggvény hívása - OK.
  v1.init(3,4);
  // HIBA! Az x és y adattagok kívülről nem érhetők el.
  v1.x=4;
}
```

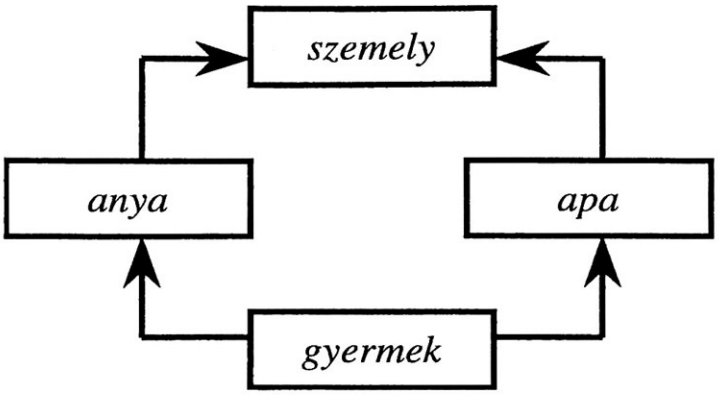
A C++ nyelvben definiált osztálytípus a nyelv szerves részévé tehető, az ún. *operator overloading* (operátor-átdefiniálás, túlterhelés) mechanizmusának felhasználásával. Ennek segítségével az általunk létrehozott típushoz definiálhatunk operátorokat, konverziókat, sőt akár specifikus I/O műveleteket is. Ugyancsak ezt a célt szolgálják az objektum automatikus inicializálását elvégző konstruktorok, illetve az objektum lebontásában fontos szerepet játszó destruktor alkalmazásának lehetősége. Ezen eszközök felhasználásával az általunk létrehozott absztrakt adattípus (*Abstract Data Type* - osztály) a C++ nyelv természetes bővítéseként használható.

Az objektum-orientált nyelvek másik fontos sajátossága az öröklés (*inheritance*). Az öröklés azt jelenti, hogy már meglévő osztály(ok)ból kiindulva újabb osztályt építhetünk fel, amely örökli a felhasznált osztály(ok) adattagjait és tagfüggvényeit. A C++-ban azt az osztályt, amelyből az új osztályt származtatjuk ős vagy alap (*base*) osztálynak, míg az új osztályt leszármaztatott (*derived*) osztálynak nevezzük. A C++ 2.0-ás változatától kezdődően használható a többszörös öröklődés (*multiply inheritance*) mechanizmusa is, amikor az új osztály származtatása során több alaposztályból indulunk ki.

Az öröklődés során tovább szűkíthető, illetve megőrizhető az alaposztály tagjainak elérhetősége a **private** és a **public** öröklődés felhasználásával. Az öröklődés lehetővé teszi, hogy az egyedi objektumok helyett a problémák leírására objektumok egymásra épülő hierarchiáját használjuk.

```
// alaposztály
class szemely {
};

// egyszeres öröklődés:
//      a személy osztályból származtatott osztály
class anya : public szemely {
};

// egyszeres öröklődés:
//      a személy osztályból származtatott osztály
class apa : public szemely {
};

// többszörös öröklődés:
//      az anya és az apa osztályokból származtatott osztály
class gyermek : public anya, public apa {
};
```

A fenti példában definiált osztály-hierarchia grafikusan is ábrázolható:



Mint említettük, a származtatott osztály örökli az alaposztály(ok) tulajdonságait (adattagjait és tagfüggvényeit), amelyek azonban meg is változtathatók:

- lehetőség van új tagok hozzáadására,
- a tagfüggvények újradefiniálására (*redefine*),
- az öröklött tagok elérhetőségének megváltoztatására.

A statikus osztályszerkezet a zártság tulajdonságának érvényesülése miatt nem minden esetben teszi lehetővé a tagfüggvények teljes újradefiniálását a származtatás során. Ekkor a többalakúság (*polymorphism*) mechanizmusát kell alkalmaznunk, amely virtuális (*virtual*) tagfüggvények formájában valósul meg a C++-ban. Az osztályhierarchián belül a virtuális tagfüggvények teljesen újradefiniálják az alaposztály ugyanilyen nevű és „kézjegyű" tagjait. Ez tehát azt jelenti, hogy attól függően, hogy programunk futása során mely szintjén vagyunk az objektum-hierarchiának, mindig más-más művelet (tagfüggvény) kerül végrehajtásra. Ily módon a program futása közben dől el, hogy végül is melyik tagfüggvényt kell aktivizálni. Ezt a jelenséget késői kötésnek (*late binding*) nevezzük, megkülönböztetésül a fordítás során megvalósított összerendeléstől (*early binding*). A virtuális tagfüggvények használata alapvető követelmény minden objektum-könyvtártól.

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

class alap {
public:
  void f1() { f2(); }
  void f2() { cout << "\n Alaposztaly"; }
};

class  uj : public alap {
public:
  void f2() { cout << "\n Szarmaztatott osztaly"; }
};
```

```
void main() {
  alap aobjektum;
  uj   uobjektum;
  aobjektum.f1();
  uobjektum.f1();
  cin.get();
}
```

Ha a fenti példában az *f2()* tagfüggvényt nem virtuális tagfüggvénynek deklaráljuk, akkor a zártság miatt a *bo.f1();* és az *uo.f1()* hívások hatására a program kimenete:

```
Alaposztaly
Alaposztaly
```

Amennyiben az alaposztályban az *f2()* tagfüggvényt virtuálisnak deklaráljuk,

```
class alap {
public:
  void f1() { f2(); }
  virtual void f2() { cout << "\n Alaposztaly"; }
};
```

a program kimenete a kívánt eredményt tartalmazza:

```
Alaposztaly
Szarmaztatott osztaly
```

A továbbiakban megismerkedünk az objektum-orientált C++ programozás eszközkészletével.

12. Osztályok

Az objektum-orientált gondolkodásmód absztrakt adattípusának (*ADT – abstract data type*) elkészítésére kétféle megoldás is a rendelkezésünkre áll a C++ nyelvben.

A C++ **struct** deklaráció a C struktúratípus kiterjesztését tartalmazza, amely a C **struct** típus minden tulajdonságával rendelkezik, azonban a kiterjesztéssel alkalmassá vált absztrakt adattípusok definiálására. A C++ rendelkezik egy új struktúrával is, a **class** (osztály) típussal.

A **struct** és a **class** típusok adattagokból (*datamember*), és ezekhez kapcsolódó műveletekből, tagfüggvényekből (*memberfunction*) épülnek fel. A **struct** és a **class** adattípusok egyaránt alkalmasak osztályok definiálására, azonban a tagok alapértelmezés szerinti hozzáférésének következtében a **class** definíció áll közelebb az objektum-orientált gondolkodásmódhoz. (Könyvünkben is ezt részesítjük előnyben.) Alapértelmezés szerint a **struct** minden tagja nyilvános elérésű, míg a **class** tagjaihoz csak az osztályból lehet hozzáférni.

Az osztálydefiníció két részből áll. Az osztály feje a **class** alapszó után az osztály nevét tartalmazza. A másik rész az osztály törzse, amelyet kapcsos zárójelek fognak közre, és pontosvessző vagy objektumlista zár. Az osztálydefiníció az adattagokon és tagfüggvényeken kívül általában a tagokhoz való hozzáférést szabályzó **public** (nyílvános, publikált), **private** (saját, rejtett) és **protected** (védett) kulcsszavakat is tartalmazza:

```
class Osztaly
{
   int num;
   float f;
 protected:
   char *ptr;
 public:
   void set (int i, float f, char *p=0) {};
   int getint(void) {};
   float getfloat(void) {};
};
```

12.1. Adattagok

Az osztályban az adattagokat a C++ nyelv változóihoz hasonlóan deklaráljuk, egyetlen különbség, hogy az adattagok nem tartalmazhatnak inicializációs listát. Ha egy osztályban valamely másik osztály objektumát kívánjuk elhelyezni, akkor a másik osztály deklarációjának meg kell előznie az aktuális osztály definícióját. A C++ mutatók és hivatkozások esetén megengedi az ún. előrevetett osztálydeklaráció használatát:

```
class Osztaly1;    // előrevetett deklaráció

class Osztaly2
{
 Osztaly1 *pO1;
 ...
};
```

Osztályon belül nem lehet közvetlenül saját osztálydefinícióval rendelkező objektumot adattagként elhelyezni. Ehhez a másik osztályt előzőleg definiálni kell.

12.2. Tagfüggvények

Az adattagokon műveletek végző tagfüggvényeket az osztály törzsén belül deklaráljuk. Ez a deklaráció a függvény prototípusát vagy pedig az **inline** definícióját tartalmazza:

```
class Osztaly
{
   int num;
   float f;
 protected:
   char *ptr;
 public:
   void set (int i, float f, char *p=0);    // prototípus
   int getint(void) { return num; }         // inline definíció
   float getfloat(void) { return f; }       // inline definíció
};
```

Az osztály deklarációját általában külön deklarációs állományban helyezzük el, hogy elkerüljük a nem **inline** tagfüggvényeinek többszöri definícióját (*Osztaly.h*). A nagyobb tagfüggvényeket az osztálydefiníción kívül adjuk meg, a hatókör (*::*) operátor felhasználásával:

```
#include "Osztaly.h"
void Osztaly::set(int i, float f, char *p) {
  num = i;
  this->f = f;
  ptr = p;
}
```

A tagfüggvények több szempontból is különböznek a hagyományos C++ függvényektől:

- Valamely osztály tagfüggvényei mindig elérik a saját osztály (**public**, **private**, **protected**) adattagjait. Nem tagfüggvény függvények csak a **public** adattagokhoz férhetnek hozzá (ha nem **friend** függvényei az osztálynak.)
- A tagfüggvények az osztály érvényességi körében definiáltak, ellentétben a közönséges függvényekkel, amelyek fájl szintű vagy névtér szintű hatókörrel rendelkeznek.
- Tagfüggvényt csak ugyanazon osztálybeli tagfüggvénnyel lehet átdefiniálni (túlterhelni), hiszen az átdefiniálás (*overloading*) mechanizmusa csak azonos érvényességi körrel rendelkező függvények között használható.

Fontos megértenünk az osztály adattagjainak és tagfüggvényeinek tárolását. Ha valamely osztálytípussal objektumot (osztály típusú változót) hozunk létre:

```
Osztaly t1, t2;
```

akkor minden objektum saját adattagokkal rendelkezik (amennyiben az adattag nem statikus), és a tagfüggvények egyetlen példányát megosztva használják.

```
t1.set(1, 3.5);
t2.set(2, 5.6, "Hello");
```

	t1
num	1
f	3.5
ptr	0x0

	t2
num	2
f	5.6
ptr	0x231279

A példában feltételeztük, hogy a „*Hello*" karaktersorozat a *0x231279* címen helyezkedik el a memóriában.

Felvetődik a kérdés, honnan tudja a *set()* függvény, hogy adott esetben mely adatterülettel kell dolgoznia? Erre a kérdésre a fordító nem látható tevékenysége adja meg a választ. Minden tagfüggvény, még a (**void**) is, rendelkezik egy nem látható paraméterrel (**this**), amelyben a fordító a hívás során az aktuális objektumra mutató pointert ad át. A fentieken kívül minden adattag-hivatkozás automatikusan formában kerül be a kódba.

```
this->adattag
```

A példánkban szereplő osztály tagfüggvényeinek ***valódi*** (csak a fordító által használt) definíciója:

```
void set (int i, float f, char *p=0, Osztaly * this) {
  this->num = i;
  this->f = f;
  this->ptr = p;
}
int getint(Osztaly * this) { return this->num;}
float getfloat(Osztaly * this) { return this->f; }
```

A **this** (ez) mutatót közvetlenül is felhasználhatjuk a tagfüggvényen belül. A előző példában a

```
this->f=f;
```

utasításban azért volt rá szükség, mivel az adattag és a tagfüggvény paramétere ugyanazt a nevet (*f*) viseli, és a fordító először mindig a „legközelebbi" (az aktuális hatókörben fellelhető) nevet használja a fordítás során.

12.2.1. Konstans tagfüggvények és a mutable típusminősítő

A közönséges C++ függvények készítése során konstans paramétereket használtunk annak érdekében, hogy a függvényen belül ne lehessen módosítani az értéküket. Egy osztály tagfüggvényei (paraméterek helyett) az osztály adattagjain fejtik ki hatásukat. Amennyiben a konstans paramétereknek megfelelő működéshez szeretnénk jutni, konstans tagfüggvényeket kell használnunk. Konstans tagfüggvények esetében a függvényfejet a **const** kulcsszó zárja.

Konstans tagfüggvényből egyetlen adattag sem módosítható. Bizonyos esetekben előfordul, hogy az adattagok többségét nem kívánjuk megváltoztatni, azonban néhány adattag értékét módosítani szükséges. Hogy az ilyen esetekben se kelljen globális változókat használnunk, a szabvány bevezette a **mutable** típusminősítőt. A **mutable** (változékony) minősítésű adattagokat a konstans tagfüggvényekből is megváltoztathatjuk.

Az *Osztaly* példánkat az elmondottak alapján módosítottuk, bevezetve egy számláló adattagot, melynek értékét a *getint()* minden hívásakor eggyel növeljük:

```
class Osztaly {
   int num;
   float f;
   mutable int szamlalo;      // megváltoztatható
 protected:
   char *ptr;
 public:
   // alapértelmezett konstruktor
   Osztaly() { f=num=szamlalo=0; ptr=NULL;}
   void set (int i, float f, char *p=0);
   // konstans tagfüggvények
   int getint(void) const { szamlalo++; return num; }
   float getfloat(void) const { return f; }
};
```

A **mutable** kulcsszó segítségével jelölt osztálytagok minden esetben megváltoztathatók, még akkor is, ha konstansként definiáljuk az osztály példányát. (Felhívjuk a figyelmet, hogy a **mutable** előírás nem használható **static** és **const** adattagokra.)

```
class szemely {
   const char * nev;
   mutable int eletkor;
   unsigned long tbszam;
   friend void szulinap(const  szemely &);     // barát függvény
  public:
   szemely(const char *, int, unsigned long); // konstruktor
};

szemely::szemely(const char * nev, int kor, unsigned long szam){
   this->nev = nev;
   eletkor = kor;
   tbszam = szam;
}

void szulinap(const szemely & valaki) {
  ++valaki.eletkor;  // ok
  // ez azonban hibas: ++valaki.tbszam;
}

void main() {
  const szemely nata("Lafenita",23,21365791223ul);
  szulinap(nata);  // Ok
  cin.get();
}
```

12.3. Az osztály tagjainak elérése

A C++ az osztály koncepcióban megvalósítja az információrejtés elvét. A **public** (nyilvános), **private** (saját) és a **protected** (védett) alapszavakkal az egyes tagok elérését szabályozhatjuk.

```
class Cls {
   // Itt helyezkednek el az alapértelmezés szerinti private
   // elérésű tagok.
   // Természetesen a private: is használható a tagok előtt
protected:
   // protected adattagok és tagfüggvények
 public:
   // korlátozás nélkül elérhető adattagok és tagfüggvények
};
```

Az osztály deklarációjában több **public**, **private** és **protected** rész is szerepeltethető. Az egyes szekciókban az adattagok és a tagfüggvények deklarációját tetszőleges sorrendben megadhatjuk. A C++ szabványban az egyes részeket **public, protected** és **private** sorrendben ajánlják elhelyezni:

```
class Osztaly {
 public:
   void set (int i, float f, char *p=0);   // prototípus
   int getint(void) { return num; }        // inline definíció
   float getfloat(void) { return f; }      // inline definíció
 protected:
   char *ptr;
 private:
   int num;
   float f;
};
```

Nézzük meg az elérést szabályzó egyes előírások jelentését!

- A **public** tag bárhonnan elérhető a programon belül, ahonnan maga az objektum elérhető. Az adatrejtés elvének érvényesüléséhez ajánlott, hogy nyilvános eléréssel csak a tagfüggvényeket deklaráljuk.
- A **protected** tagok külső függvények számára **private,** de a származtatott osztályok tagfüggvényei számára nyilvános elérésűek. Az osztálytagok védett elérése az osztályhierarchia kialakításánál, vagyis az öröklésnél (*inheritance*) játszik szerepet.
- A **private** tagokat csak az osztály saját tagfüggvényeiből, illetve az osztály „barátaiból" (***friend***) érhetjük el. A külső függvények és a származtatott osztály tagfüggvényei (bár a saját tagok is örökölődnek), nem rendelkeznek hozzáférési joggal a **private** osztálytagokhoz. Általában az osztály adattagjait **private** vagy **protected** eléréssel deklaráljuk. Mindkét elérési direktíva védi a tagokat az osztály *felhasználójától*, aki példányosítja az osztályt. A védett tagok azonban elérhetők maradnak az osztály *továbbfejlesztője* számára, aki saját osztályt származtat belőle.

Mivel az adattagok általában nem elérhetők az osztály felhasználói számára, ún. nyílvános elérési tagfüggvényeket szokás definiálni az adattagokhoz (*setxxx()*, *getxxx()*). Az elérési tagfüggvények hívásával az adattagokhoz való hozzáférés, ellentétben a közvetlen hozzáféréssel, az ellenőrzésünk mellet megy végbe.

12.4. Az osztályok friend mechanizmusa

Vannak esetek, amikor az adatrejtési szabályok nem kívánt korlátozást jelentenek a programozó számára. A **friend** (barát) mechanizmus lehetővé teszi, hogy az osztály **private** és **protected** tagjait nem saját tagfüggvényből is elérjük. A **friend** deklarációt az osztály deklarációján belül kell elhelyeznünk. A barát lehet egy külső függvény, de akár egy egész osztály is (annak minden tagfüggvénye):

```
class Sclass;

class Tclass {
  int szamlal(int x) {};
};

class Fclass {
  // Az Sclass osztály barát, így minden tagfüggvényére
  // vonatkozik a baráti viszony. Ez a kapcsolat azonban nem
  // kölcsönös!
  friend Sclass;

  // A Tclass osztálynak csak a szamlal() tagfüggvénye barát.
  friend int Tclass::szamlal(int x);

  // A sum függvény barát.
  friend long sum(void);
  // ...
};
```

Példaként tekintsük a síkbeli pontok leírásához használható egyszerűsített *Pont* osztályt! A pontok távolságát számító művelet nem kapcsolható egyik ponthoz sem, így teljesen jogos az igény, hogy külső függvényt használjunk a távolság meghatározásához, amely paraméterként kapja a két pontot. Az adattagok gyors eléréséhez azonban szükséges a közvetlen hozzáférés biztosítása, ami a „barát" mechanizmus révén meg is valósítható.

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

class Pont{
  friend double tavolsag(const Pont & p1, const Pont & p2);
  private:
    int x,y;
  public:
    void beallit(int _x, int _y) {x=_x; y=_y;}
};

double tavolsag(const Pont & p1, const Pont & p2) {
  return sqrt(pow(p1.x-p2.x,2)+pow(p1.y-p2.y,2));
}

void main() {
  Pont p,q;
  p.beallit(1,2);
  q.beallit(4,6);
  cout<<tavolsag(p,q)<<endl;
  cin.get();
}
```

12.5. Az osztály objektumai

C++ nyelvben az objektumok az osztály példányait (*instance*) jelölik, melyeket a szokásos változó-definícióval hozunk létre. Az *Osztaly* osztály példányait az alábbiak szerint definiálhatjuk:

`Osztaly x;`	**Statikusan** létrehozott *Osztaly* típusú objektum.
`Osztaly & xr=x;`	Referencia az *x* objektumhoz.
`Osztaly *xp;`	Mutató *Osztaly* típusú objektumra.
`xp= new Osztaly;`	Az *Osztaly* típusú objektum **dinamikus** létrehozása.

Az objektumtagok eléréséhez statikus esetben a pont (.) operátort, míg dinamikusan létrehozott objektumok tagjaihoz a nyíl (->) műveletet használjuk.

```
x.set(4,5.6);

xr.set(4,5.6);

xp->set(4,5.6);
```

A dianmikusan létrehozott objektumok megszüntetéséről magunknak kell gondoskodni a **delete** operátor felhasználásával:

```
delete xp;
```

A függvényargumentumként használt, illetve függvényértékként definiált objektum érték szerint adódik át. Ezekben az esetekben a fordító ideiglenes példányt készít az objektumról, ahova az adattagokat átmásolja. Az objektum adattagjainak másolása egyszerű értékadáskor is megtörténik.(Az értékadás az egyetlen művelet, melyet az objektumokra minden további nélkül alkalmazhatunk.)

```
Osztaly t1, t2;
t1.set(4,5.6);
t2 = t1;
```

Felhívjuk a figyelmet arra, hogy az értékadás után a *ptr* mutató mindkét objektumban ugyanazt a címet tartalmazza - amire a *t1.ptr* mutat, az nem másolódott át. Vannak esetek, amikor ez működés nem elfogadható. A C++ lehetőséget biztosít arra, hogy az alapértelmezés szerinti másolási műveletet *(X::operátor=(const X&))* átdefiniáljuk (túlterheljük). Hasonló a probléma akkor is, ha egy új objektumot már meglévővel inicializálunk:

```
Osztaly a;
Osztaly b=a;
```

Ekkor a meglévő objektum adattagjai szintén a fenti mechanizmus szerint másolódnak át az új objektum adattagjaiba. A másolást az ún. másoló (*copy*) konstruktor (*X::X(const &X)*) végzi, amelyet szintén átdefiniálhatunk.

Az elmondottak összefoglalásaként tekintsük az alábbi programrészletet! (Az osztály nevével (*RosszOsztaly*) jeleztük, hogy az adattag nyilvános elérésű.)

```
class RosszOsztaly {
 public:
   int a;
   void init(int x) { a = x; }
};

void main() {
   RosszOsztaly x;      // alapértelezett konstruktor
   RosszOsztaly, y=x;   // másoló konstruktor
   RosszOsztaly *px;
   px = new RosszOsztaly;
   x.a = 12;         px->a=23;
   x.init(12);       px->init(23);
   *px = x;              // az objektum másolása
   delete px;
}
```

Az előzőekben már szóltunk a **this** mutató szerepéről. Most csak egy gondolat erejéig térünk vissza hozzá. Gyakori megoldás, hogy valamely tagfüggvénynek objektum-referenciát, objektum-értéket vagy objektum-mutatót kell visszaadnia. Ekkor a **return** utasításban a **this** mutatót használjuk. A **return *this**; utasítás magát az objektumot (referencia esetén), vagy egy másolatot az objektumról (érték esetén) ad vissza függvényértékként. A **return this**; kifejezés pedig az objektumra mutató pointerrel tér vissza.

12.6. Statikus osztálytagok használata

Érdekes felhasználási lehetőséget kínál a statikus adattagok definiálása az osztályban. A **static** adattagot (mint ahogy a tagfüggvényeket) megosztva használják az osztály objektumai. Ezzel elkerülhetjük azokat a problémákat, amelyek a nem osztálytag globális változók használatából származhatnak. A statikus adattag közvetlenül az osztályhoz tartozik, így az akkor is elérhető, ha egyetlen objektuma sem létezik az adott osztálynak. A statikus adattag inicializálását az osztályon kívül kell elvégezni (függetlenül az adattag elérhetőségétől).

Ha a statikus adattag **public** elérésű, akkor a programban bárhonnan felhasználhatjuk az osztály neve és a hatókör (*::*) operátor magadásával.

Az alábbi példában a statikus tagok használatának bemutatásán túlmenően, szemléltetjük a konstansok osztályban való elhelyezésének megoldásait (**const** és **enum**). Az általunk definiált matematikai osztály (*Math*) lehetővé teszi, hogy a *Sin()* és a *Cos()* tagfüggvények hívásakor radián vagy fok mértékegységet használjunk:

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

class Math {
 private:
    static double dFokRad;
    static bool eRadian;
 public:
    typedef double valos;
    enum Egyseg {fok, rad};
    static const double Pi;

    static double Sin(double x)
           {return sin(eRadian==rad ? x : dFokRad*x);}
    static double Cos(double x)
           {return cos(eRadian==rad ? x : dFokRad*x);}
    static void Mertek(Egyseg e=rad) { eRadian=e; }
};

// A statikus adattagok létrehozása, és a kezdőértékek beállítása
const double Math::Pi=M_PI;
double Math::dFokRad = Math::Pi/180;
bool Math::eRadian = Math::rad;
```

A példában látható módon, az osztályon belül **typedef** tárolási osztállyal definiált típust is elhelyezhetünk. Az így elkészített típusnévre és a felsorolt (**enum**) konstansokra az osztálynévvel minősített név használatával hivatkozhatunk:

```
Math::valos
Math::rad
Math::fok
```

Az így létrehozott nevek osztály hatókörrel rendelkeznek, függeltenül az **enum** kulcsszó után megadott típusnévtől (*Math::Egyseg*).

Az osztály lehetséges alkalmazását az alábbiakban láthatjuk:

```
void main (){
   Math::valos y = Math::Sin(Math::Pi/6); // radiánban számol
   Math::Mertek(Math::fok);               // fokokban számol
   y = Math::Sin(30);
   Math::Mertek(Math::rad);               // radiánban számol
   y = Math::Sin(Math::Pi/6);
}
```

A statikus adattagok kezelésére statikus tagfüggvényeket használhatunk. A statikus tagfüggvényekkel azonban a normál adattagokat nem érhetjük el, hiszen a paraméterek között nem szerepel a **this** mutató.

12.7. Osztálytagra mutató pointerek

C++-ban egy függvényre mutató pointer még akkor sem veheti fel valamely tagfüggvény címét, ha különben a típusuk és a paraméterlistájuk teljesen megegyezik. Ennek oka az, hogy a (nem statikus) tagfüggvények az osztály példányain fejtik ki hatásukat. Ugyanez igaz az adattagokhoz rendelt mutatók esetén is. A mutató helyes definiálásakor az osztály nevét és a hatókör operátort is használnunk kell:

`class POsztaly;`	Osztálynév-deklaráció,
`int POsztaly::*x;`	*x* mutató egy **int** típusú adattagra,
`int (POsztaly::*fx)(void);`	*fx* mutató egy olyan tagfüggvényre mutathat, amelyet argumentum nélkül hívunk, és **int** értéket ad vissza.

A következő példában bemutatjuk az osztálytagokra mutató pointerek használatát, ahol mindhárom tagot mutató segítségével érjük el. Ilyen mutatók alkalmazása esetén a tagokra a szokásos operátorok helyett a .* (*pont csillag*), illetve a ->* (*nyíl csillag*) operátorokkal kell hivatkoznunk.

```
class POsztaly {
  public:
    int a;
    void set(int x)  { a = x;}
    int get() const { return a;}
 };

void main()
{
   int POsztaly::*ip;
   void (POsztaly::*sp)(int);
   int (POsztaly::*gp)(void);
   ip=&POsztaly::a;              // a mutatók beállítása
   sp= POsztaly::set;
   gp= POsztaly::get;
   POsztaly o1,*o2;
   (o1.*sp)(12);                 // az o1.set() meghívása
   (o1.*ip)++;                   // az o1.a léptetése
   o2 =new POsztaly;
   (o2->*ip)*=23;                // az o2->a szorzása 23-mal
   int iv=(o2->*gp)();           // az o2->get() meghívása
   delete o2;
}
```

13. Konstruktorok és destruktorok

A programozás során egyik legnehezebben felfedhető hiba, ha az automatikus változók értékét kezdőértékadás nélkül használjuk. A hiba különlegessége abban áll, hogy az ilyen változók értékét a memória-területen található „szemét" adja, ami sok esetben véletlenszerű. C++ objektumok esetén az inicializálásról, már az osztály készítése során gondoskodhatunk.

Ebben a részben két speciális tagfüggvénnyel ismerkedünk meg, melyek feladata az osztály példányainak inicializálása (konstruktor), illetve a megszüntetés előtti „takarítás" (destruktor).

13.1. Konstruktorok

Mint láttuk, a programban használt változók és objektumok megfelelő inicializálása fontos feladat, hisz e lépés nélkül a program viselkedése véletlenszerűvé válhat. A C nyelvben megszokott struktúra-inicializálás a **class** típusú objektum esetén akkor használható, ha az csak **public** elérésű adattagokkal rendelkezik. A C++ programokban az objektumok inicializálásának feladatát speciális tagfüggvényekkel a konstruktorokkal oldjuk meg.

A konstruktor olyan speciális tagfüggvény, amelynek neve megegyezik az osztály nevével, és nem rendelkezik típussal. Az osztály konstruktorát a fordító minden olyan esetben automatikusan meghívja, amikor az adott osztály objektuma létrejön. A konstruktor nem rendelkezik visszatérési értékkel, de különben ugyanúgy viselkedik, mint bármely más tagfüggvény. A konstruktor átdefiniálásával (túlterhelésével) többféleképpen is inicializálhatjuk az objektumokat.

Fontos megértenünk, hogy a konstruktor nem foglal tárterületet a létrejövő objektum számára - ezt a fordító végzi az alaptípusoknál használt módszerrel. (Ha az objektum dinamikus, akkor a **new** operátor foglal memóriaterületet.) A konstruktor feladata a már lefoglalt memóriaterület inicializálása. Ha az objektum valamilyen mutatót tartalmaz, ami elég gyakori megoldás, akkor természetesen a konstruktorból kell gondoskodnunk a mutató által kijelölt terület létrehozásáról.

Az alábbi egydimenziós, egész tömböt megvalósító *Vektor* osztályban több konstruktort is definiáltunk:

```
#ifndef VektorH
#define VektorH
class Vektor {
  public:                              // Példa az inicializálásra:
   Vektor();                           // Vektor a;
   Vektor(int n);                      // Vektor a(12);
   Vektor(const Vektor& v);            // Vektor a(12), b=a, c(b);
   Vektor(const int a[],int n); // int x[2]={1,2}; Vektor a(x,2)
  private:
        int *p;
        int meret;
};
#endif
```

Egy osztály alapértelmezés szerint két konstruktorral rendelkezik: a paraméter nélküli (*default -(X::X())* és a másoló (*copy - X::X(const &X))* konstruktorral. Ha valamilyen saját konstruktort készítünk, akkor a paraméter nélküli alapértelmezett (*default*) konstruktor többé nem lesz elérhető, így azt is definiálnunk kell. Saját másoló konstruktort általában akkor készítünk, ha valamilyen dinamikus tárterület tartozik az osztály példányaihoz (mint a példánkban).

Ha a fenti osztálydefiníciót a *vektor.h* fejállományban tároljuk, akkor az alábbi *vektor.cpp* fájl tartalmazhatja a tagfüggvények definícióját:

```
#include "Vektor.h"

// A paraméter nélküli alapértelmezett konstruktor 10-elemű
// tömböt hoz létre:
Vektor::Vektor(void) {
    p = new int[meret = 10];
}

// Adott méretű vektor inicilaizálása
Vektor::Vektor(int n) {
    p = new int[meret=n];
}

// Inicializálás másik vektorral - másoló konstruktor
Vektor::Vektor(const Vektor& v) {
     p = new int[meret=v.meret];
     for (int i = 0; i < meret; ++i)  // az elemek átmásolása
        p[i] = v.p[i];
}

// Inicializálás hagyományos n-elemű vektorral
Vektor::Vektor(const int a[], int n) {
    p = new int[meret=n];
    for (int i = 0; i < meret; ++i)
       p[i] = a[i];
}
```

Az első két tagfüggvényt össze is vonhatjuk az alapértelmezett függvény-argumentum kijelölésével:

```
class Vektor {
  public:
   Vektor(int n=10);
   Vektor(const Vektor& v);
   Vektor(const int a[],int n);
  private:
        int *p;
        int meret;
};

Vektor::Vektor(int n) {
    p = new int[meret=n];
}
```

Az eredeti megoldásnál maradva, mind a négy konstruktor helyet foglal a memóriából a vektor elemei számára. Nézzünk néhány példát a *Vektor* típusú objektumok definiálására!

Az első (*default*) konstruktor hívódik meg az alábbi objektumok létrehozását követően. Felhívjuk a figyelmet arra, hogy objektumtömb (3. és 4. definíció) esetén minden tömbelem konstruktora lefut.

```
Vektor a, b;                // minkét vektor 10-elemű
Vektor *c = new Vektor;     // c pointer egy 10-elemű vektorra
Vektor d[5];                // 5 darab 10-elemű vektor tömbje
Vektor *e = new Vektor[5]; // 5 darab 10-elemű vektor tömbje
delete c;
delete []e;
```

A második konstruktor aktivizálódik az alábbi definiciókban, melynek során 20-elemű egész tömbök jönnek létre. Az utolsó példában 3-elemű objektumtömböt hozunk létre, melynek elemei 3-, 7- és 9-elemű egész tömböket tartalmaznak.

```
Vektor f = Vektor(20);
Vektor g(20);
Vektor h = 20;
Vektor *i = new Vektor(20);
delete i;
Vektor j[]={3,7,9};
```

A következő definíciókban a harmadik (másoló) konstruktor használatára láthatunk példákat:

```
Vektor k;              // a default konstruktor hívódik meg
Vektor l=k;
Vektor m(k);
Vektor n[3]={k,l,m};
Vektor *o=new Vektor(k);
delete o;
```

Az alábbiakban a negyedik konstruktor hívódik meg:

```
int z[5] = {4,13,7,26,30};
Vektor p(z,5);
Vektor *q=new Vektor(z,5);
delete q;
```

A dinamikus objektumtömbök létrehozása során csak a paraméter nélküli (*default*) konstruktor hívására van mód. Ezzel szemben a statikus objektumtömbök definíciójában szereplő inicializációs lista más konstruktorok aktivizálását is lehetővé teszi.

A konstruktorok egyaránt lehetnek **public**, **private** és **protected** elérésűek. A csak **private** konstruktorokat tartalmazó osztályt rejtett osztálynak nevezzük. Az ilyen osztály objektumait csak *barát* függvényben, illetve *barát* osztályban hozhatjuk létre. A csak **protected** konstruktorokat tartalmazó osztály, amely nem rendelkezik *barátokkal* (*friends*), jól használható absztrakt alaposztályként, amelyből más osztályokat származtatunk

Érdemes összefoglalva áttekinteni a konstruktorok használatával kapcsolatos szabályokat!

- Neve megegyezik az őt tartalmazó osztály nevével.
- Nincs visszatérési értéke (**void** típusú sem).
- Deklarációjában nem használhatjuk a **virtual**, **static**, **const**, **mutable** és **volatile** kulcsszavakat.
- A neve átdefiniálható (túlterhelhető).
- Nem öröklődik.
- Nem készíthető mutató a konstruktorhoz.
- Az objektumpéldány létrehozása után nem hívható.
- Az argumentum típusa és az osztálytípus közötti konverzió letiltható az **explicit** kulcsszó megadásával.

13.1.1. A konstruktorok explicit paraméterezése

Általában az egyparaméteres konstruktorral rendelkező osztályok példányainak olyan kifejezést adunk kezdőértékül, amely típusával illeszkedik a konstruktor paraméterének típusához. Ha a típusegyezés nem 100-százalékos, akkor a fordító implicit konverziókat használ a megfelelő konstruktor kiválasztásához. Az **explicit** kulcsszó megadásával megtilthatjuk az alapértelmezett implicit konverziók alkalmazását a konstruálás során.

```
class Vektor {
  public:
      explicit Vektor();
      explicit Vektor(int n);
      explicit Vektor(const Vektor& v);
      explicit Vektor(const int a[],int n);
private:
      int *p,
      int meret;
};
```

Az előző *Vektor* osztály módosítása után az alábbi két definícióban hibát jelez a fordító, mindkettőben az *int->Vektor* átalakítást kifogásolja:

```
Vektor h = 20;
Vektor j[] = {3,7,9};
```

13.2. Destruktorok

A fenti *Vektor* példában a konstruktorok memóriát foglalnak a vektor elemei számára, a **new** operátor felhasználásával. Ez a memória mindaddig lefoglalt marad, amíg fel nem szabadítjuk. Erre a célra a C++ biztosít egy speciális tagfüggvényt, a destruktort, amelyben gondoskodhatunk a lefoglalt tárterület felszabadításáról. A destruktor nevét hullám karakterrel (*tilde*, ~) egybeépített osztálynévként kell megadni. A destruktor, akárcsak a konstruktor, szintén nem rendelkezik visszatérési típussal.

A destruktort tartalmazó *Vektor* osztályban a destruktor felszabadítja a konstruktor által lefoglalt memóriát, amikor az objektum megszűnik:

```
class Vektor {
  public:
        Vektor();
        Vektor(int n);
        Vektor(const Vektor& v);
        Vektor(const int a[],int n);
        ~Vektor() {delete[] p; }   // inline destruktor
 private:
        int *p;
        int meret;
};
```

Ha az osztály rendelkezik destruktorral, a fordító minden olyan esetben meghívja azt, amikor az objektum érvényessége megszűnik. Kivételt képeznek a **new** operátorral dinamikusan létrehozott objektumok, melyek destruktorát csak a **delete** operátor megadásával aktivizálhatjuk. Szintén fontos megjegyeznünk, hogy a destruktor nem magát az objektumot szűnteti meg, hanem automatikusan elvégez néhány általunk megadott „takarítási" műveletet. (A destruktor közvetlenül is meghívható, például: *a.~Vektor();*)

Míg objektumtömb esetén az elemek konstruktorai az indexek növekvő sorrendjében hívódnak meg, addig a destruktorhívások sorrendje fordított. Az elmondottak dinamikus helyfoglalású tömbökre csak akkor érvényesek, ha a **delete[]** operátort alkalmazzuk a tömb megszüntetésére:

```
Vector * vt;
vt = new Vector[5];
// ...
delete []vt;
```

Amennyiben a tömb törlését (helytelenül) a **delete** művelettel végezzük, csak a 0. tömbelem destruktora fut le:

```
delete vt;
```

A destruktorok használatához szintén egy sor, a konstruktoroknál említett szabály kötődik:

- Neve a hullám (~) karaktert követően megegyezik az őt tartalmazó osztály nevével.
- Ha nem készítünk destruktort, akkor a fordító az alapértelmezettet használja.
- Nincs visszatérési értéke (**void** típusú sem).
- Deklarációjában nem szerepeltethetjük a **static**, **const**, **mutable** és **volatile** kulcsszavakat.
- Nem készíthető mutató a destruktorhoz.

A destruktorok azonban több pontban különböznek a konstruktoroktól:

- Nincsenek paraméterei.
- Nem definiálható át (nem terhelhető túl), azonban az öröklés során lecserélhető.
- A destruktor virtuálissá (**virtual**) tehető. Ha az alaposztály destruktora virtuális, akkor a származtatott osztály destruktora is automatikusan virtuális lesz.
- Explicit módon is hívható.

13.3. Az objektum tagosztályainak inicializálása

Ha egy osztály tagként valamely másik osztály példányát tartalmazza, akkor a tagosztály konstruktorának inicializálását a külső osztály konstruktorában oldjuk meg. Ez a mechanizmus a tag-inicializációs lista használatát jelenti, amelyet a konstruktor után kettősponttal elválasztva adunk meg. A lista vesszővel elválasztott elemei az osztály tagjai, melyek után zárójelben áll az inicializációs argumentum. A példában a *Kulso* osztályon belüli *Belso* típusú objektumot inicializáljuk a *Kulso* konstruktorában.

```
class Belso {
    int cnt;
  public:
    Belso(int v) : cnt(v) {} // inline konstruktor
};

class Kulso {
   int num;
   Belso obj;
  public:
   Kulso(int, int); // a konstruktor prototípusa
};

Kulso:: Kulso(int k, int b) : obj(b)
{
  num = k;
}
```

A tag-inicializációs listát csak a konstruktor-definícióban adhatjuk meg. Ezt a módszert azonban tetszőleges típusú adattag esetén is használhatjuk. A fenti konstruktor egy lehetséges másik alakja:

```
Kulso::Kulso(int k, int b) : obj(b), num(k)
{
}
```

A tag-inicializációs lista használata kötelező, ha az osztály referencia típusú adattagot vagy paraméterezett konstruktorral rendelkező objektumot tartalmaz:

```
class Kulso {
   int num;
   Belso & obj;
 public:
   Kulso(int a, Belso & br) : obj(br), num(a) {};
};
```

Az inicializációs lista feldolgozása a konstruktor törzsének végrehajtása előtt megy végbe.

14. Operátorok túlterhelése (operator overloading)

A C++ nyelv biztosítja annak a lehetőségét, hogy valamely, programozó által definiált függvényt szabványos operátorhoz kapcsoljunk, kibővítve ezzel az operátor működését. Ez a függvény automatikusan meghívódik, amikor az operátort egy meghatározott szövegkörnyezetben használjuk.

Operátorfüggvényt azonban csak akkor definiálhatunk, ha annak legalább egyik argumentuma osztály (**class, struct**) típusú. Ez azt jelenti, hogy a **void** függvények, illetve a csak alap adattípusú argumentumokat használó függvények nem lehetnek operátorfüggvények. Az operátorfüggvény deklarációjának formája:

```
típus  operator op(paraméterlista);
```

ahol az *op* helyén az alábbi C++ operátorok valamelyike állhat:

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

Nem definiálhatók át a tagkiválasztás (.), az indirekt tagkiválasztás (.*), a hatókörfeloldás (::) , a feltételes (?:), a **sizeof** és a **typeid** operátorok. Felhívjuk a figyelmet arra, hogy az operátorok túlterhelésével nem változtatható meg az operátorok elsőbbsége (precedenciája) és csoportosítása (asszociativitása), valamint nincs mód új műveletek bevezetésére sem.

Az operátorfüggvényeket általában osztályon belül definiáljuk, a felhasználói típus lehetőségeink kiterjesztése céljából. Az =, *()*, *[]* és -> operátorokat azonban csak *nem statikus tagfüggvénnyel* lehet átdefiniálni. A **new** és a **delete** operátorok esetén a túlterhelés *statikus tagfüggvénnyel* történik. Minden más operátorfüggvény megadható tagfüggvényként, vagy az osztály **friend** függvényeként.

A szabványos C++ műveleti jeleket az operandusok száma alapján két csoportra oszthatjuk. Erre a két esetre az alábbi táblázatban foglaltuk össze az átdefiniált operátor és a tagfüggvények kapcsolatát.

Kétoperandusú operátor esetén:

Megvalósítás	*Szintaxis*	*Aktuális hívás*
tagfüggvény	*X op Y*	*X.**operator** op(Y)*
friend függvény	*X op Y*	***operator** op(X,Y)*

Egyoperandusú operátor esetén:

Megvalósítás	*Szintaxis*	*Aktuális hívás*
tagfüggvény	*op X*	*X.**operator** op()*
tagfüggvény	*X op*	*X.**operator** op(0)*
friend függvény	*op X*	***operator** op(X)*
friend függvény	*X op*	***operator** op(X,0)*

Példaként tekintsük a *Vektor* osztály kibővített változatát, amelyben túlterheltük az indexelés (*[]*), az értékadás (=) és az összeadás (+, +=) műveletfüggvényeket! Az értékadás megvalósítására a tömb elemeinek másolása érdekében volt szükség. A + operátort barát függvénnyel valósítjuk meg, mivel a keletkező *Vektor* logikailag egyik operandushoz sem tartozik. Ezzel szemben a += művelet megvalósításához tagfüggvényt használunk.

```
class Vektor {                                  // Vektor.h
  public:
    Vektor();
    Vektor(int n);
    Vektor(const Vektor& v);
    Vektor(const int a[],int n);
    ~Vektor() {delete[] p; }
    int fh() const { return meret; }            // a méret lekérdezése
    int& operator[] (int i);                    // az [] operátor
    Vektor operator= (const Vektor& v);         // az =  operátor
    Vektor operator+= (const Vektor& v);        // a  += operátor
                                                // a  +  operátor
    friend Vektor  operator+ (const Vektor& v1,
                              const Vektor & v2);
 private:
    int *p;
    int meret;
};

// Vektor.cpp (részlet)

int& Vektor::operator [] (int i) {
    if (i < 0 || i > meret-1)  // indexhatár-ellenőrzés
        throw i;
    return p[i];
}
```

```
Vektor Vektor::operator = (const Vektor& v) {
    delete []p;
    p=new int [meret=v.meret];
    for (int i = 0; i < meret; ++i)
       p[i] = v.p[i];
    return *this;
}

Vektor Vektor::operator += (const Vektor& v) {
    int m = (meret < v.meret) ? meret : v.meret;
    for (int i = 0; i < m; ++i)
       p[i] += v.p[i];
    return *this;
}

Vektor operator + (const Vektor& v1, const Vektor& v2) {
    Vektor sum(v1);
    sum+=v2;
    return sum;
}
```

A *Vektor* osztály felhasználását az alábbi programrészlet szemlélteti:

```
#include <iostream>
using namespace std;
#include "Vektor.h"

void show(Vektor & v) {
  for (int i=0; i<v.fh(); i++)
    cout<<v[i]<<'\t';
  cout<<endl;
}

void main() {
    int a[]={4,9}, b[]={12,23,79};
    Vektor x(a,2),y(b,3),z;
    try {
       show(x);          // 4       9
       show(y);          // 12      23      79
       x=y;
       show(x);          // 12      23      79
       x=Vektor(a,2);
       show(x);          // 4       9
       x+=y;
       show(x);          // 16      32
       z=x+y;
       show(z);          // 28      55
    }
    catch (int n) {
        cout<<"Hibas tombindex: "<<n<<endl;
    }
    cin.get();
}
```

14.1. A new és a delete operátorok túlterhelése

A **new** és a **delete** operátorok átdefiniálásakor további megkötéseket kell szem előtt tartanunk. A **new** operátor osztálytag operátorfüggvényét **void** * visszatérési értékkel és ***size_t*** típusú első paraméterrel kell definiálni. Ebben az argumentumban a fordító automatikusan átadja az osztály bájtban kifejezett méretét.

A ***delete*** operátor esetén, az osztálytag operátorfüggvényt **void** * első és ***sizet_t*** típusú opcionális második paraméterrel kell létrehoznunk. Az utóbbi paramétert, amennyiben megadtuk, a fordító használja az első paraméter által megcímzett objektum méretének átadására.

Az átdefiniált **new** és **delete** operátorok mellett, a hatókör operátor megadásával mindig elérhetjük az eredeti **new** és **delete** műveleteket:

```
char *p=::new char[1001];
::delete p;
```

A **new** és **delete** operátorok automatikusan az őket definiáló osztály statikus tagjaként kerülnek lefordításra. Ebből következik, hogy a **this** mutató nem használható, tehát az operátorfüggvényekből csak a statikus adattagokat érhetjük el. Ez azért szükséges, mivel a **new** és **delete** hívását az osztályobjektum létezése nélkül is végre kell tudni hajtani.

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

class Osztaly {
 public:
  double d;
  void * operator new(size_t n);
  void operator delete(void *p, size_t n);
};

void * Osztaly::operator new(size_t n) {
  cout<<"Foglalas:"<<n<<endl;
  return ::new char[n];
}

void Osztaly::operator delete(void *p, size_t n){
  cout<<"Torles"<<endl;
  ::delete p;
}

void main() {
    Osztaly *p=new Osztaly; // Foglalas:8
    p->d=1223;
    delete p;               // Torles
    cin.get();
}
```

14.2. Felhasználó által definiált típuskonverzió

A C++ nyelv támogatja, hogy a programozó a saját adattípusához (osztály) típuskonverziókat rendeljen hozzá. Az ilyen felhasználó által definiált típuskonverziót végrehajtó tagfüggvény deklarációja:

```
operator típus();
```

A függvény visszatérési értékének típusa megegyezik a függvény nevében használt típussal. A típuskonverziós operátor csak visszatérési típus és argumentumlista nélküli tagfüggvény lehet.

Az alábbi példában a komplex típust osztályként valósítjuk meg. Az egyetlen nem *cplx* típusú argumentummal rendelkező konstruktor elvégzi a más típusról - a példában **double** - *cplx* típusra történő konverziót. A fordított irányú konverzióhoz **double** nevű konverziós operátort definiáltunk.

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

class cplx {
 public:
   cplx () { re=im=0; }
   cplx(double a) { re=a; im=0; }   // konverziós konstruktor
   cplx(double a, double b) {re=a; im=b;}
   // konverziós operátor
   operator double() {return sqrt(re*re+im*im);}
 private:
  double re, im;
};

void main() {
   cplx a(3,4);
   cout << double(a)<< endl;        // a kiírt érték:  5
   cout <<double(cplx(10))<< endl; // a kiírt érték: 10
}
```

Megjegyezzük, hogy a *clpx* osztály három konstruktora egyetlen konstruktorral helyettesíthető, amelyben alapértelmezett paramétereket használunk:

```
cplx(double a=0, double b=0) {re=a; im=b;}
```

14.3. Az osztályok bővítése input/output műveletekkel

A C++ nyelv lehetővé teszi, hogy az osztályokon alapuló adatfolyamoknak „megtanítsuk" saját készítésű osztályunk objektumainak kezelését. Az adatfolyam osztályok közül az ***istream*** az adatbevitelért, míg az ***ostream*** az adatkivitelért felelős. Az input/output műveletek végzéséhez pedig az átdefiniált >> és << operátorokat használjuk. A szükséges működés eléréséhez *friend* operátorfüggvényként el kell készítenünk a fenti műveletek saját átdefiniált változatát, mint ahogy az a *cplx* osztály bővített változatában látható:

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

class cplx {
 public:
    cplx(double a=0, double b=0) {re=a; im=b;}
    operator double() {return sqrt(re*re+im*im);}
    friend istream & operator>>(istream &, cplx &);
    friend ostream & operator<<(ostream &, const cplx &);
 private:
   double re, im;
};

// Az adatbevitel formátuma: 12.23+7.79i, illetve 12.23-7.79i
istream & operator>>(istream & is, cplx & c) {
    char p[256];
    is.getline(p,256);
    if (sscanf(p,"%lf%lfi",&c.re, &c.im)!=2)
       c=cplx(0);
    return is;
}

// Adatkivetei formátum: 12.23+7.79i, illetve 12.23-7.79i
ostream & operator<<(ostream & os, const cplx & c) {
    os<<c.re<<(c.im<0? '-' : '+')<<abs(c.im)<<'i';
    return os;
}

void main() {
    cplx a, b;
    cout<<"Kerek egy komlex szamot: ";
    cin >>a;
    cout<<"A komplex szam: "<<a<<endl;
    cin.get();
}
```

14.4. A C++ nyelv bővítése saját típussal

Az eddigi ismereteink elegendőek ahhoz, hogy a C++ nyelvet saját adattípussal bővítsük. Az adattípushoz osztályt kell készítenünk megfelelő adattagokkal, majd gondoskodnunk kell a különféle konstruálási lehetőségekről. Ezt követően a szükséges operátorok túlterhelésével a típusunk ugyanúgy használható, mint a nyelv alaptípusai.

Példaként tekintsük a C-karaktersorozatokat lefedő *stringc* osztályt!

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

// Az osztály deklarációja
class stringc{
  public:
    stringc();                          // alapértelmezett konstruktor
    stringc(char );                     // char -> stringc konv.
    stringc(const char *);              // char * -> stringc konv.
    stringc(const stringc &);           // másoló konstruktor
    ~stringc(void) {delete[]p;}         // destruktor
    char&   operator[]( int);                 // index operátor
    stringc operator = (stringc);             // =  operátor
    bool    operator == (stringc);            // == operátor
    operator char *() const {return p;} // stringc->char * konv.
    operator int()  {return h;}               // a sztring hossza
    stringc  operator + ( stringc);           // +  operátorok
    friend  stringc operator+(stringc, char);
    friend  stringc operator+(char, stringc);
                                              // i/o műveletek
    friend  ostream& operator<<(ostream& , const stringc &);
    friend  istream& operator>>(istream& , stringc &);
  private:
    char * p;
    int    h;
    stringc(int);          // helykészítő konstruktor
};

// Az osztály definíciója
// Konstruktorok

stringc :: stringc() {
        h = 0;
        p = new char[1]; assert(p!=0);
        *p = 0;
}
```

```
stringc :: stringc(char c) {
        h = 1;
        p = new char[2]; assert(p!=0);
        p[0] = c;
        p[1] = 0;
}

stringc :: stringc(const char * s) {
        h = strlen(s);
        p = new char[h+1]; assert(p!=0);
        strcpy(p, s);
}

stringc :: stringc(const stringc& s ) {
        h = s.h;
        p = new char[h+1]; assert(p!=0);
        strcpy(p, s.p);
}

stringc :: stringc(int n) {
        p = new char[n+1]; assert(p!=0);
        p[0] = 0;
        h = n;
}

// Operátor tagfüggvények

char& stringc :: operator[](int i) {
        if (i < 0  || i > h)
                throw "Indexhatar tullepes!\n";
        return p[i];
}

stringc stringc :: operator=(stringc s) {
        if ( this == &s )
          return *this;       // önmagába nem másolunk
        delete p;
        h = s.h;
        p = new char[h+1]; assert(p!=0);
        strcpy(p, s.p);
        return * this;
}

bool stringc :: operator == (stringc s) {
        return !strcmp(p, s.p);
}
stringc stringc :: operator + (stringc s) {
        int th = h + s.h;
        stringc ts(th);
        strcpy(ts.p, p);
        strcat(ts.p, s.p);
        return ts;
}
```

```
// Friend függvények

stringc operator + (stringc s, char c) {
        stringc cs(c);
        return s + cs;
}
stringc operator + (char c, stringc s) {
        stringc cs( c );
        return cs + s;
}
ostream& operator<<(ostream& out, const stringc & s) {
        out<<s.p;
        return out;
}
istream& operator>>(istream& in, stringc & s) {
         char cp[256];
         in.getline(cp,256);
         s=stringc(cp);
         return in;
}

// Az osztály használatát bemutató főprogram

void main( ) {
    stringc s1;
    stringc s2("Lafenita");
    stringc s3 = s2;
    stringc s4 = "C#  nyelv";

    cout<<"Kerek egy nevet: ";
    cin>>s1;
    cout<<s1<<endl;

    s3="Stroustrup";
    s1 = s3;

    s4[1] = s4[2] ='+';
    cout<<s4<<endl;

    if ( s1 == s3) {
            s2 = s1 + stringc(" & ") + s4;
            s3 = s2 + '\n';
            cout<<s3;
    }

    s2 = '{' + s2 + '}';
    cout << s2<<endl;

    char *sp=new char[s2];
    strcpy(sp,s2);
    cout<<s2<<endl;
    cin.get();
}
```

A *main()* függvény futásának eredményét az alábbiakban láthatjuk

```
Kerek egy nevet: Bjarne Stroustrup
Bjarne Stroustrup
C++ nyelv
Stroustrup & C++ nyelv
{Stroustrup & C++ nyelv}
{Stroustrup & C++ nyelv}
```

Gyakran használjuk a függvényhívás operátorának átdefiniált változatát, az ún. *iterátor* tagfüggvény elkészítéséhez. Az iterátor segítségével egymás után elérhetjük a karaktersorozat karaktereit.

A *stringc* osztály bővítése után

```
char operator() () ;                    // iterátor

char stringc :: operator () () {
        static int index=0;
        if (index<h)
            return p[index++];
        else
            return (index=0);
}
```

az alábbi programrészlettel egymás alatt jelenítjük meg az *s4* sztring karaktereit:

```
char ch;
do
   cout<<(ch=s4())<<endl;
while (ch);
```

15. Az öröklés (öröklődés) mechanizmusa

Az öröklés (*inheritance*) az objektum-orientált C++ nyelv legfőbb sajátossága. Ez a mehanizmus teszi lehetővé, hogy bizonyos osztályokból más osztályokat származtassunk, mely osztályok a származtatás során adattagokat és tagfüggvényeket örökölnek. Az öröklött tulajdonságok tetszőlegesen kiterjeszthetők és megváltoztathatók. A C++ támogatja a többszörös öröklődést (*multiple inheritance*), melynek során valamely új osztályt több alaposztályból származtatunk.

15.1. A származtatott osztályok

A származtatott osztály (*derived class*) olyan osztály, amely az adattagjait és a tagfüggvényeit egy vagy több előzőleg definiált osztálytól örökli. Azt az osztályt, amelytől a származtatott osztály örököl, alaposztálynak (*base class*) nevezzük. A származtatott osztály szintén lehet alaposztálya további osztályoknak, lehetővé téve ezzel osztályhierarchia kialakítását.

A származtatott osztály az alaposztály minden tagját örökli, de az alaposztályból csak a **public** és **protected** tagokat éri el sajátjaként. A származtatott osztályban az öröklött tagokat saját adattagokkal és tagfüggvényekkel egészíthetjük ki. A származtatás kijelölésére az osztály fejét használjuk:

```
class Szarmaztatott : public Alap1, ...private AlapN
{
 // az osztály törzse
}
```

A származtatási listában megadott **public, protetected** és **private** kulcsszavak az öröklött (*nyilvános* és *védett*) tagok elérhetőségét szabályozzák.

Az öröklés módja	*Alaposztálybeli elérés*	*Hozzáférés a származtatott osztályban*
public	public	public
	protected	protected
	private	-
protected	public	protected
	protected	protected
	private	-
private	public	private
	protected	private
	private	-

A **public** származtatás során az öröklött tagok megtartják az alaposztálybeli elérhetőségüket, míg a **private** származtatás során az öröklött tagok a származtatott osztály privát tagjaivá válnak. Védett (**protected**) öröklés esetén az öröklött tagok védett tagok lesznek az új osztályban. (Az alapértelmezés szerinti származtatási mód **class** típusú alaposztály esetén a **private**, míg a **struct** típust használva a **public.**)

Fontos megjegyeznünk, hogy a **public** származtatással létrehozott osztály minden esetben (értékadás, függvény-argumentum,...) helyettesítheti az alaposztályt. Ezen nem kell csodálkozni, hisz a származtatott osztály magában foglalja az alaposztályt.

A friend viszony az öröklés során

Az alaposztály barátja (**friend**) a származtatott osztályban csak az alaposztályból öröklött tagokat érheti el. A származtatott osztály barátja (**friend**) az alaposztályból csak a **public** és a statikus **protected** tagokat érheti el.

Az örökölt tagok elérése

Általában a származtatott osztály öröklött tagjai ugyanúgy érhetők el, mint a saját tagok. Azonban elképzelhető olyan eset, amikor az öröklött tagok elérése akadályba ütközik. Ha származtatott osztály azonos névvel újradefiniálja az öröklött tagot, az láthatatlanná válik. Ilyen esetben a hatókör operátort kell használnunk a hivatkozáshoz:

```
Alap::tag
```

Nézzünk egy példát az elmondottak szemléltetésére!

```
// Az alaposztály
class B {
   int x, y;
 public:
   int b1, b2;
   int Bfunc1(void) { return x;}
   int Bfunc2(void) { return y;}
};

// A származtatott osztály
class D: private B  {
   // az öröklött b2 elfedése
   int b2;
   int d;
 public:
   // a private származtatással öröklött b1 public elérésű lesz
   B::b1;
   // az öröklött Bfunc1() elfedése
   void Bfunc1(void);
};
```

```
void D::Bfunc1(void) {
     d =B::Bfunc1(); // a nem látható Bfunc1() elérése
     b1=Bfunc2();    // a Bfunc2() látható
     b2=B::b2;       // a nem látható b2 adattag elérése
}
void main() {
   D od;
   od.b1=23;
}
```

Az alábbi táblázatban összefoglaltuk, hogy mely osztály milyen tagokkal rendelkezik:

A B alaposztály tagjai:	*A D származtatott osztály tagjai*
`private: x, y`	`private: B::b2, b2, d, B::Bfunc1(), Bfunc2()`
`public: b1, b2, Bfunc1(), Bfunc2()`	`public: b1, Bfunc1()`

Öröklés vagy beágyazás?

Valamely probláma objektum-orientált megoldása során mérlegelni kell, hogy az öröklés vagy a beágyazás segítségével jutunk-e pontosabb modellhez. A döntés általában nem egyszerű, mi is csak a technika bemutatására szorítkozunk.

Az alábbi két példaprogramból jól láthatók a két megoldás közötti különbségek, illetve azonosságok.

Megoldás örökléssel

```
class Pont {
 protected:
  int x,y;
 public:
  Pont(int a=0, int b=0){x=a, y=b;}
  Pont(const Pont & p){x=p.x, y=p.y;}
  int getx() {return x;}
  int gety() {return y;}
};
class Kor : public Pont { // öröklés
  protected:
    int r;
  public:
    Kor(int x=0, int y=0, int r=0)
        : Pont(x, y), r(r) {}
    Kor(const Pont & p, int r=0)
        : Pont(p), r(r) {}
    int getr() {return r;}
};

void main() {
   Kor k1(100, 200, 23);
   cout<<k1.getx()<<endl;
   cout<<k1.getr()<<endl;
   cin.get();
}
```

Megoldás beágyazással

```
class Pont {
 protected:
  int x,y;
 public:
  Pont(int a=0, int b=0) {x=a, y=b;}
  Pont(const Pont & p) {x=p.x, y=p.y;}
  int getx() {return x;}
  int gety() {return y;}
};
class Kor {
  protected:
    int r;
  public:
    Pont p; // beágyazás
    Kor(int x=0, int y=0, int r=0)
        : p(x, y), r(r) {}
    Kor(const Pont & p, int r=0)
        : p(p), r(r) {}
    int getr() {return r;}
};
void main() {
    Kor k1(100, 200, 23);
    cout<<k1.p.getx()<<endl;
    cout<<k1.getr()<<endl;
    cin.get();
}
```

15.2. Az alaposztály inicializálása

Az alaposztály(ok) inicializálására a tag-inicializációs lista kibővített változatát használjuk, amelyben a tagok mellett a közvetlen ős osztályok konstruktorait is felsoroljuk.

Példaként tekintsük az *Alap1* és *Alap2* osztályokból származtatott *Szarmaztatott* osztály konstruktorait!

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

class Alap1 {
  private:
     int na;
     long ta;
  public:
     Alap1(int _na, long _ta) {na=_na; ta=_ta;}
     Alap1(Alap1 & nata) {*this = nata;}
};
class Alap2 {
  private:
     char *pt;
     bool lv;
  public:
     Alap2(bool _lv, char * _pt) {lv=_lv; pt=_pt;}
     Alap2(Alap2 & lvpt) {*this = lvpt;}
};
class Szarmaztatott : public Alap1, public Alap2 {
  private:
    char ln;
  public:
    Szarmaztatott(int na, long ta, bool lv, char *pt, char ln);
    Szarmaztatott (Szarmaztatott & dc);
};

Szarmaztatott::Szarmaztatott(int na, long ta,
       bool lv, char *pt, char ln) : Alap1(na, ta), Alap2(lv, pt)
{
   // a származtatott osztály konstruktora
}

Szarmaztatott:: Szarmaztatott (Szarmaztatott & dc)
                                      : Alap1(dc), Alap2(dc)
{
   // a származtatott osztály konstruktora
}

void main() {
    Szarmaztatott d1(23, 1979, true, "Lafenita", 'N');
    Szarmaztatott d2(d1);
    cin.get();
}
```

Az alaposztályok konstuktorai az inicializációs lista szerinti sorrendben hívódnak, balról-jobbra haladva. Először az alaposztályok konstruktorai kerülnek végrehajtásra, majd a tagosztályok konstruktorai következnek (ha vannak), és végül a sort a származtatott osztály konstruktorának végrehajtása zárja.

Az alaposztály inicializálása elvégezhető alaposztály típusú objektummal, vagy a származtatott osztály objektumával, ha **public** módon származtattuk az új osztályt. (A fenti példa második konstruktorában használtuk ezt a megoldást.) Az inicializálást ekkor a másoló konstruktor végzi.

15.3. Virtuális tagfüggvények – polimorfizmus

A virtuális függvény olyan **public** vagy **protected** tagfüggvénye a **public** alaposztálynak, amelyet a származtatott osztályban újradefiniálhatunk az osztály „viselkedésének" megváltoztatása (polimorfizmus) érdekében. A virtuális függvény általában a publikált alaposztály referenciáján vagy mutatóján keresztül hívódik meg, melynek aktuális értéke a program futása során alakul ki (dinamikus kapcsolás, késői kötés).

Ahhoz, hogy egy tagfüggvény virtuális legyen a **virtual** kulcsszót kell használnunk a függvény deklarációja előtt:

```
class Pelda {
  public:
    virtual int pf();
};
```

Nem szükséges, hogy az alaposztályban a virtuális függvénynek a definíciója is szerepeljen. Ebben az esetben ún. tisztán virtuális függvénnyel (*pure virtual function*) van dolgunk:

```
class Pelda {
  public:
    virtual int pf()=0;
};
```

Egy vagy több tisztán virtuális függvényt tartalmazó osztállyal (*absztrakt osztállyal*) nem készíthetünk objektumpéldányt. Az absztrakt osztály csak az öröklés alaposztályként használható.

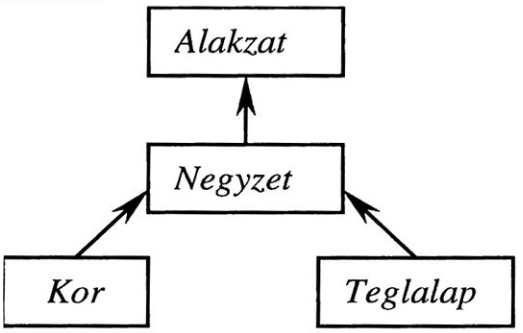
15.3.1. A virtuális függvények újradefiniálása (redefine)

Ha egy függvényt az alaposztályban virtuálisként deklarálunk, akkor ezt a tulajdonságát az öröklődés során is megőrzi. A származtatott osztályban a virtuális függvényt saját változattal újradefiniálhatjuk, de az öröklött verziót is használhatjuk. Saját verzió definiálásakor nem szükséges a **virtual** szót megadnunk.

Ha egy származtatott osztály tiszta virtuális függvényt örököl, akkor ezt mindenképpen saját verzióval kell újradefiniálni, mivel különben a származtatott osztály is absztrakt osztály lesz. A származtatott osztály tartalmazhat olyan virtuális függvényeket is, amelyeket nem az alaposztálytól örökölt.

A származtatott osztályban a virtuális függvény újradefiniált változatának pontosan (név, típus, paraméterlista) meg kell egyeznie az alaposztályban definiálttal. Ha a két deklaráció nem pontosan egyezik, akkor az újradefiniálás helyett a túlterhelés (*overloading*) mechanizmusa érvényesül.

Az alábbi példaprogramban mindegyik alakzat saját maga számolja ki a területét és a kerületét, azonban a megjelenítést az absztrakt alaposztály (*Alakzat*) végzi. Az osztályok hierarchiája az ábrán látható:




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

// Absztrakt alaposztály
class Alakzat {
  protected:
     int x, y;
  public:
     Alakzat(int _x=0, int _y=0) {x=_x; y=_y;}
     virtual double terulet()=0;
     virtual double kerulet()=0;
     void megjelenit() {
          cout<<'('<<x<<','<<y<<")\t";
          cout<<"\tTerulet: "<<terulet();
          cout<<"\tKerulet: "<<kerulet()<<endl;
     }
};

class Negyzet : public Alakzat {
  protected:
     double a;
  public:
     Negyzet(int _x=0, int _y=0, double _a=0)
          : Alakzat(_x,_y) {a=_a;}
     double terulet() {return a*a;}
     double kerulet() {return 4*a;}
};
```

```
class Teglalap : public Negyzet {
  protected:
     double b;
  public:
     Teglalap(int _x=0, int _y=0, double _a=0,  double _b=0)
           : Negyzet(_x,_y,_a) { b=_b; }
     double terulet() {return a*b;}
     double kerulet() {return 2*(a+b);}
};

class Kor : public Negyzet {
  static const double pi;
  public:
     Kor(int _x=0, int _y=0, double _a=0)
           : Negyzet(_x,_y,_a) {}
     double terulet() {return a*a*pi;}
     double kerulet() {return 2*a*pi;}
};

const double Kor::pi=3.14159265;

void main() {
     Negyzet n(12,23,10);
     cout<<"Negyzet: ";
     n.megjelenit();

     Kor k(23,12,10);
     cout<<"Kor: ";
     k.megjelenit();

     Teglalap t(12,7,10,20);
     cout<<"Teglalap: ";
     t.megjelenit();
     cin.get();
}
```

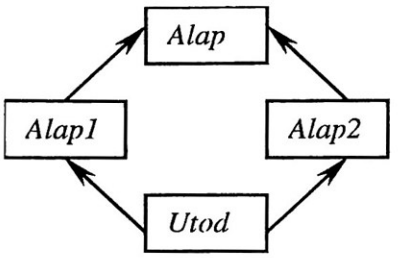
Virtuális tagfüggvényeket tartalmazó osztályok esetén a fordítóprogram az osztályhoz kapcsolódóan egy ún. virtuális táblát (VT) épít fel, amely az aktuális újradefiniált virtuális függvények címét tartalmazza. Az osztályhierarchiában található azonos nevű virtuális függvények azonos indexszel szerepelnek ezekben a táblákban, ami lehetővé teszi a virtuális tagfüggvények teljes lecserélését.

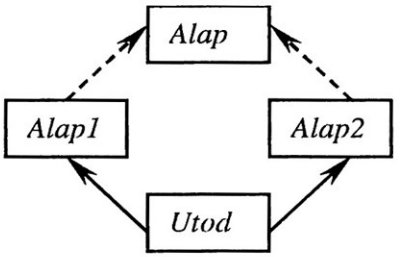
15.3.2. Virtuális destruktorok

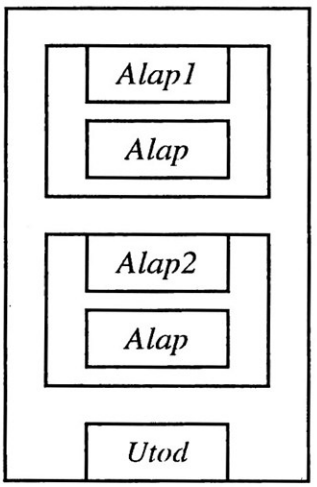
A destruktort virtuális függvényként is definiálhatjuk. Ha az alaposztály destruktora virtuális, akkor minden ebből származtatott osztály destruktora is virtuális lesz. Ezáltal biztosak lehetünk abban, hogy a megfelelő destruktor hívódik meg, amikor az objektum megszűnik.

15.4. Virtuális alaposztályok

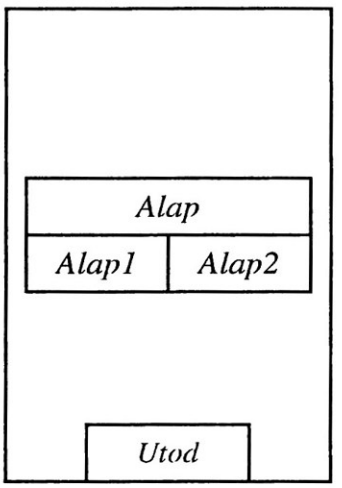
A többszörös öröklődés során problémát jelenthet, ha ugyanazon alaposztály több példányban jelenik meg a származtatott osztályban. A virtuális alaposztályok használatával az ilyen jellegű problémák kiküszöbölhetők.







az *Alap* nem virtuális alaposztály



az *Alap* virtuális alaposztály

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

class Alap {
 public:
   int q;
};
// az Alap virtuális alaposztály
class Alap1 : virtual public Alap {
   int x;
 public:
   Alap1(int i) { x=i;}
};

// az Alap virtuális alaposztály
class Alap2: public virtual Alap {
   int y;
  public:
   Alap2(int i) : y(i) {}
};
```

```
class Utod:  public Alap1,  public Alap2 {
   int a,b;
  public:
   Utod(int i,int j): Alap1(i*5),Alap2(j+i), a(i) {b=j;}
};

void main() {
  Utod ux(10,20);
  ux.Alap1::q=100;
  cout << ux.Alap2::q<<endl;     // 100
  ux.Alap2::q=200;
  cout << ux.q<<endl;            // 200
  cin.get();
}
```

Az alaposztály a **virtual** öröklés hatására csak egyetlen példányban lesz jelen a származtatott osztályokban, függetlenül attól, hogy hányszor fordul elő az öröklődési láncban. A példában a virtuális alaposztály *q* adattagját öröklik az *Alap1* és az *Alap2* alaposztályok. A virtualitás miatt az *Alap* alaposztály egyetlen példányban szerepel, így az *Alap1::q* és az *Alap2::q* ugyanarra az adattagra hivatkoznak. A **virtual** szó használata nélkül az *Alap1::q* és az *Alap2::q* különböző adattagokat jelölnek.

15.5. Futás közbeni típusinformációk (RTTI) osztályok esetén

A különböző vizuális fejlesztőrendszerek futás közbeni típusinformációkat (*Run Time Type Information*) tárolnak az objektumpéldányok mellett. Ennek segítségével a futtatórendszerre bízhatjuk az objektumok típusának azonosítását, így nem kell nekünk erre a célra adattagokat bevezetnünk. Az *RTTI* mechanizmus helyes működéséhez polimorf alaposztályt kell kialakítanunk, vagyis legalább egy virtuális tagfüggvényt el kell helyeznünk benne, és engedélyeznünk kell az *RTTI* tárolását. (Az engedélyezési lehetőséget általában a fordító beállításai között találjuk meg.)

Az alábbi példaprogramban az elérési problémák akkor jelentkeznek, amikor osztályonként különböző tagokat szeretnénk használni.

```
#include <iostream>
#include <string>
using namespace std;
class Allat {
  protected:
     int labak;
     virtual string fajta()=0;
  public:
     Allat(int n) {labak=n;}
     void Info() {
       cout<<"A(z) "<<fajta()<<"nak "
           <<labak<<" laba van."<<endl;
      }
};
```

```
class Hal : public Allat {
  protected:
     string fajta() {return "hal";}
  public:
     Hal(int n=0) : Allat(n) {}
     void Uszik() {cout<<"Uszik"<<endl;}
};

class Madar : public Allat {
  protected:
     string fajta() {return "madar";}
  public:
     Madar(int n=2) : Allat(n) {}
     void Repul() {cout<<"Repul"<<endl;}
};

class Emlos : public Allat {
  protected:
     string fajta() {return "emlos";}
  public:
     Emlos(int n=4) : Allat(n) {}
     void Fut() {cout<<"Fut"<<endl;}
};

void main() {
    const int db=3;
    Allat * p[db];
    p[0]=new Hal;
    p[1]=new Madar;
    p[2]=new Emlos;

    // RTTI nélkül is működő lekérdezés
    for (int i=0; i<db; i++)
      p[i]->Info();

    // RTTI-alapú feldolgozás
    for (int i=0; i<db; i++)
       if (dynamic_cast<Hal*>(p[i]))     // Hal?
             dynamic_cast<Hal*>(p[i])->Uszik();
       else
       if (dynamic_cast<Madar*>(p[i]))   // Madár?
             dynamic_cast<Madar*>(p[i])->Repul();
       else
       if (dynamic_cast<Emlos*>(p[i]))   // Emlős?
             dynamic_cast<Emlos*>(p[i])->Fut();
    for (int i=0; i<db; i++)
        delete p[i];
    cin.get();
}
```

A program futásának eredménye:

```
A(z) halnak 0 laba van.
A(z) madarnak 2 laba van.
A(z) emlosnak 4 laba van.
Uszik
Repul
Fut
```

16. Általánosított osztályok (templates)

A paraméterezett (általánosított) osztály (*generic class*, vagy *class generator*), lehetővé teszi, hogy más osztályok definiálásához a paraméterezett osztályt, mint mintát használjuk. Ezáltal egy adott osztálydefiníció minden típus esetén alkalmazható.

A sablonosztály általános alakja, ahol *típus1,..típusn* az általánosított típusokat jelöli:

```
template <class típus1, … class típusn> class osztalynev {

}
```

Az osztály nem **inline** tagfüggvényeinek definícióját az alábbiak szerint kell megadnunk:

```
template <class típus1, … class típusn >
fvtípus osztalynev< típus1, … típusn> :: fvnév(paraméterlista) {

}
```

Példaként tekintsük a *Pont* osztályból készített általánosított osztályt!

```
template <class tipus>
class Pont {
   private:
      tipus x,y;
   public:
      Pont(tipus a=0, tipus b=0) {setxy(a,b);}
      tipus getx() {return x;}
      tipus gety() {return y;}
      void setxy(tipus a, tipus b) {x=a; y=b;}
};
```

16.1. Specializáció és példányosítás

A sablonokat különböző módon használhatjuk. Az *implicit* példányosítás során (*instantiation*) az általánosított típusokat konkrét típusokkal helyettesítjük. Ekkor először létrejön az osztály adott típusú változata (ha még nem létezett), majd pedig az objektumpéldány:

```
Pont<double> p1, p2(12,23);
```

Explicit példányosítás során kérjük a fordítót, hogy hozza létre az osztály példányát a megadott típusok felhasználásával, így az objektum készítésekor már kész osztállyal dolgozhat:

```
template class Pont<double>;

Pont<double> p1, p2(12,23);
```

Vannak esetek, amikor a sablon felhasználását könnyíti, ha az általános változatot valamilyen szempont szerint specializáljuk (*explicit specialization*). Az alábbi deklarációk közül az első az általános sablont, a második a mutatókhoz készített változatot, a harmadik pedig a **void*** mutatókra specializált változatot tartalmazza.

```
template <class tipus> class Pont {
  // a fenti osztálysablon
};
template <class tipus> class Pont <tipus *> {
  // el kell készíteni!
};
template <> class Pont <void *> {
  // el kell készíteni!
};
```

A specializált változatokat az alábbi példányosítások során használhatjuk:

```
Pont<double> pa;
Pont<int *> pp;
Pont<void *> pv;
```

A teljesség érdekében függvénysablonok esetén is tekintsük át a példányosítás és a specializáció működését!

```
#include <iostream>
#include <cstring>
using namespace std;
// általános függvénysablon
template <class tipus>
int compare(tipus a, tipus b) {
  return int(a-b);
}
// explicit példányosítás
template int compare(int, int);

// explicit specializáció char * típusra
template <> int compare<char *>(char *p, char *q) {
   return strcmp(p,q);
}
void main() {
    int m;
    // az általános függvény hívódik meg
    m = compare(12.7,23.7);  // implicit példányosítás
    char *s1="Lafenita", *s2="Nata";
    // a specializált függvény hívódik meg
    m=compare(s1,s2);
    cin.get();
}
```

16.2. A sablonosztály „barátai" és statikus adattagjai

Az osztálysablonnak is lehetnek barátai, melyek többféleképpen viselkedhetnek. Azok amelyek nem tartalmaznak sablonelőírást, minden specializált osztály közös barátai lesznek. Ellenkező esetben a külső fügvény csak az adott példányosított változat *friend* függvényeként használható.

Az általánosított osztályban definiált statikus adattagokat osztálypéldányonként létre kell hoznunk.

16.3. Érték- és alapértelmezett sablonparaméterek

A következő egyszerű osztálysablont a típuson kívül értékkel is paraméterezzük. Mindkét esetben alapértelmezett paramétereket is megadtunk.

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

template<class Tipus=int, int MaxMeret=100>
class Stack {
   Tipus tomb[MaxMeret];
   int sp;
  public:
   Stack(void) { sp = 0; };
   void Push(Tipus adat) {
      if (sp<MaxMeret) tomb[sp++] = adat; };
   Tipus Pop(void){ return sp>0? tomb[--sp] : 0; };
   bool Ures(void) { return sp== 0; };
};

void main(void)
{
  Stack<double,1000> dStack; // 1000 elemű double verem
  Stack<char *> sStack;      // 100 elemű char * verem
  Stack<> iStack;            // 100 elemű int verem

  int a=134, b=307;
  iStack.Push(a);
  iStack.Push(b);
  a=iStack.Pop();
  b=iStack.Pop();

  sStack.Push("nyelv");
  sStack.Push("C++");
  do {
     cout << sStack.Pop()<<endl;;
  } while (!sStack.Ures());
}
```

16.4. A typename kulcsszó

A **typename** kulcsszó a típussablonok használatához kötődik. Definiálatlan osztályok esetén a **class** kulcsszó helyett használhatjuk a típus-definíciókban (**typedef**), illetve a sablon-deklarációkban is szerepeltethetjük a **class** kulcsszó helyett.

Az első példában a **typename** kulcsszót azért használjuk, mert a változókat úgy kell definiálnunk, hogy a típus (*T::Tipus*) még nem jött létre:

```
template <class T>
void fv() {
  typedef typename T::Tipus TTipus;
  TTipus a;
  typename T:: Tipus b;
  TTipus * pta;
}

class Valami{
 public:
   typedef char * Tipus;
};

// Például a függvényhívás:  fv<Valami>();
```

A **typename** kulcsszót használhatjuk a **template** deklarációkban is a **class** helyett:

```
template <typename Tip1, typename Tip2>
Tip2 Konverzio (Tip1 t1) {
   return (Tip2)t1;
}

template <typename TipX, class TipY>
bool Egyenlo (TipX x, TipY y) {
   return x==y;
}

void main() {
   bool x=Egyenlo(10,20);
   double xc=Konverzio<double,int>(10);
}
```

16.5. Összetettebb sablonpélda

Sablonhierarchia (sablonosztályok hierarchiájának) kialakításához nagy gyakorlatra van szükség, ezért csak egyszintű sablonosztályok készítésével foglalkozunk.

Készítsük el a 13. és 14. fejezetekben használt *Vektor* osztály általánosított, kibővített válozatát! Egészítsük ki az osztályt egy iterátorral is!

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

template <class T>
class Vektor {
  public:
    typedef T* iterator;

    explicit Vektor(int n=10);
    Vektor(const Vektor& v);
    Vektor(const T* a,int n);
    ~Vektor() {delete[] p; }
    int fh() const { return meret; }

    T& operator[] (int i);
    Vektor& operator= (const Vektor& v);
    Vektor  operator+= (const Vektor& v);

    iterator begin() {return p;}
    iterator end() {return p+meret;}
private:
    T *p;
    int meret;
};

// Konstruktorok
template <class T> Vektor<T>::Vektor(int n) {
    p = new T[meret=n];
}

template <class T> Vektor<T>::Vektor(const Vektor& v) {
     p = new T[meret=v.meret];
     for (int i = 0; i < meret; ++i)
        p[i] = v.p[i];
}

template <class T> Vektor<T>::Vektor(const T a[], int n) {
    p = new T[meret=n];
    for (int i = 0; i < meret; ++i)
       p[i] = a[i];
}

// operátor-tagfüggvények
template <class T> T& Vektor<T>::operator[](int i) {
    if (i < 0 || i > meret-1) throw i;
    return p[i];
}

template <class T> Vektor<T>& Vektor<T>::operator=
                                    (const Vektor<T>& v) {
    delete []p;
    p=new T [meret=v.meret];
    for (int i = 0; i < meret; ++i)
       p[i] = v.p[i];
    return *this;
}
```

```
template <class T> Vektor<T> Vektor<T>::operator+=
                                        (const Vektor<T>& v) {
    int m = (meret < v.meret) ? meret : v.meret;
    for (int i = 0; i < m; ++i)
       p[i] += v.p[i];
    return *this;
}

// külső operátorfüggvény
template <class Tipus> Tipus operator+
                                (const Tipus& v1, const Tipus& v2) {
    Tipus sum(v1);
    sum+=v2;
    return sum;
}
```

Egy lehetséges főprogram, amelyben elvégezzük az osztálysablon példányosítását, vagyis konkrét típus behelyettesítésével való megvalósítását:

```
void main() {
   int a[5]={1,2,3,4,5};
   int b[3]={7,12,23};

   // Példányosítás typedef-ben
   typedef Vektor<int> iVektor;
   iVektor v1(a,5), v2(b,3), v3;
   v3=v1;
   v3=v2+v1;

   // Lokális példányosítás double típussal
   Vektor<double>::iterator p;
   Vektor<double> x,y,z;
   for (p=x.begin(); p!=x.end(); p++)
     *p=1223.79;
   y=x;
   z=x+y;
}
```

Függelék

A C++ nyelv könyvtára

F1. A szabványos C++ nyelv könyvtárainak áttekintése

A szabványos C++ különböző összetevőkből épül fel. Egyrészt átvette a szabványos (ANSI) C nyelv függvénykönyvtárát, amely egy sor jól használható függvényt és típust tartalmaz. Az átvett könyvtárak lehetőségeit (például *climits*) néhány esetben C++ átiratban is megtaláljuk (például *limits*). A C++ nyelvhez kapcsolódóan új elemek is megjelentek (például *new*, *exception, iostream* stb.). A C++ könyvtár elemeit az alábbiak szerint csoportosíthatjuk. A csoportosítás során kiemeltük a szabványos sablonkönyvtárhoz (*Standard Template Library – STL*) tartozó állományokat.

A C++ nyelvet támogató könyvtár

Típusok (*NULL*, *size_t* stb.)	<cstddef>
Az implementáció jellemzői	<limits> <climits> <cfloat>
Programindítás és –befejezés	<cstdlib>
Dinamikus memóriakezelés	<new>
Típusazonosítás	<typeinfo>
Kivételkezelés	<exception>
Egyéb futásidejű támogatás	<cstdarg> <csetjmp> <ctime> <csignal> <cstdlib>

Hibakezelési könyvtár

Kivételosztályok	<stdexcept>
Assert makrók	<cassert>
Hibakódok	<cerrno>

Általános szolgáltatások könyvtára

Műveleti elemek (*STL*)	<utility>
Funkció objektumok (*STL*)	<functional>
Memóriakezelés (*STL*)	<memory>
Dátum- és időkezelés	<ctime>

Sztring könyvtár	
Sztring osztályok	<string>
Nullavégű karakterláncok kezelését segítő függvények	<cctype> <cwctype> <cstring> <cwchar> <cstdlib>
Országfüggő (helyi) beállítások könyvtára	
A helyi sajátosságok szabványos kezelés	<locale>
A C könyvtár helyi beállításai	<clocale>
A tárolók könyvtára (STL)	
Adatsorok (*STL*)	<deque> <list> <queue> <stack> <vector>
Asszociatív tárolók (*STL*)	<map> <set> <bitset>
Iterárotok (általánosított mutatók) könyvtára (STL)	
Iterátor elemek, előre definiált iterátorok, adatfolyam-iterátorok (*STL*)	<iterator>
Algoritmusok könyvtára	
Adatsor-kezelés, rendezés, keresés stb. (*STL*)	<algorithm>
A C könyvtár algoritmusai	<cstdlib>
Numerikus könyvtár	
Komplex számok	<complex>
Számtömbök	<valarray>
Általánosított numerikus műveletek (*STL*)	<numeric>
A C könyvtár numerikus elemei	<cmath> <cstdlib>

Input/output könyvtár	
Forward (előrevetett) deklarációk	<iosfwd>
Szabványos *iostream* objektumok	<iostream>
Az *iostream* osztályok alaposztálya	<ios>
Adatfolyam pufferek	<streambuf>
Adatformázás és manipulátorok	<istream> <ostream> <iomanip>
Sztringadatfolyamok	<sstream> <cstdlib>
Fájladatfolyamok	<fstream> <cstdio> <cwchar>

A szabványos C++ könyvtár azonosítóit az ***std*** névterületen keresztül érhetjük el. A C++ szabvány a fentieken kívül tartalmaz még 18 deklarációs állományt a szabványos C könyvtárból, melyek különbözhetnek az új változataiktól. (A C könyvtár nevek a globális névtérben tárolódnak)

C deklarációs állomány	*C++ megfelelő*
<ctype.h>	<cctype>
<errno.h>	<cerrno>
<float.h>	<cfloat>
<iso646.h>	<ciso646>
<limits.h>	<climits>
<locale.h>	<clocale>
<math.h>	<cmath>
<setjmp.h>	<csetjmp>
<signal.h>	<csignal>
<stdarg.h>	<cstdarg>
<stddef.h>	<cstddef>
<stdio.h>	<cstdio>
<stdlib.h>	<cstdlib>
<string.h>	<cstring>
<time.h>	<ctime>
<wchar.h>	<cwchar>
<wctype.h>	<cwctype>

F2. Adatok bevitele és kivitele

A C++ nyelv nem rendelkezik adatbeviteli és adatkiviteli utasításokkal, minden ilyen feladatot könyvtári függvényekkel, illetve objektumokkal kell megoldanunk.

Az operációs rendszer szintjén a C++ program számára az adatátvitel mindig adatállomány olvasását és írását jelenti, jóllehet a program bemenete valójában a billentyűzet, kimenete pedig a képernyő vagy a nyomtató. Az adatátvitel egysége a bájt (szó), amit a C++ nyelvben a **char (wchar_t)** típus valósít meg.

Az adatfolyam-kezelő (*stream*) függvények és objektumok az állományokat, mint karakterek folyamát tekintik. A C nyelvtől örökölt lehetőségekhez a ***cstdio***, míg az objektum-orientált megoldásokhoz az ***iostream*** fejállományt kell a programunkba beépíteni.

A legtöbb operációs rendszer a C++ programok indításakor három előre definiált adatfolyamot automatikusan megnyit, lehetővé téve ezzel az alapvető perifériális egységek elérését. A C nyelvtől örökölt módon a (***FILE*** * típusú) ***stdin*** a szabványos input-, az ***stdout*** a szabványos output-, míg az ***stderr*** a szabványos hiba-adatfolyamot jelöli. A C++ könyvtár objektumokat biztosít ezen adatfolyamok elérésére, melyek rendre a ***cin***, a ***cout*** és a ***cerr***. (A **wchar_t** típusú széles-karakterekből álló adatfolyamok objektumai a ***wcin***, a ***wcout*** és a ***wcerr***). Az elmondottakat táblázatban foglaltuk össze:

C adatfolyam	*C++ objektum*	*Periféria*	*A művelet iránya*
Stdin	***cin***	billentyűzet (átirányítható)	*input*
stdout	***cout***	képernyő (átirányítható)	*output*
stderr	***cerr***	képernyő	*output*
-	***clog***	képernyő – a ***cerr*** pufferelt változata	*output*

A következők két alfejezetben először áttekintjük a fenti adatfolyamok elérését támogató megoldásokat. A fejezetet az adatállományok kezeléséhez szükséges ismeretek összegzése zárja.

F2.1. A C-könyvtár alapvető I/O műveletei (*cstdio*)

A következő táblázatban összefoglaltuk az azonos jellegű input-, illetve output- műveletek elvégzésére alkalmas szabványos C-függvényeket, külön kiemelve a függvények széles-karaktereket használó párját:

stdin	*stdout*	*sztring*	*stream I/0*	*széles-karakteres függvények*
getchar()	-	-	*getc()*	*getwchar(), getwc()*
-	-	-	*Ungetc()*	*ungetwc()*
fgetchar()	-	-	*fgetc()*	*fgetwchar(), fgetwc()*
gets()	-	-	*fgets()*	*getws(), fgetws()*
-	-	-	*getw()*	
-	*putchar()*	-	*putc()*	*putwchar(), putwc()*
-	*fputchar()*	-	*fputc()*	*fputwchar(), fputwc()*
-	*puts()*	-	*fputs()*	*putws(), fputws()*
-	-	-	*putw()*	
-	-	-	*fread()*	
-	-	-	*fwrite()*	
scanf()	-	*sscanf()*	*fscanf()*	*wscanf(), swscanf() ,fwscanf()*
vscanf()	-	*vsscanf()*	*vfscanf()*	*vwscanf(), vswscanf(), vfwscanf()*
-	*printf()*	*sprintf()*	*fprintf()*	*swprintf(), swprintf(), fwprintf()*
-	*vprintf()*	*vsprintf()*	*vfprintf()*	*vwprintf(), vswprintf(), fwprintf()*

A fejezetben bemutatásra kerülő függvények prototípusát a *cstdio* fejállomány tartalmazza:

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

F2.1.1. Karakterek kiírása és beolvasása

Az adatfolyam jellegű adatkezelésnél alapvető műveletnek számít egyetlen karakter beolvasása és kiírása, mely feladatok elvégzésére makrókat tartalmaz a szabvány:

```
#define getchar()  getc(stdin)
#define putchar(c) putc((c), stdout)
```

A legegyszerűbb adatbeviteli függvényszerű makró a ***getchar()***,

```
int getchar(void);
```

amely egy karaktert olvas a szabványos inputeszközről (billentyűzetről). Ha az olvasás sikeres volt, akkor a beolvasott karakter kódját kapjuk meg visszatérési értékként. A fájlvége esemény bekövetkeztekor *EOF* értékkel tér vissza a ***getchar()***. (Billentyűzet esetén a *<Ctrl+Z>* billentyűkombináció váltja ki ezt az eseményt.)

A legegyszerűbb adatkiviteli függvényszerű makró a ***putchar()***,

```
int putchar(int ch);
```

amely a megadott *ch* karaktert kiírja a szabványos outputeszközre (képernyőre). Sikeres működés esetén a *ch* karakter kódját kapjuk meg visszatérési értékként.

F2.1.2. Karaktersorozatok kiírása és beolvasása

A C könyvtárban találunk függvényeket a karaktersorozatok (sztringek) egyetlen hívással történő beolvasására (***gets()***) és kiírására (***puts()***).

```
char* gets(char *sptr);
```

A **gets**() függvény egy sort (<*Enter*> lenyomásáig) olvas a szabványos inputról, majd a karaktereket az argumentumban megadott *sptr* mutató által kijelölt területre másolja. A beolvasott sztringben az újsor ('\n') karakter '\0' (*EOS*) karakterrel helyettesítődik, ezért a függvény nem alkalmas az újsor karakter beolvasására. (Erre a célra a ***getchar()*** makrót kell használnunk.) A ***gets()*** visszatérési értéke sikeres olvasás esetén a megadott puffer címe, illetve *NULL*, ha hiba lép fel.

```
int puts(char *sptr);
```

A ***puts()*** függvény az argumentumban megadott karaktersorozatot a szabványos kimenetre (a képernyőre) írja. A sztring megjelenítését automatikusan a '\n' (újsor) karakter kiírása követi. A függvény egy nem negatív szám visszaadásával jelzi a sztring sikeres kiírását, ellenkező esetben pedig *EOF* (*-1*) értékkel tér vissza.

F2.1.3. Formázott adatbevitel és –kivitel

A szabványos C-könyvtár ***printf()*** és ***scanf()*** függvényeivel megadott formátum szerint lehet kiírni, illetve beolvasni alaptípusú adatokat és karaktersorozatokat.

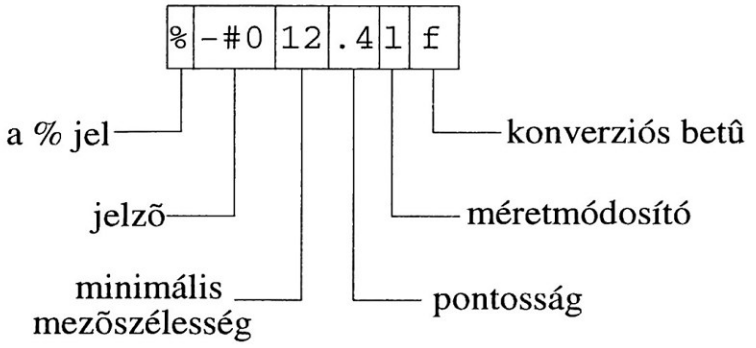
A printf() függvény

A ***printf()*** függvény hívásának formája:

```
printf(formátum, argumentumlista);
```

A formátum, amely általában sztringkonstans, kétféle karaktert tartalmazhat. Az első csoport karakterei normál karaktersorozatokat alkotnak a formátumsztringen belül, amelyek minden átalakítás nélkül kerülnek kiírásra.

A másik csoport karakterei speciális műveleteket írnak elő, amelyek egyrészt meghatározzák az argumentumok értelmezésének módját, másrészt pedig a megjelenítés formáját. A konverziós előírások százalékjellel (%) kezdődnek és valamilyen konverziós karakterrel zárulnak. Az előírás teljes formájában hat jól elkülöníthető részre osztható:



Az alábbiakban áttekintjük az egyes részek szerepét és a részek megadásának lehetőségeit:

Jelző

-	a mínuszjel megadása esetén a konvertált argumentum a mező bal széléhez igazítva jelenik meg,
+	(pluszjel) számok kiírásakor az előjel (+ vagy -) mindig megjelenik,
szóköz	az előjel helyén a mínuszjel vagy szóköz jelenik meg,
0	a szóköz helyett 0-val töltődik fel a kiírt adat előtti terület,
#	(*hashmark*) az ún. alternatív kiírási kép jele. (A *%#x* formátum hatására a hexadecimális számok előtt a *0x* előtag, míg a *%#o* formátum esetén az oktális számok előtt a *0* előtag jelenik meg.)

Ha a jelző karaktereket elhagyjuk, akkor az output a megadott mezőben jobbra igazítva jelenik meg, és balról szóköz karakterek töltik ki a nem használt területet.

Minimális mezőszélesség

A minimális mezőszélesség decimális jegyeket tartalmazó egész konstans, amely a konvertált argumentum kiírásakor felhasznált mező szélességét határozza meg. Amennyiben az első jegy *0*, akkor azt jelzőként értelmezi a rendszer. Ha a megadott mezőben nem fér el az eredmény, akkor az a szükséges méretű mezőben fog megjelenni. Ha a csillag (*) karakterrel adjuk meg a mezőszélességet, akkor a tényleges mezőszélesség-értéket az argumentumlista soron következő eleme adja meg. Az elmondottak alapján az alábbi két kiírási művelet eredménye ugyanaz lesz:

```
printf("%7d", 1223);
```

```
int w=7;
printf("%*d", w, 1223);
```

Konverziós karakterek

A formátumelem mindig tartalmaz legalább egy betűt, amely alapján az argumentum feldolgozását és konvertálását végzi a rendszer. Minden konverziós karakternek megfelel valamilyen argumentumtípus és nyomtatási kép.

Konverziós karakter	*Az argumentum típusa*	*Nyomtatási kép*
c	**int**	az **unsigned char** típusra való konvertálás eredményének megfelelő karakter.
d, i	**int**	Előjeles decimális egész.
u	**unsigned int**	Előjel nélküli decimális egész.
f	**double**	*[-]ddd.dddd*, ahol a tizedes jegyek száma az előírt pontosságtól függ. A pontosság alapértelmezésben *6*, a *0* pontossággal a tizedespont kiírása elnyomható.
e, E	**double**	*[-]d.ddde±ddd*, vagy *[-]d.ddddE±ddd*, ahol a tizedes jegyek száma az előírt pontosságtól függ.
g, G	**double**	Ugyanaz, mint a *%e* vagy a *%E*, ha az exponens kisebb, mint *-4*, vagy nagyobb-egyenlő, mint a pontosság, különben a hatása a *%f* formátuméval azonos. Nincsenek bevezető nullák és szóközök.
o	**unsigned int**	Előjel nélküli oktális egész a *0* előtag nélkül.
x, X	**unsigned int**	Hexadecimális egész a *0x* előtag nélkül.
p	**void***	Mutató értéke (implementációfüggő).
n	**int ***	A formátumig kiírt karakterek számát adja vissza a függvény az argumentummal kijelölt egész (**int**) változóban.
s	**char***	A karaktersorozat karaktereit írja ki az *EOS*-ig, vagy a kijelölt mező határáig.
%	-	Nincs konverzió - kiírja a % karaktert.

Pontosság p

A formátumban a pont után megadott decimális szám a megjelenítés pontosságát határozza meg. A pontosság értelmezése típusonként eltérő:

- egész számok esetén a jegyek minimális számát (*d*, *i*, *o*, *u*, *x*, *X*),
- lebegőpontos számok esetén tizedes jegyek (*e*, *E*, *f*), illetve az értékes jegyek (*g*, *G*) számát,
- sztring esetén pedig a kiíratni kívánt karakterek maximális számát határozza meg.

Ha a pontosság a specifikációból kimarad, akkor az alapértelmezés szerinti érték (*6*) érvényesül. Ha a pontosságnak *0*-át adunk meg, akkor a *d*, *i*, *o*, *u*, *x*, illetve *X* konverziós karakterek esetén az alapértelmezés szerinti számú jegy kerül kiírásra, az *e*, *E* és *f* konverziós karakterek esetén pedig elmarad a tizedespont. A csillag (*) megadása azt jelenti, hogy a tényleges pontossági értéket az argumentumlista soron következő eleme tartalmazza.

Méretmódosító előtag

A méretmódosító betű közvetlenül a konverziós karakter előtt áll. Lehetséges értékei:

H	a *d*, *i*, *o*, *u*, *x* és *X* konverziós karakterrel együtt megadva a **short int** típust jelöli,
L	a *d*, *i*, *o*, *u*, *x* és *X* konverziós karakterrel együtt megadva a **long int** típust jelöli, míg az *e*, *E*, *f*, *g* és *G* betűk esetén a **double** típust,
L	az *e*, *E*, *f*, *g* és *G* konverziós betűkkel együtt használva a **long double** típuselőírásnak felel meg, míg a *d*, *i*, *o*, *u*, *x* és *X* konverziós karakterrel együtt megadva a **long int** (bizonyos változatokban a 64-bites **__int64** egész) típust jelöli.

Nézzünk néhány példát a különböző formátum-megadási módokra! Az alábbi táblázatban a "*Lafenita*" karaktersorozat kiírásának eredményét foglaltuk össze. A kiíráshoz használt formátum-előírást a bal oldali oszlop tartalmazza, míg a nyomtatási kép a jobb oldali oszlopban tanulmányozható (a | karakterek a mezőhatárt jelölik):

%s	|Lafenita|
%7s	|Lafenita|
%.7s	|Lafenit|
%23s	| Lafenita|
%-23s	|Lafenita |
%23.7s	| Lafenit|
%-23.7s	|Lafenit |

Az egész számok decimális, oktális és hexadecimális formátumban történő kiírását a következő táblázat tartalmazza.

	? = d	**? = o**	**? = x**	**? = X**
%?	|1223|	|2307|	|4c7|	|4C7|
%2?	|1223|	|2307|	|4c7|	|4C7|
%6?	| 1223|	| 2307|	| 4c7|	| 4C7|
%-6?	|1223 |	|2307 |	|4c7 |	|4C7 |
%.7?	|0001223|	|0002307|	|00004c7|	|00004C7|
%#?	|1223|	|02307|	|0x4c7|	|0X4C7|

A formátumban a kérdőjel (*?*) helyén a táblázat felső sorában megadott konverziós betűket kell behelyettesíteni (a | karakterek a mezőhatárt jelölik):

Végezetül nézzünk néhány példát **double** típusú lebegőpontos értékek konverziójára!

	7.12	**-23.0**	**0.0000023**
%lf	|7.120000|	|-23.000000|	|0.000002|
%e	|7.120000e+00|	|-2.300000e+01|	|2.300000e-06|
%g	|7.12|	|-23|	|2.3e-06|
%10.1lf	| 7.1|	| -2.30|	| 0.0|
%+5.0E	|+7E+00|	|-2E+01|	|+2E-06|
%+5.1E	|+7.1E+00|	|-2.3E+01|	|+2.3E-06|
%+7.2E	|+7.12E+00|	|-2.30E+01|	|+2.30E-06|

A scanf() függvény

Az

```
int scanf(const char *formatum, ...);
```

deklarációjú függvény formázott adatbevitelt valósít meg. A ***scanf()*** függvény a szabványos beviteli eszközről olvas be adatokat, és az esetek többségében a ***printf()*** konverziók fordítottját hajtja végre. Hívásának általános alakja:

```
scanf(formátum, argumentumlista);
```

A ***scanf()*** függvény karaktereket olvas a szabványos adatbeviteli eszközről, majd a formátumsztring specifikációi szerint értelmezi, és konvertálja azokat. A művelet eredményét a soron következő argumentum által kijelölt memória-területen tárolja.

A ***scanf()*** befejezi az olvasást, ha a specifikációnak megfelelő számú adatot már feldolgozott, vagy ha az adatbevitel valamely oknál fogva nem felel meg a formátumsztring előírásainak. A ***scanf()*** visszatérési értéke a sikeresen beolvasott adatok száma, ami jól használható arra, hogy ellenőrizzük, hogy valóban annyi adatot olvasott-e be a programunk, mint ahányat előírtunk. A *0* visszatérési érték azt jelöli, hogy egyetlen változónk sem kapott értéket a ***scanf()*** hívása során.

A formátumsztring felépítése hasonlít a ***printf()*** függvénynél használt formátumra, azonban több ponton is különbözik attól. A ***scanf()*** formátumsztring az alábbi elemekből épül fel:

- tagoló (*whitespace*) karakterek, melyeket a ***scanf()*** figyelmen kívül hagy,
- konverziós előírások a % karakterrel kezdődnek, és opcionálisan tartalmazhatják a:

*	csillag karaktert (melynek hatására a beolvasott adatot a ***scanf()*** eldobja),
n	mezőszélességet megadó decimális egész számot,
h, l, L	opcionális méretkijelölő karaktert: *h* (**short**), *l* (**long** vagy **double**), *L* (**long double**).

A specifikációt a konverziós karakter zárja, amelynek lehetséges értékeit az alábbi táblázatban foglaltuk össze.

Konverziós karakter	***Input adat***	***Argumentum típus***
d **D, (ld)**	Decimális egész.	**int** * **long** *
i **I, (li)**	Egész szám, akár oktális (0), akár hexadecimális (0x vagy 0X) formában.	**int*** **long** *
o **O, (lo)**	Oktális egész a 0 előtag nélkül.	**int*** **long** *
u **U, (lu)**	Előjel nélküli decimális egész.	**unsigned*** **unsigned long** *
x **X, (lx)**	Hexadecimális egész, akár a 0x vagy 0X előtaggal, akár anélkül.	**int*** **long** *
C	Karakter(ek). A soron következő karakter (alapértelmezésben csak egy) kerül a megfelelő memóriaterületre. A tagoló karakterek is beolvasásra kerülnek. A következő nem tagoló karakter olvasásához a %1s specifikációt használjuk.	**char***
S	Sztring olvasása tagoló karakterig. A specifikációhoz tartozó pointer egy olyan tömbre kell mutasson, amely elegendően nagy a beolvasandó karakterek és a sztring végét jelző EOS tárolására.	**char***
f, e, g, **E, G**	E három vezérlőkarakter szolgál lebegőpontos számok beolvasására. Az előjel, a tizedespont és az exponens megadása opcionális. Ha a specifikációban az l előtag is szerepel (például *%lf*), akkor az argumentum típusa: Az L előtag esetén az argumentum típusa pedig:	**float*** **double** * **long double** *
P	A **printf** által kiírt formátumú mutató értékét olvassa be.	**void** **
n	A formátumig beolvasott karakterek számát adja vissza a függvény az argumentummal kijelölt egész (**int**) típusú változóban.	**int** *

Ha az adatbevitel során a százalékjelet kell megadnunk, akkor a formátumsztringben a %% elemet kell szerepeltetni. Az alábbiakban a c és az n konverziós karakterek együttes alkalmazására mutatunk példát. A program 10 tetszőleges karaktert tartalmazó karaktersorozatot olvas be:

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

void main() {
  int a;
  char s[20];
  scanf("%10c%n", s, &a);      // az a értéke mindig 10 lesz
  s[a]=0;
  cout<<s<<endl;
  cin.sync_with_stdio(false); // nincs szinkronizálás
  cin.get();
}
```

A *scanf()* szűrni is képes a beolvasandó karaktersorozat karaktereit. Ha a százalékjel után álló szögletes zárójelben karakterhalmazt adunk meg, akkor csak az ott szereplő karakterek kerülnek be a sztringbe (szűrés).

```
char str[80];
scanf("%[INin]", str);
```

csak a felsorolt karaktereket tartalmazó karaktersorozatot olvassa be az inputsorból (a példa csak kis- és nagy I, N betűket tartalmazó sztringet olvassa be);

```
scanf("%[0-9a-fA-F]", str);
```

csak a megadott intervallumba eső karaktereket olvassa be (a példa csak hexadecimális jegyeket tartalmazó sztringet fogad el);

```
scanf("%[^abcd]", str);
```

csak azokat a karaktereket olvassa be, amelyek nem szerepelnek a felsorolásban;

```
scanf("%[^0-9]", str);
```

csak a megadott karakter intervallumon kívülre eső karaktereket olvassa be (a példa minden számjegyet nem tartalmazó sztringet elfogad). Természetesen a karakterhalmaz kijelölésénél a fenti megoldásokat együttesen is megadhatjuk.

F2.1.4. Írás sztringbe és olvasás sztringből

C++ nyelven adott a lehetőség olyan formázott adatátviteli műveletek végzésére, amelyek karaktersorozatba írnak, illetve karaktersorozatból olvasnak. A sztringbe való formázott kiíráshoz az *sprintf()* függvényt használjuk, melynek prototípusa:

```
int sprintf(char* puffer, const char* formatum, ... );
```

A sztringből való olvasás művelete az *sscanf()* függvénnyel végezhető el:

```
int sscanf(const char* puffer, const char* formatum, ...);
```

A fenti két függvény jól használható különböző típusú számok és szövegek közötti konverzió elvégzésére.

A függvények használatát a STRINGIO.C program szemlélteti, ahol a sztringtömbből olvassuk a **double** értékeket, amiket kerekített **long** típusú értékként írunk vissza a sztringtömbbe:

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

void main() {
   char adat[3][16] = { "22.7","12.23","1979.12"};
   double e;
   int i;

   for (i=0; i<3; i++) {
       sscanf(adat[i],"%lf",&e);
       e +=0.5;      // kerekítés
       sprintf(adat[i], "%04ld", long(e));
   }

   for (i=0; i<3; i++)
       printf("%d. \t %s\n",i ,adat[i]);

   cin.sync_with_stdio(false);
   cin.get();
}
```

Eredmény:

```
0.      0023
1.      0012
2.      1979
```

F2.1.5. Az stdio és stdout adatfolyamok átirányítása

A legtöbb operációs rendszer (a Unix-hoz hasonlóan) támogatja a szabványos be- és kimenetek átirányítását. A C++ programunkon belül semmit sem kell tennünk az átirányítás megvalósítása érdekében. Tekintsük újra a *chario.cpp* példaprogramot, melynek készítsük el a futtatható változatát (*chario.exe*)!

```
// chario.cpp
#include <cstdio>
using namespace std;

void main() {
   int c;
   while ((c = getchar()) != EOF)
      putchar(c);
}
```

Ezek után parancssor készenléti jelénél futtatva a *chario.exe* programot, használjuk az operációs rendszer adatátirányító operátorait (<, >, >>):

`CHARIO`
Indítás után a begépelt karaktersort az *<Enter>* lenyomása után visszaírja a képernyőre a program. A kilépéshez a *<Ctrl+Z>* billentyűt kell megnyomnunk.

`CHARIO >ADAT.TXT`
A begépelt karakterek az *adat.txt* szöveges állományba másolódnak. A kilépéshez a *<Ctrl+Z>* billentyűt kell megnyomnunk. (Az ***stdout*** átirányítása az *adat.txt* állományba.)

`CHARIO <ADAT.TXT`
A program megjeleníti a képernyőn az *adat.txt* szövegfájl tartalmát. (Az ***stdin*** átirányítása az *adat.txt* állományból.)

`CHARIO <ADAT.TXT >ADAT2.TXT`
A program az *adat.txt* szöveges állomány tartalmát átmásolja az *adat2.txt* fájlba. (Az ***stdin*** az *adat.txt* állományt, míg az ***stdout*** *adat2.txt* állományt jelenti.)

Az átirányítás segítségével interaktív programot is lehet futtatni állományban tárolt adatokkal, illetve a program kimenetét fájlba vagy nyomtatóra lehet küldeni.

F2.2. A C++ alapvető I/O műveletei (iostream)

Mint a fejezet bevezetőjében említettük, a C++ nyelv objektumokkal támogatja az input/output műveletek végzését. Az objektumok deklarációját az *iostream* fejállomány tartalmazza:

deklaráció	***leírás***
`extern ostream cout;`	szabványos adatkivitel (átirányítható),
`extern istream cin;`	szabványos adatbevitel (átirányítható),
`extern ostream cerr;`	szabványos hiba-adatfolyam,
`extern ostream clog;`	szabványos puffert használó hiba-adatfolyam.

Az objektumok eléréséhez az alábbi megoldások közül választhatunk:

a deklarációk elérése	*felhasználás*

`#include <iostream>`
`using namespace std;`
	`void main() {`
	`  int a;`
	`  cin>>a;`
	`  cin.get();`
	`  cout<<a<<endl;`
	`  cin.get();`
	`}`

`#include <iostream>`	`void main() {`
	`  int a;`
	`  std::cin>>a;`
	`  std::cin.get();`
	`  std::cout<<a<< std::endl;`
	`  std::cin.get();`
	`}`

Karakteres (**char**) adatfolyamokon végezhető műveletekhez a biteltoló operátorok átdefiniált változatai, illetve az objektumok tagfüggvényei közül egyaránt választhatunk.

F2.2.1. Az << és a >> műveletek

Az esetek többségében az adatbevitelt a ***cout*** objektum << operátorának, míg az adatkivitelt a **cin** objektum >> operátorának túlterhelt változataival végezzük. Az alábbi táblázatban összefoglaltuk, hogy mely C++ típusok ismeretére készítették fel az objektumok osztályait:

bool	**float**
char, signed char, unsigned char	**double**
short, unsigned short	**long double**
int, unsigned int	**char *, signed char *, unsigned char ***
long, unsigned long	**void ***

Felhívjuk a figyelmet arra, hogy karaktersorozatok beolvasása esetén az adatbevitel az első elválasztó karakternél (például szóköz) megszakad. Ezért írásjelekkel tagolt szövegsorok bevitelére tagfüggvényt kell használnunk, amely a bekért szöveg hosszát is ellenőrzi, illetve adott karakterig olvassa be a szöveget.

Nézzünk néhány megoldást a tagfüggvények alkalmazására!

Egyetlen karakter beolvasása:	`int ch=cin.get(); // vagy cin.get(ch);`
Egyetlen karakter kiírása:	`cout.put('N');`
Legfeljebb 12-karakteres szöveg beolvasása az első szóköz eléréséig:	`char s[13]; cin.get(s,13,' ');`
Legfeljebb 23-karakteres szöveg beolvasása az *<Enter>* billentyű leütéséig:	`char n[24]; cin.getline(n,24);`

Felhívjuk a figyelmet arra, hogy az inputpuffer fel nem dolgozott karakterei a következő művelet rendelkezésére állnak. A felesleges adatoktól például az alábbi hívással szabadulhatunk meg:

```
cin.ignore(79,'\n');
```

F2.2.2. I/O manipulátorok használata

A ***cin*** és a ***cout*** objektumok egy sor beállítási lehetőséggel rendelkeznek, melynek segítségével szabályozhatjuk a műveletek elvégzését. Az ún. formátumjelző bitek beállításához, illetve törléséhez az ***ios_base*** alaposztály **enum** konstansait, illetve tagfüggvényeit használjuk.

```
inline long ios::flags() const;
inline long ios::flags(long _l);
inline long ios::setf(long _f, long _m);
inline long ios::setf(long _l);
inline long ios::unsetf(long _l);
```

Mivel a bitek beállításánál, illetve törlésénél egyszerűbb megoldás is rendelkezésünkre áll ún. manipulátorok formájában, így a tárgyalásnál azokra helyezzük a hangsúlyt.

A manipulátorok olyan speciális függvények, melyek hívását a <<, illetve a >> operátorok operandusaként adjuk meg. Például, a *2003* hexadecimális formában *12* pozíción balra igazítva, az alábbi utasítással jeleníthető meg:

```
cout<<left<<hex<<setw(12)<<2003<<endl;
```

A paraméter nélküli manipulátorok deklarációját az *iostream*, míg a paraméterrel rendelkező manipulátorok leírását az *iomanip* fejállományok tartalmazzák. A fenti példa működéséhez mindkét deklarációs állományra szükségünk van:

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

Az alábbiakban áttekintjük az alkalmazható manipulátorokat. Érdekesség kedvéért – főleg haladó programozók részére – tagfüggvényekkel elvégezhető alternatív megoldást is mutatunk az egyes manipulátorok helyettesítésére. (Ezek többségében a formátumjelző biteket állítjuk.)

Paraméter nélküli manipulátorok

Az alábbi manipulátorokat párban használjuk, hatásuk a bekapcsolástól a kikapcsolásig terjed:

Manipulator	*Megoldás tagfüggvénnyel*	*Irány*	*Leírás (bekapcsolva)*
boolalpha ***noboolalpha***	`ios.setf(ios_base:: boolalpha);` `ios.unsetf(ios_base:: boolalpha);`	I/O	A logikai értékek megadhatók a *true* és a *false* szavakkal.
showbase ***noshowbase***	`os.setf(ios_base:: showbase);` `os.unsetf(ios_base:: showbase);`	O	A számok előtt a számrendszer jele (0, vagy 0x) is megjelenik.
showpoint ***noshowpoint***	`os.setf(ios_base:: showpoint);` `os.unsetf(ios_base:: showpoint);`	O	A tizedespont mindig megjelenik.
showpos ***noshowpos***	`os.setf(ios_base:: showpos);` `os.unsetf(ios_base:: showpos);`	O	A pozitív előjel is megjelenik.
skipws ***noskipws***	`is.setf(ios_base:: skipws);` `is.unsetf(ios_base:: skipws);`	I	Az elválasztó karakterek átlépése.
unitbuf ***nounitbuf***	`os.setf(ios_base:: unitbuf);` `os.unsetf(ios_base:: unitbuf);`	O	Automatikus pufferürítés minden művelet után.
uppercase ***nouppercase***	`os.setf(ios_base:: uppercase);` `os.unsetf(ios_base:: uppercase);`	O	A számok megjelenítésében az (*e* és az *x*) nagybetűk lesznek

Néhány adatfolyamot módosító manipulátor:

Manipulator	*Megoldás tagfüggvénnyel*	*Irány*	*Leírás*
endl	`os.put(os,'\n'); os.flush();`	O	Sorvége karakter.
ends	`os.put('\0');`	O	Sztringvége karakter.
flush	`os.flush();`	O	Pufferürítés
ws		I	Az elválasztó karakterek átlépése.

Az alábbi manipulátorok hatását az ugyanazon csoportban megadottal módosíthatjuk:

Manipulator	*Megoldás tagfüggvénnyel*	*Irány*	*Leírás*
A kiigazítás beállítása			
left	`os.setf(ios_base::left, ios_base::adjustfield);`	O	Balra igazít.
right	`os.setf(ios_base::right, ios_base::adjustfield);`	O	Jobbra igazít – alapértelmezés
internal	`os.setf(ios_base::internal, ios_base::adjustfield);`	O	A mezőben a lehető legjobban széthúzva jelennek meg az értékek.
A számrendszer beállítása			
dec	`os.setf(ios_base::dec, ios_base::basefield);`	I/O	Tízes - alapértelmezés
hex	`os.setf(ios_base::hex, ios_base::basefield);`	I/O	Tizenhatos.
oct	`os.setf(ios_base::oct, ios_base::basefield);`	I/O	Nyolcas.
A valós számok megjelenítése (hiányukban a fordító választ formátumot			
fixed	`os.setf(ios_base::fixed, ios_base::floatfield);`	O	Tizedes tört alakban.
scientific	`os.setf(ios_base:: scientific, ios_base::floatfield);`	O	Hatványkitevős formában.

Paraméterrel rendelkező manipulátorok

A paraméterrel rendelkező manipulátorok hatása általában csak a közvetlenül utánuk álló adatelemre terjed ki.

Manipulator	*Megoldás tagfüggvénnyel*	*Irány*	*Leírás*
`setiosflags(lFlags )`	`long ios::flags(long lFlags);` `long ios::flags() const;`	I/O	Jelzőbitek beállítása.
`resetiosflags(lFlags )`	`long ios::unsetf(long lFlags);`	I/O	Jelzőbitek törlése.
`setfill(nFill )`	`char ios::fill(char nFill);` `char ios::fill() const;`	O	Kitöltő-karakter
`setprecision(np)`	`int ios::precision(int np);` `int ios::precision() const;`	O	Tizedes jegyek
`setw(nw);`	`int ios::width(int nw);` `int ios::width() const;`	O	Mező-szélesség
`setbase(nb);`	`…; os.setf(maszk, ios_base::basefield);`	O	Szám-rendszer.

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

void main() {
   cout<<setfill('*');
   cout<<setw(23);
   cout<<setprecision(4);
   cout<<1.22379<<endl;
   cout<<1.22379<<endl;
   cin.get();
}
```

Futási eredmény:
********************1.224*
1.224

F2.2.3. I/O manipulátorok készítése

Saját manipulátorok készítését az alábbi példaprogrammal szemléltetjük:

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

// paraméter nélküli output manipulátor
ostream& tab(ostream& os) {
  os<<'\t';
  return os;
}

// paraméter nélküli input manipulátor
istream& urit(istream& is) {
  is.ignore(128,'\n');
  return is;
}

// tetszőleges saját manipulátor
class format {
  private:
     int w, p;
     friend ostream & operator<<(ostream& s, const format & m);
  public:
     format(int w=12, int p=7) : w(w), p(p) {}
};

ostream & operator<<(ostream& os, const format & m) {
  os.width(m.w);
  os.precision(m.p);
  os.setf(ios::fixed, ios::floatfield);
  return os;
}
```

```
void main() {
   int d;
   cout<<"Osszeg:";
   cin>>d>>urit;
   cout<<"Az osszeg: "<<tab<<d<<endl;

   long sv=cout.flags();  // a beállítások mentése
   double pi=3.14159265;
   cout<<"pi="<<tab<<format()<<pi<<endl;
   cout<<"pi="<<tab<<format(7,5)<<pi<<endl;

   cout.flags(sv);        // a beállítások visszatöltése
   cout<<"pi="<<pi<<endl;
   cin.get();
}
```

A program futásának eredménye:

```
Osszeg:1223
Az osszeg:       1223
pi=          3.1415927
pi=      3.14159
pi=3.1416
```

A példából jól látható, hogy a paraméter nélküli manipulátor valójában egy speciális szerkezetű függvény, melynek címe adódik át a << és >> operátorok operandusaként.

Ezzel szemben a paraméteres manipulátor egy olyan osztály, melynek példánya jelenik meg a << és >> operátorok operandusában.

F2.2.4. Írás sztringbe és olvasás sztringből

Az *strstream* fejállományban deklarált osztályok (***istrstream***, ***ostrstream***, ***strstream***) segítségével a karaktersorozatokat adatfolyamként használhatjuk.

Az adatfolyamot a karaktersorozathoz (*str*) a konstruktor segítségével rendeljük hozzá. Ha az adatbevitelre használható adatfolyam esetén nem adjuk meg a sztring hosszát, akkor annak nullával kell záródnia:

```
istrstream::istrstream(const char* str, int meret=0);
```

Az adatkiviteli *ostrstream* adatfolyam esetében a méretet mindig meg kell adni. Az opcionális harmadik paraméterrel eldönthetjük, hogy a sztring elejétől (*ios::out*) írjuk az adatokat, vagy pedig a tárolt karaktersorozat végéhez (*ios::app, ios::ate*) illesztjük azokat:

```
ostrstream::ostrstream (char * str, int meret, int mod=ios::out);
```

Az F.2.1.4. alfejezetben megoldott feladat objektumos változata:

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

void main() {
   char adat[3][16] = { "22.7","12.23","1979.12"};
   double e;
   for (int i=0; i<3; i++) {
       istrstream sin(adat[i]);
       sin>>e;
       e +=0.5;      // kerekítés
       ostrstream sout(adat[i], 16);
       sout<<setfill('0')<<setw(4)<<long(e)<<ends;
   }
   for (int i=0; i<3; i++)
       cout<<i<<". \t "<<adat[i]<<endl;
   cin.get();
}
```

F2.3. Állományok kezelése

Általános megfogalmazásban fájlok alatt a számítógép háttértárain (hajlékonylemez, merevlemez, mágnesszalag, CD stb.) tárolt, névvel ellátott adathalmazokat értünk. A fájlokat a bennük tárolt adatok elérése alapján csoportosíthatjuk. Azokat a fájlokat, amelyek adatelemeit csak sorban, egymás után érhetjük el, soros (szekvenciális) állományoknak nevezzük. A másik csoportot a közvetlen elérésű állományok alkotják. Ezekben a pozicionálás műveletét használva közvetlenül kiválaszthatjuk a feldolgozni kívánt adatelemeket.

F2.3.1. Állománytípusok

Az adatállományokat tartalmuk alapján szöveges és bináris fájlok csoportjára osztjuk. A szöveges állományokat általában soros fájlként használjuk, a bináris fájlok tartalmához pedig soros és közvetlen eléréssel egyaránt hozzáférhetünk.

F2.3.1.1. Szöveges állományok

A szöveges állományok általában olvasható információt tartalmaznak. A szövegfájlok karaktereket tartalmazó, különböző hosszúságú sorokból épülnek fel. Minden sort az *EOLN* (*End Of LiNe*) jel-, az egész állományt pedig az *EOF* (*End Of File*) jel zárja (ami hiányozhat is a fájl végéről).

jel	*vezérlőkarakterek*	*billentyűzetről*
EOLN	*13,10 (CR/LF)*	*<Enter>*
EOF	*26*	*<Ctrl+Z>*

A szöveges állományok feldolgozása során az *EOF* karakter beolvasását a C++ alkalmazás fájlvége eseményként érzékeli. Mivel a *Unix* operációs rendszer alatt a szöveges állományokban a sorokat egyetlen *NL* ('\n') karakter zárja, a C++ alkalmazások a *Windows* rendszerben az állományok olvasása és írása során konverziót hajtanak végre. Amikor a C++ program szöveges állományba ír adatokat, akkor az *NL* karaktert két karakterré (*CR/LF*) alakítja át, illetve olvasáskor a fájlban található *CR/LF* karaktereket egyetlen *NL* karakterre cseréli.

F2.3.1.2. Bináris állományok

A bináris fájlok bájtok sorozatát tárolják. Az állomány tartalmát tetszőleges sorszámú bájttól kezdve elérhetjük (a fájl első bájtjának sorszáma 0). Ellentétben a szöveges állományokkal, semelyik bájtérték sincs kitüntetve. (Természetesen a szövegfájlok mindig feldolgozhatók, mint bináris állományok.)

Alapvető különbség van a kétféle fájltípus között a különböző típusú számok tárolásában. Ha például a **short int** típusú 2003 számot szöveges állományba írjuk, akkor az olvasható formában, a *'2'*, *'0'*, *'0'*, *'3'* karaktersorozatként tárolódik. Ugyanezt a számot bináris fájlba írva, a szám a memóriában tárolt formában (2 bájton) kerül az állományba.

F2.3.2. Szabványos C/C++ fájlkezelés

A számítógépeken az operációs rendszer feladatai közé tartozik a fájlrendszer támogatása és a fájlok elérésének biztosítása. Az állományok tartalmának eléréséhez a programozási nyelvtől és az operációs rendszer fajtájától függetlenül mindig ugyanazokat a főbb lépéseket kell végrehajtani:

0. Előkészületek.
1. Az állomány megnyitása.
2. Az állomány tartalmának feldolgozása fájlműveletek (olvasás, írás, pozicionálás stb.) felhasználásával.
3. A szükséges műveletek elvégzése után az állomány lezárása.

A C nyelv állománykezelési lehetőségei közül csak a szabványos ún. adatfolyam-kezelést tekintjük át, ahol a fájlt egy ***FILE*** struktúrára mutató pointer azonosítja:

```
#include <cstdio>
using namespace std;
...
FILE *bfp;
```

A fájl megnyitását a ***fopen()*** függvénnyel végezhetjük el, az állomány nevének és megnyitási módjának megadásával. A *módsztring* (*mode*) betűivel kijelölhetjük a fájl típusát (*text*, ***b**inary*) és elérési módját (***r**ead*, ***w**rite*, ***a**ppend*). Amennyiben írni és olvasni is szeretnénk az állományt megnyitása után, az *r+*, *w+* és *a+* jeleket kell használnunk.

```
FILE *fopen(const char *filename, const char * mode);
```

Ha a függvény által visszaadott érték nem *NULL*, használhatjuk a szokásos fájlműveleteket: írás, olvasás, pozicionálás. Minden műveletre függvények állnak rendelkezésünkre a szabványos függvénytárban:

Fájlművelet	***Függvény***
karakter írása	`int fputc(int c, FILE *stream);`
karakter olvasása	`int fgetc(FILE *stream);`
c-sztring írása	`int fputs(const char *s, FILE *stream);`
c-sztring olvasása	`char *fgets(char *s, int n, FILE *stream);`
formázott írás	`int fprintf(FILE *stream, const char *format[, argument, ...]);`
formázott olvasás	`int fscanf(FILE *stream, const char *format[, address, ...]);`
memóriaterület írása	`size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream)`
memóriaterület olvasása	`size_t fread (void *ptr, size_t size, size_t n, FILE *stream);`
pozicionálás	`int fseek(FILE *stream, long offset, int whence);`
pozíció lekérdezése	`long int ftell(FILE *stream);`
fájlvége vizsgálata	`int feof(FILE *stream);`
hibavizsgálat a műveletek után	`int ferror(FILE *stream);`

Munkánk végeztével le kell zárni az állományt, különben adatvesztés léphet fel:

```
int fclose(FILE *stream);
```

Az írási műveletek után, a lezárást megelőzően ajánlott az adatátviteli puffer tartalmát a fájlba üríteni:

```
int fflush(FILE *stream);
```

F2.3.3. A szabványos C++ állománykezelés

Az objektum-orientált fájlkezelés alapját a szabványos sablonkönyvtár általánosított adatfolyam-osztályai (*stream*) képezik, melyek karakteralapú megvalósításait használjuk a fejezetben. Az osztályok deklarációinak eléréséhez az alábbi sorokat kell a programunkban elhelyezni:

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

A fájlkezelés során használható adatfolyam-osztályokat táblázatban foglaltuk össze:

Általánosított osztály	*Karakteralapú megvalósítás*	*Ismertető*
basic_ios	*ios*	alaposztály
basic_istream	*istream*	input alaposztály
basic_ostream	*ostream*	output alaposztály
basic_fstream	*fstream*	input/output fájl
basic_ifstream	*ifstream*	input fájl
basic_ofstream	*ofstream*	output fájl
basic_filebuf	*filebuf*	állomány-puffer

Az adatfolyam-osztályok alkalmazásának első lépése az objektumpéldány létrehozása és inicializálása. Már ebben a lépésben el kell döntenünk, hogy milyen irányú műveleteket kívánunk végezni: írni (***ofstream***), olvasni (***ifstream***) vagy mindkettőt (***fstream***). Az állomány megnyitása a konstruálás során is elvégezhető, de az ***open()*** tagfüggvény segítségével ezt később is megtehetjük. Alaphelyzetben adatfolyamok a szöveges fájlok kezelését támogatják. Amennyiben bináris állományt szeretnénk kezelni, meg kell adni az *ios::binary* konstanst a fájlnyitás során. Az elmondottak alapján az adatfolyam-objektum létrehozása és az állomány nyitása:

```
ifstream::ifstream();
void  ifstream::open(const  char*  szName,  int  nMode=ios::in,
                     int nProt = filebuf::openprot );
ifstream::ifstream(const  char*  szName,  int  nMode=ios::in,
                   int nProt = filebuf::openprot );
ofstream::ofstream();
void  ofstream::open(const  char*  szName,  int  nMode=ios::out,
                     int nProt = filebuf::openprot );
ofstream::ofstream(const  char*  szName,  int  nMode=ios::out,
                   int nProt = filebuf::openprot );
fstream::fstream ();
void fstream::open(const char* szName, int nMode,
                   int nProt=filebuf::openprot );
fstream::fstream(const char* szName, int nMode,
                 int nProt=filebuf::openprot );
```

A tagfüggvények argumentumaként az állomány nevét (*szName*), elérési módját (*nMode*) és védelmét (megosztását) (*nProt*) kell megadnunk. Az elérési módot az alábbi táblázatban szereplő konstansok felhasználásával, bitenkénti vagy (|) művelettel állíthatjuk elő:

Konstans	*ifstream*	*ofstream*	*fstream*	*Jelentés*
ios::app		•	•	hozzáírás a fájl végétől,
ios::ate		•	•	pozicionálás a fájl végére,
ios::in	•	•	•	nyitás olvasásra,
ios::out		•	•	nyitás írásra,
ios::trunc		•	•	régi fájltartalom eldobása,
ios::nocreate	•	•	•	csak létező fájlt nyit meg,
ios::noreplace		•	•	csak új fájlt nyit meg
ios::binary	•	•	•	bináris állomány.

Az *nProt* paraméter értéke, szintén előre definiált konstansokból, logikai vagy művelettel keletkezik (|).

Konstans	*Jelentés*
filebuf::openprot, *filebuf::sh_compat*	kompatibilis megosztási mód,
filebuf::sh_none	kizárólagos mód – nincs megosztás,
filebuf::sh_read	a megosztott olvasás engedélyezett,
filebuf::sh_write	a megosztott írás engedélyezett.

A megnyitási (konstruálási) művelet, illetve a későbbiekben az adatfolyam-műveletek sikerességét (sikertelenségét) különböző módon ellenőrizhetjük:

```
fstream stream ("Proba.dat",ios::in | ios::out);
if (stream)         cout<< "Sikeres!";
if (!stream)        cout<< "Sikertelen!";
if (stream.good())  cout<< "Sikeres!";
if (stream.bad())   cout<< "Hiba történt!";
if (stream.fail())  cout<< "Súlyos hiba történt!";
if (stream.eof())   cout<< "Fájlvége!";
stream.close();
```

Munkánk végeztével az adatfolyamot a ***close()*** tagfüggvény hívásával zárjuk.

Az adatfolyam sikeres megnyitása után különböző input-, illetve outputműveletek végzésére nyílik lehetőségünk. Az átdefiniált >> és << operátorok segítségével – alaptípusú adatokkal - csak szöveges adatforgalom valósítható meg. A bináris és a szöveges fájltartalomhoz tagfüggvények segítségével férhetünk hozzá. Az áttekintés során csak az alapműveletek bemutatására szorítkozunk.

Adatbeviteli műveletek végzése az istream tagfüggvényekkel

A adatbeviteli műveletek többségének több túlterhelt változata is létezik a különböző karaktertípusokhoz, melyek közül csak a **char** típushoz készített változat prototípusát közöljük.

Tagfüggvény	*Művelet*
`istream & operator >>alaptípusú_balérték;`	szöveges adatbeolvasás,
`int get();` `istream& get( char& rch );`	karakter (bájt) olvasása,

`istream& get( char* pch, int nCount,`
`char delim = '\n' );`
`istream& getline( char* pch, int nCount,`
`char delim = '\n' );`
	karaktersor (bájtsor) bevitele,

`istream& read( char* pch, int nCount );`	bájtsor beolvasása.

Szükség esetén pozícionálhatunk is az állományban az abszolút, illetve a (fájl elejéhez – *ios::beg*, aktuális pozíciójához – *ios::cur* vagy végéhez viszonyított – *ios::end*) relatív pozíció megadásával:

```
istream& seekg( streampos pos );
istream& seekg( streamoff off, ios::seek_dir dir );
```

Az aktuális adatfolyam olvasási pozícióját kérdezhetjük le a ***tellg()*** tagfüggvénnyel:

```
streampos tellg();
```

Néhány további speciális művelet:

Tagfüggvény	*Művelet*
`istream& ignore( int nCount = 1,` `int delim = EOF );`	nCount bájt átlépése,
`int peek();`	a következő bájt tesztelése,
`int gcount() const;`	az utolsó művelet során bevitt bájtok száma,
`istream& putback( char ch );`	bájt visszaírása az adatfolyamba.

Adatkiviteli műveletek végzése az ostream tagfüggvényekkel

Az adatkiviteli tagfüggvények csoportosítása az előző alfejezetben bemutatott műveletekéhez hasonlóan végezhető el.

Tagfüggvény	*Művelet*
`ostream& operator <<alaptípusú_jobbérték;`	szöveges adatkivitel,
`ostream& put( char ch );`	karakter (bájt) kiírása,
`ostream& write( const char* pch, int nCount );`	bájtsor kivitele.

Az írási pozíció lekérdezéséhez, illetve beállításához külön tagfüggvényeket használhatunk:

```
streampos tellp();

ostream& seekp( streampos pos );
ostream& seekp( streamoff off, ios::seek_dir dir );
```

Az adatkivitelre megnyitott adatfolyam lezárása előtt ajánlott üríteni az adatpuffereket:

```
ostream& flush();
```

F3. A C-könyvtár legfontosabb elemei

Mielőtt rátérnénk a C++ nyelv sablonkönyvtárának ismertetésére, röviden áttekintjük a szabványos C nyelvtől örökölt, leggyakrabban használt függvénycsoportokat.

F3.1. Karakterek osztályozása és átalakítása

A C++ nyelv könyvtára gazdag készletét tartalmazza az egyetlen karaktert feldolgozó makróknak. A karakterkezelő makrókat és függvényeket a *cctype* deklarációs állományon keresztül érjük el. A C++ nyelv a karaktereket két csoportba osztja:

- A nyomtatható karaktereket
  a számítógép meg tudja jeleníteni a képernyőn. A nyomtatható karakterek a szóköz (*0x20*) és a hullámvonal (~, *0x7E*) karakterek között helyezkednek el.
- Vezérlőkarakterek,
  melyek kódja a (0) és az (0x1F) intervallumba esik, de ide tartozik a Del (0x7F) karakter is.

A karaktertesztelő és karakter-átalakító függvények (makrók) **int** típusú paraméterrel rendelkeznek, melynek azonban csak az alsó bájtját használják. A széles karakterrel (**wchar_t**) működő változatok nevében az *is/to* után megjelenik a *w* betű, például: ***iswalnum()***, ***towlower()***.

`nt isalnum(int c);`	Az ***isalnum()*** makró visszatérési értéke nem nulla, ha a *c* értéke betű (*'A'-'Z'*, *'a'-'z'*) vagy számjegy (*'0'-'9'*).
`int isalpha(int c);`	Az ***isalpha()*** makró visszatérési értéke nem nulla, ha a *c* értéke betű (*'A'-'Z'*, *'a'-'z'*).
`int iscntrl(int c);`	Az ***iscntrl()*** makró visszatérési értéke nem nulla, ha a *c* értéke vezérlőkarakter.
`int isdigit(int c);`	Az ***isdigit()*** makró visszatérési értéke nem nulla, ha a *c* értéke számjegy ('0'-'9').
`int isgraph(int c);`	Az ***isgraph()*** makró visszatérési értéke nem nulla, ha a *c* nyomtatható karakter, de nem szóköz.
`int islower(int c);`	Az ***islower()*** makró visszatérési értéke nem nulla, ha a *c* értéke kisbetű (*'a'-'z'*).
`int isprint(int c);`	Az ***isprint()*** makró visszatérési értéke nem nulla, ha a *c* nyomtatható karakter.

`int ispunct(int c);`	Az ***ispunct()*** makró visszatérési értéke nem nulla, ha a *c* elválasztó karakter. Elválasztó karakternek számít az összes nyomtatható karakter a betűk, a számok és a szóköz nélkül.
`int isspace(int c);`	Az ***isspace()*** makró visszatérési értéke nem nulla, ha a *c* szóköz, kocsivissza, újsor, vízszintes és függőleges tabulátor vagy lapdobás karakter (*0x09-0x0D*, *0x20*).
`int isupper(int c);`	Az ***isupper()*** makró visszatérési értéke nem nulla, ha a *c* nagybetű (*'A'-'Z'*).
`int isxdigit(int c);`	Az ***isxdigit()*** makró visszatérési értéke nem nulla, ha a *c* hexadecimális számjegy (*'0'-'9'*, *'A'-'F'*, *'a'-'f'*).
`int tolower(int c);`	A ***tolower()*** függvény az angol nagybetűket kisbetűre alakítja. Visszatérési érték a *c* konvertált értéke.
`int toupper(int c);`	A ***toupper()*** függvény az angol kisbetűket nagybetűre alakítja. Visszatérési érték a *c* konvertált értéke.

F3.2. Konverziós függvények

Az alábbi könyvtári függvények a különböző típusú numerikus értékek és karaktersorozatok között végeznek átalakításokat. (Erre a célra természetesen a F2.1.4. fejezetben ismertetett ***sscanf()*** és ***sprintf()*** függvényeket is használhatjuk.) A konverziós függvények prototípusát a *cstdlib* fejállomány tartalmazza:

Karaktersorozat átalakítása numerikus értékké

`double atof(const char *s);`	A függvény sztringet lebegőpontos (**double**) értékké konvertál. Ha az átalakítás sikertelen, a visszatérési érték *0*.
`int atoi(const char *s);`	Karaktersorozatot egész számmá alakít. Ha a konvertálás sikertelen, a visszatérési érték *0*.
`long atol(const char *s);`	Sztringet hosszú egész számmá konvertál. Ha a konvertálás sikertelen, a visszatérési érték *0L*.
`double strtod(const char *s, char **endptr);` `double wcstod(const wchar_t *s, wchar_t **endptr);`	A függvények az *s* sztringet duplapontosságú számmá alakítják. A függvények visszatérési értéke a duplapontosságú szám. Túlcsordulás esetén a függvényérték a *HUGE_VAL*. Ha az átalakítás hibás, akkor az **endptr* tartalmazza a hibás karakterre mutató pointert.

```
long strtol(const char *s,
       char **endptr, int radix);

long wcstol(const wchar_t *s,
    wchar_t **endptr, int radix);
```

Az *s* sztringet **long** értékké alakítják, és visszaadják függvényértékként. A *radix* a konvertálás alapszámát határozza meg (*2-36*). Ha *radix 0*, akkor az *s* sztring első néhány karaktere ('*0*', '*0x*') határozza meg a konvertálás alapját. Ha az átalakítás hibás, az **endptr* tartalmazza a hibás karakterre mutató pointert.

```
unsigned long strtoul(
   const char *s,
   char **endptr, int radix);

unsigned long wcstoul(
   const wchar_t *s,
   wchar_t **endptr, int radix);
```

A függvények az *s* sztring tartalmát **unsigned long** értékké alakítják. A visszatérési érték **unsigned long** típusú szám. Ha az átalakítás során hiba lép fel, a **endptr* paraméter a hibás karakterre mutató pointert tartalmazza.

Numerikus érték alakítása karaktersorozattá

```
char* itoa(int value, char *string, int radix);
```

Az ***itoa()*** függvény egész számot sztringgé alakít. A *value* értékét *EOS* karakterrel lezárt sztringgé konvertálja a *string* mutatta tömbbe. A *radix* a konvertálás számrendszerét határozza meg (*2-36*). Ha a *value* negatív, és a *radix 10*, akkor előjelesen konvertál, egyébként előjel nélkül. A visszatérési érték a sztringre mutató pointer.

```
char *ltoa(long value, char *string, int radix);
```

Az ***ltoa()*** függvény a *value* hosszú egész számot karaktersorozattá alakítja a *string* mutatta tömbbe. A *radix* a konvertálás számrendszerét határozza meg (*2-36*). Ha a *value* negatív, és a *radix 10*, akkor előjelesen konvertál, egyébként előjel nélkül. A visszatérési érték a sztringre mutató pointer.

```
char *ultoa(unsigned long value, char *string, int radix);
```

Az ***ultoa()*** függvény az **unsigned long** típusú *value* értéket sztringgé konvertálja. A *radix* a konvertálás számrendszerét határozza meg (*2-36*).

```
char *ecvt(double value, int ndig, int *dec, int *sign);
```

Az ***ecvt()*** függvény a *value* lebegőpontos számot *ndig* számjegyet tartalmazó karaktersorozattá konvertálja a tizedespont és az előjel elhelyezése nélkül. A **dec* változóba a tizedespont pozícióját teszi, a **sign* változóba pedig nem 0 kerül, ha az érték negatív. A visszatérési érték a sztringre mutató pointer, ami egy statikus pufferre mutat, amit az ***fcvt()*** vagy ***ecvt()*** függvények következő hívása felülír.

```
char *fcvt(double value, int ndig, int *dec, int *sign);
```

Az ***fcvt()*** függvény a *value* lebegőpontos értéket *ndig* számjegyet tartalmazó sztringgé alakítja a tizedespont és az előjel elhelyezése nélkül. A **dec* változóba a tizedespont pozícióját teszi a rutin, a **sign* változóba pedig nem 0 kerül, ha az érték negatív. A visszatérési érték a sztringre mutató pointer, ami egy statikus pufferre mutat, amit az ***fcvt()*** vagy ***ecvt()*** következő hívása felülír.

```
char *gcvt(double value, int ndec, char *buf);
```

A függvény a lebegőpontos *value* értéket karaktersorozattá konvertálja tizedes tört alakban. A *value* számot, amennyiben lehetséges, *ndec* értékes számjegyet tartalmazó fixpontos (*FORTRAN F* formátumú) alakban kapjuk meg, egyébként lebegőpontos (***printf()*** *E* formátumú) alakban. Az eredmény a *buf* mutatta pufferbe kerül, *EOS* karakterrel lezárva.

F3.3. Memória-területek (pufferek) kezelése

A puffer kifejezés alatt a memóriának adott címén kezdődő, adott hosszúságú területét értjük. A pufferek kezeléséhez szükséges függvények prototípusát a *cstring* állomány tartalmazza:

```
void *memchr(const void *s, int c, size_t n);
```

A ***memchr()*** függvény egy mutatót ad vissza, amely a *c*-ben adott karakter első előfordulásának helyét jelzi az *s* pufferben. (*NULL* jelzi, ha a karakter nincs benne.)

```
int memcmp(const void *s1, const void *s2, size_t n);
```

A ***memcmp()*** függvény összehasonlítja az *s1* és *s2* puffer első *n* bájtját. A visszatérési érték negatív, ha *s1<s2*, nulla, ha *s1==s2* és pozitív, ha *s1>s2*.

```
void *memcpy(void *dest, const void *src, size_t n);
```

A ***memcpy()*** függvény az *src* pufferből *n* bájtnyi blokkot átmásol a *dest* területre. Átfedésben levő területek esetén nem alkalmazható!

```
void *memmove(void *dest, const void *src, size_t n);
```

A ***memmove()*** függvény az *src* pufferből *n* bájtot átmozgat a *dest* területre. Egymást átfedő pufferek esetén is jól működik.

```
void *memset(void *s, int c, size_t n);
```

A ***memset()*** függvény az *s* puffer első *n* bájtját feltölti a *c* karakterrel, és *s* címével tér vissza.

F3.4. A dinamikus memória-kezelés függvényei

A dinamikus memória-kezelés elvégzéséhez szükséges függvények prototípusát a szabványos *cstdlib* állomány tartalmazza. Megjegyezzük, hogy a C++ alkalmazásokban a **new** és a **delete** műveletekkel végezzük el ezeket a feladatokat.

```
void *malloc(size_t m);
```
A ***malloc()*** függvény *m* bájt méretű memóriablokkot foglal le a halomterületen. A függvény visszatérési értéke az újonnan lefoglalt memóriablokkra mutató pointer

```
void *calloc(size_t n,
             size_t m);
```
A ***calloc()*** függvény először lefoglal egy *n*m* bájt méretű memória-területet, majd pedig feltölti *0* értékű bájtokkal. Visszatérési értéke szintén a memóriablokkra mutató pointer.

```
void *realloc(void *ptr,
              size_t m);
```
A ***realloc()*** függvény újrafoglalással módosítja a már lefoglalt memóriablokk méretét. A *ptr* paraméter által kijelölt memóriablokk méretét *m* bájtra növeli vagy csökkenti. Ha a blokkot növelni kell, akkor a régi blokk tartalmát átmásolja az új helyre. A visszatérési érték az újonnan lefoglalt memóriablokkra mutató pointer. (Ha a *ptr* értéke *NULL*, akkor ***malloc()*** függvényként működik.)

Mindhárom függvény *NULL* (nulla) mutató visszatérési értékkel jelzi, ha nem sikerült elvégezni a kijelölt műveletet, ezért a programunkban mindig gondoskodnunk kell a függvényérték vizsgálatáról.

Ha egy memóriablokkra nincs többé szükségünk, akkor azt fel kell szabadítanunk, átengedve annak területét a későbbi foglalási műveleteknek:

```
void free(void *ptr);
```
A **free** függvény felszabadítja a *ptr* mutató által kijelölt memóriablokkot, amelyet előzőleg a ***calloc()***, a ***malloc()*** vagy a ***realloc()*** hívások valamelyikével foglaltunk le

F3.5. Matematikai függvények

A szabványos C könyvtár különféle matematikai függvényeket tartalmaz, amelyek többsége **double** típusú paraméterrel és visszatérési értékkel rendelkezik. A függvények hibás működésüket a (*cerrno* állományban deklarált) ***errno*** változón keresztül jelzik:

EDOM	-	*Domain error*	-	Értelmezési tartomány hiba
ERANGE	-	*Range error*	-	Értékhatár átlépés
HUGE_VAL	-	*Overflow error*	-	Túlcsordulás

A matematikai függvények használata esetén a *cmath* deklarációs állományt szükséges a programunkba beépíteni:

Trigonometrikus függvények

```
double acos(double arg);
```

Az ***acos()*** függvény az *arg* árkusz koszinusz értékével tér vissza. Az argumentumnak -1 és 1 közé kell esnie, máskülönben „*domain error*" hiba lép fel. A függvényértékek a 0 és π között helyezkednek el.

```
double asin(double arg);
```

Az **asin()** függvény az argumentum árkusz szinusz értékével tér vissza. Az argumentumnak -1 és 1 között kell lennie, máskülönben „*domain error*" hiba keletkezik. A függvényértékek -π/2 és π/2 között helyezkednek el. Az **acos()** és **asin()** függvények hibás argumentum-megadás esetén 0 értékkel térnek vissza, és az ***errno*** hibaváltozó értéke *EDOM* lesz.

```
double atan(double arg);
```

Az **atan()** függvény az *arg* árkusz tangens értékével tér vissza. A függvényértékek a π/2 és π/2 közé esnek.

```
double atan2(double y, double x);
```

Az **atan2()** függvény az *y/x* kifejezés árkusz tangens értékével tér vissza. A függvény felhasználja az argumentum előjelét a visszatérési érték térnegyedének meghatározásához. A -π és π között szolgáltatott értékekek akkor is helyesek, amikor az eredményszög π/2, illetve -π/2 közelében található (*x* 0-hoz közel van). Ha az *x* és az *y* értéke *0*, akkor az ***errno*** változó értéke *EDOM* lesz.

```
double cos(double arg);
```

A **cos()** függvény az argumentum koszinusz értékével tér vissza. Az argumentum értékét radiánban kell megadni.

```
double sin(double arg);
```

A **sin()** függvény az *arg* szinusz értékével tér vissza. Az argumentum értékét radiánban kell megadni.

```
double tan(double arg);
```

A **tan**() függvény az *arg* tangens értékével tér vissza. Az argumentum értékét radiánban kell megadni. Ha az argumentum $\pi/2$ vagy $-\pi/2$ értékekhez közel van, a **tan** függvény 0 értékkel tér vissza, és az ***errno*** változó értéke *ERANGE* lesz.

Hiperbolikus függvények

`double cosh(double arg);`	A függvény az argumentum koszinusz hiperbolikus értékével tér vissza.
`double sinh(double arg);`	A függvény kiszámítja az argumentum szinusz hiperbolikus értékét.
`double tanh(double arg);`	A függvény megadja az argumentum tangens hiperbolikus értékét.

A ***cosh()*** és ***sinh()*** függvények túlcsordulás esetén *HUGE_VAL* értékkel térnek vissza (előjelhelyesen), és az ***errno*** felveszi az *ERANGE* értéket.

Hatvány- és logaritmusfüggvények

```
double exp(double arg);
```

Az ***exp()*** függvény az e^{arg} kifejezés értékével tér vissza. Túlcsordulás esetén a függvény értéke *HUGE_VAL*, míg alulcsordulás esetén *0.0* lesz. Mindkét esetet az ***errno*** változó *ERANGE* értéke jelzi.

```
double log(double num);
```

A ***log()*** függvény a *num* argumentum természetes alapú logaritmusával tér vissza. „*Domain error*" üzenetet kapunk, ha a *num* argumentum előjele negatív, és „*Range error*" hibaüzenetet kapunk, ha az argumentum zérus.

```
double log10(double num);
```

A ***log10()*** függvény kiszámolja az argumentum tízes alapú logaritmusát. Ha a *num* értéke *0* vagy negatív, akkor mindkét logaritmusfüggvény esetén az ***errno*** változó értéke *EDOM* lesz.

```
double pow(double x, double y);
```

A ***pow()*** függvény az *x* argumentum *y* hatványkitevőre emelt értékével (x^y) tér vissza. Túlcsordulás esetén a függvény értéke *HUGE_VAL*, és az ***errno*** változó értéke *ERANGE*. Ha az $x \leq 0$ és *y* nem egész, akkor *EDOM* hiba keletkezik, és a függvény értéke *HUGE_VAL* lesz. Ha *x* és *y* értéke *0* (0^0) a függvény értéke *1*.

```
double sqrt(double num);
```

Az ***sqrt()*** függvény a *num* négyzetgyökének értékével tér vissza. Ha az argumentum negatív előjelű, akkor az ***errno*** változó értéke *EDOM* lesz, és *0.0* értékkel tér vissza a függvény.

```
double ldexp(double x, int exp);
```

Az ***ldexp()*** függvény kiszámítja az $x \cdot e^{exp}$ kifejezés értékét.

```
double frexp(double x, int *n);
```

A függvény meghatározza az *x* valós szám $m \cdot e^n$ alakú előállítását, ahol a mantissza (*m*) a függvény értéke ($0{,}5 \leq m < 1$), *n* pedig a kitevő.

Egyéb cmath függvények

`double fabs(double x);`	Abszolút érték számítása.
`double ceil(double num);`	A függvény a legkisebb olyan egész számmal tér vissza, amelyik nem kisebb a *num* értékénél. Például *ceil(2.15)* → *3.0* és *ceil(-2.15)* → *-2.0* értéket ad vissza.
`double floor(double num);`	A függvény a legnagyobb olyan egész számmal tér vissza, amely nem nagyobb *num* értékénél. Például *floor(2.15)* esetén *2.0* és *floor(-2.15)* esetén *-3.0* lesz a visszatérési érték.
`double fmod(double x, double y);`	A függvény az *x* *modulo* *y*-t (az *x/y* osztás maradékát) adja vissza. A függvény az $x = iy+f$ kifejezést állítja elő, ahol az *i* egész szám, az *f* ($0 \leq f < y$) pedig a függvény által visszaadott maradék.
`double modf(double x, int *ipart);`	A függvény az *x*-et szétvágja két részre: egy egészre, amelyet az *ipart*-ban tárol, és a törtrészre, amit függvényértékként ad vissza.

A cstdlib matematikai függvényei

`int abs(int num);`	A függvény az **int** típusú *num* abszolút értékével tér vissza.
`long labs(long num);`	A függvény a **long** típusú argumentum abszolút értékével tér vissza.
`div_t div(int x, int y);`	A függvény elosztja az argumentumként megadott két egész számot, és a ***div_t*** típusú struktúrában visszaadja a hányadost (*quot*) és a maradékot (*rem*).
`ldiv_t ldiv(long int x, long int y);`	A függvény elosztja az argumentumként megadott két hosszú egészet, és a ***ldiv_t*** típusú struktúrában visszaadja a hányadost (*quot*) és a maradékot (*rem*).
`int rand(void);`	A függvény álvéletlen számot ad vissza 0 és *RAND_MAX* (*0x7FFFU*) között.
`void srand(unsigned seed);`	Az ***srand()*** függvény a pszeudovéletlen számok sorozatát inicializálja. Argumentum nélküli hívás esetén 1 lesz a kezdőérték, egyébként pedig a *seed* paraméter segítségével adhatjuk meg a kívánt értéket, mint például: `srand((unsigned) time(NULL));`

F3.6. Dátum- és időkezelő függvények

A dátum és az idő kezeléséhez a *ctime* deklarációs állományt kell beépíteni a programunkba. Az időkezelő függvények többsége a ***tm*** struktúrát használja:

```
struct tm {
  int    tm_sec;      // másodperc
  int    tm_min;      // perc
  int    tm_hour;     // óra (0÷23)
  int    tm_mday;     // a hónap napja (1÷31)
  int    tm_mon;      // hó (0÷11)
  int    tm_year;     // év (a naptári év - 1900)
  int    tm_wday;     // a hét napja (0÷6; vasárnap = 0)
  int    tm_yday;     // az év napja (0÷365)
  int    tm_isdst;    // 0 ha nincs nyári időszámítás
};

char *asctime(const struct tm *tblock);
```

A **asctime***()* függvény 26-karakteres karaktersorozatban adja vissza a **tblock* struktúrában tárolt dátumot és időt (a 26. karakter az *EOS*). A sztringet a következő **asctime** vagy **ctime** hívás felülírja. A sztring felépítésére: *Sun Dec 23 07:12:23 1979\n\0.*

```
clock_t clock(void);
```

A ***clock()*** függvény segítségével két esemény közötti időintervallum határozható meg. Ha az időértékeket másodpercben kívánjuk megkapni, a visszaadott értéket el kell osztani a *CLK_TCK* konstanssal.

```
char *ctime(const time_t *time);
```

A ***ctime()*** függvény a dátumot és az időt karaktersorozattá alakítja, majd visszatér a 26-karakteres sztringre mutató pointerrel. A **time* paraméter értéke a ***time()*** függvény hívásával állítható be.

```
double difftime(time_t time2, time_t time1);
```

A ***difftime()*** függvény a megadott két ***time_t*** típusú időpont különbségét határozza meg másodpercekben.

```
struct tm *gmtime(const time_t *timer);
```

A ***gmtime()*** függvény az aktuális dátumot és az időpontot Greenwich-i idővé (*GMT*) alakítja, és egy **struct *tm*** típusú struktúrába tölti. (Ez a statikus struktúra minden híváskor felülíródik.)

```
struct tm *localtime(const time_t *timer);
```

A ***localtime()*** függvény a (helyi) *time_t* típusú dátumot és az időpontot egy **struct *tm*** típusú struktúrába tölti. (Ez a statikus struktúra minden híváskor felülíródik.)

```
time_t mktime(struct tm *t);
```

Az ***mktime()*** függvény kitölti a ***tm*** struktúra hiányzó adattagjait, majd ***time_t*** formátumúra alakítja a struktúrában tárolt időt és dátumot.

```
time_t time(time_t *timer);
```

A ***time()*** függvény visszatérési értékként a nem *NULL timer* paraméterrel kijelölt tárterületen megadja az aktuális, az *1970. január 1. 00:00:00* óra *GMT* óta eltelt időt másodpercekben.

F3.7. Rendezés és keresés

Az alábbi függvények prototípusát a *cstdlib* fejállomány tartalmazza:

A quicksort algoritmust megvalósító sorbarendező függvény

```
void qsort(void *base, size_t nelem, size_t width,
           int(*fcmp)(const void *, const void *) );
```

A ***qsort()*** függvény gyorsrendezést hajt végre a *base* mutató által kijelölt tömbben. A függvény további paramétereinek jelentése:

nelem	az elemek száma a tömbben,
width	egy elem mérete byte-okban kifejezve,
Fcmp	az összehasonlító függvényre mutató pointer.

Bináris keresés elemek rendezett tömbjében

```
void * bsearch(const void *key, const void *base,
               size_t nelem, size_t width,
               int(*fcomp)(const void *, const void *));
```

A ***bsearch()*** függvény a **key* elemet keresi a rendezett *base* tömbben. A függvény visszatérési értéke az elsőnek megtalált elemre mutató pointer, vagy *NULL*, ha nem találta meg. A paraméterek jelentése megegyezik a ***qsort()*** függvény azonos nevű paramétereinek értelmezésével. Az összehasonlító függvény visszatérési értéke:

<0,	ha **elem1 < *elem2*,,
0,	ha **elem1 == *elem2*,
>0,	ha **elem1 > *elem2*.

F4. A Szabványos Sablonkönyvtár (STL) elemei

A *Hewlett-Packard Company* által fejlesztett Szabványos Sablonkönyvtár (STL, *Standard Template Library*) a C++ szabványos könyvtára, amely általánosított (sablon-) osztályokat és függvényeket tartalmaz a leggyakrabban használt adatstruktúrák és algoritmusok megvalósításaként. A könyvtár az általánosított (*generic*) programozást konténerekkel (tárolókkal), iterátorokkal (általánosított mutatókkal) és algoritmusokkal támogatja. Az alábbiakban rövid áttekintést adunk erről a három összetevőről. Az STL elemeit és az eléréshez szükséges deklarációs állományokat az alábbi táblázat tartalmazza:

Rövid leírás	***Fejállomány***
Adatsor-kezelés, rendezés, keresés stb. (*STL*)	<algorithm>
Általánosított numerikus műveletek	<numeric>
Asszociatív tároló: bithalmaz (***bitset***)	<bitset>
Asszociatív tároló: halmazok (elem-ismétlődéssel – ***multiset***, illetve ismétlődés nélkül - ***set***)	<set>
Asszociatív tároló: kulcs/érték adatpárok tárolása 1:1 (***map***), illetve 1:n (***multiset***) kapcsolatban (leképzések)	<map>
Funkció-objektumok	<functional>
Iterátorelemek, előre definiált iterátorok, adatfolyam-iterátorok	<iterator>
Memória-kezelés	<memory>
Műveleti elemek	<utility>
Tároló: dinamikus tömb (***vector***)	<vector>
Tároló: kettősvégű sor (***deque***)	<deque>
Tároló: lineáris lista (***list***)	<list>
Tároló: sor (***queue***), prioritásos sor (***priority_queue***)	<queue>
Tároló: verem (***statck***)	<stack>

F4.1. Tárolók (konténerek)

A tárolókat két fő csoportba sorolhatjuk: adatsorok (soros) és asszociatív tárolók. A soros tárolókra (vektorok, listák, sorok stb.) jellemző, hogy rendezettek, az elemek sorrendje által. Az asszociatív tárolók (leképzés, halmaz stb.) tulajdonsága, hogy az elemeket egy kulcs alapján lehet elérni.

A tárolóinterfészek általában az alábbi összetevőkből épülnek fel: konstruktorok (az alapértelmezett és a másoló konstruktor is), elem elérése, elem beszúrása, elem törlése, destruktor és iterátorok. A tárolókat iterátorok segítségével kezeljük. Az iterátor olyan „mutatószerű" objektum, amelyet az STL tárolókhoz hoztak létre. Példaként tekintsük az alábbi programot, ahol egy vektor elemeit függvény segítségével összegezzük!

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

double osszeg(const vector<double> & dv)
{
   vector<double>::const_iterator p; // konstans iterátor
   double s=0;

   for (p=dv.begin(); p!=dv.end(); p++)
      s += *p;
   return s;
}

void main()
{
   ostream_iterator<double>out(cout, " "); // kimeneti iterátor
   vector<double> v(4,0);
   v[0]=7; v[1]=12; v[2]=23; v[3]=79;
   copy(v.begin(), v.end(), out); // a tömb kiírása
   cout<<"Elemosszeg: "<<osszeg(v)<<endl;
   cin.get();
}
```

Az alábbi táblázatban összefoglaltuk a CC-vel jelölt konténerosztályok interfészeit:

Típus	*Leírás*
`CC::container_type`	a konténer típusa,
`CC::value_type`	a konténerben tárolt adatok típusa,
`CC::size_type`	a konténer méretének típusa,
`CC::reference`	a tárolt értékek referenciájának típusa,
`CC::const_reference`	konstans referencia
`CC::pointer`	mutató a referenciatípushoz,
`CC::iterator`	iterátor típus
`CC::const_iterator`	a konstans iterátor típusa
`CC::reverse_iterator`	a fordított iterátor típusa
`CC::const_ reverse_iterator`	a konstans fordított iterátor típusa
`CC::difference_type`	két CC::iterator típusú érték különbségének típusa

Minden tároló rendelkezik a fenti típusokkal. Például, egész elemű vektor *vector<**int**>* esetén az elemek típusa *vector<**int**>::value_type*, a bejáráshoz pedig a *vector<**int**>::iterator* típussal készíthetünk iterátort.

Az alábbi táblázat a leggyakrabban használt konténertagokat tartalmazza. (Míg *CC* a konténer osztályát jelölte, addig a *c* a konténerobjektum.

Tagfüggvény	***Leírás***
`CC::CC()`	alapértelmezett konstruktor,
`CC::CC(c)`	másoló konstruktor,
`c.begin()`	a *c* konténer első eleme (kezdete),
`c.end()`	a *c* tároló utolsó eleme utáni pozíció (vége),
`c.rbegin()`	kezdet a fordított iterátor számára,
`c.rend()`	vég a fordított iterátor számára,
`c.size()`	a konténerben tárolt elemek száma,
`c.max_size()`	a legnagyobb elemszám,
`c.empty()`	igaz, ha a tároló üres,
`c.swap(d)`	két tároló elemeinek felcserélése.

A konténerekhez műveleti jelek is tartoznak, amelyek *CC::value_type* típust használják: ==, !=, <, >, <=, >=.

Soros konténerek

Soros tárolók (***vector***, ***list*** és ***deque***) esetén az elemeket sorban, egymás után is elérhetjük. Az esetek többségében ide sorolható a C++ nyelv tömbtípusa is. Az alábbi táblázatban összefoglaltuk a soros (*sequence*) konténerek (*SEQ*) specifikus tagfüggvényeit:

Tagfüggvény	***Leírás***
`SEQ::SEQ(n, v)`	konstruálás *n* darab *v* értékű elemmel,
`SEQ::SEQ(b_it, e_it)`	konstruálás a megadott iterátorok közötti (*b_it*, *e_it -1*) elemek átmásolásával,
`c.insert(w_it, v)`	*v* beszúrása a *w_it* elé,
`c.insert(w_it,v, n)`	*n* darab *v* érték beszúrása a *w_it* elé,
`c.insert(w_it,b_it,e_it)`	*b_it* és *e_it* közötti elemek beszúrása *w_it* elé,
`c.erase(w_it)`	a w_it által kijelölt elem törlése,
`c.erase(b_it, e_it)`	a *b_it* és *e_it* közötti elemek törlése.

Asszociatív tárolók

Asszociatív tárolók (***set***, ***map***, ***multiset*** és ***multimap***) esetén az elemeket kulcs (*key*) segítségével érjük el. Ezek a konténerek rendelkeznek a ***Compare*** rendezési relációval, amely valójában egy összehasonlító objektum. Az asszociatív konténerek használatát az alábbi példa szemlélteti:

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

void main()
{
   map<string, int, less<string> > nev_kor;

   nev_kor["Aletta"] = 7;
   nev_kor["Lafenita"] = 23;
   nev_kor["Nata"] = 12;
   cout<<"Lafenita "<<nev_kor["Lafenita"]<<" eves."<<endl;
   cin.get();
}
```

Az asszociatív tárolók osztályainak (ASSOC) interfészei kibővítik a CC-vel jelölt konténerosztályok interfészeit:

Típus	*Leírás*
`ASSOC::key_type`	a kereső kulcs típusa,
`ASSOC::key_compare`	az összehasonlító objektum típusa,
`ASSOC::value_compare`	az *ASSOC::value_type* típusú értékek összehasonlításához használt típus.

Az asszociatív tárolók több szabványos konstruktorral is rendelkeznek. Az asszociatív konténerek konstruktora elsősorban az összehasonlító objektum használatában különbözik a soros tárolók konstruktorától.

Konstruktor	*Leírás*
`ASSOC()`	a *Compare* összehasonlító objektumot használó alapértelmezett konstruktor,
`ASSOC(cmp)`	a *cmp* összehasonlító objektumot használó konstruktor,
`ASSOC(b_it, e_it)`	konstruálás a *b_it* és *e_it* közötti elemekből a *Compare* felhasználásával,
`ASSOC(b_it, e_it, cmp)`	konstruálás a *b_it* és *e_it* közötti elemekből a *cmp* összehasonlító objektum felhasználásával,

A törlés és a beszúrás műveletek is kibővültek a kulcs alkalmazásának következtében:

Tagfüggvény	*Leírás*
`c.insert(t)`	*t* beszúrása, ha egyetlen elem sem rendelkezik *t* kulcsával; a visszaadott *pair<iterator, bool>* értékben a *bool* **true** értékű, ha a *t* nem volt benne,
`c.insert(w_it, t)`	*t* beszúrása a *w_it* kezdőpozíciótól; ***set*** és ***map*** esetén nem végzi el a műveletet, ha a kulcs már használt; a beszúrás pozíciójával tér vissza,
`c.insert(b_it, e_it)`	*b_it* és *e_it* közötti elemek beszúrása,
`c.erase(k)`	törli a *k* kulcsú elemeket, és visszaadja a törölt elemek számát,
`c.erase(w_it)`	a *w_it* által kijelölt elem törlése,
`c.erase(b_it, e_it)`	a *b_it* és *e_it* közötti elemek törlése.

További hasznos tagfüggvények:

Tagfüggvény	*Leírás*
`c.find(k)`	a *k* kulcsú elemre hivatkozó iterátorral tér vissza,
`c.count(k)`	megadja a *k* kulcsú elemek számát,
`c.lower_bound(k)`	megadja az első olyan elem iterátorát, amelyik kulcsa nagyobb vagy egyenlő, mint *k*,
`c.upper_bound(k)`	megadja az első olyan elem iterátorát, amelyik kulcsa kisebb vagy egyenlő, mint *k*,
`c.equal_range(k)`	visszaadja a (*lower_bound(k)*, *upper_bound(k)*) iterátorpárt.

Konténer adapterek

A konténer-adapter osztályok, olyan konténerosztályok, amelyek módosítják az alapvető tároló osztályokat, a létező megvalósítástól eltérő viselkedés megvalósítása érdekében. A támogatott konténer adapterek: ***stack***, ***queue***, ***priority_queue***.

A „*last-in, first-out*" működésű ***stack*** egyaránt adaptálható a ***vector***, a ***list*** és a ***deque*** tárolókból. Az adaptált ***stack***-függvényeket táblázatban foglaltuk össze:

Tagfüggvény	*Leírás*
`void push (const value_type& v)`	*v* bevitele a verembe,
`void pop ()`	a verem felső elemének levétele,
`value_type& top ()`	a verem felső elemének elérése,
`const value_type& top () const`	a verem felső elemének lekérdezése,
`bool empty () const`	**true** érték jelzi, ha a verem üres,
`size_type size ()const`	a veremben lévő elemek száma,
`operatror == és operator<`	egyenlőség és lexikografikus kisebb művelet.

Az alábbi példaprogramban adott számrendszerbe való átváltásra használjuk a vermet:

```
#include <iostream>
#include <stack>
#include <vector>
using namespace std;
void main() {
   int szam=2003, alap=16;
   stack<int, vector<int> > iverem;
   do {
     iverem.push(szam % alap);
     szam /= alap;
   } while (szam>0);

   while (!iverem.empty()) {
      szam = iverem.top();
      iverem.pop();
      cout<<(szam<10 ? char(szam+'0'): char(szam+'A'-10));
   }
   cin.get();
}
```

A *„first-in, first-out"* működésű sort (***queue***) ***list*** vagy ***deque*** tárolókból adaptálhatjuk. Az adaptált ***queue***-függvényeket az alábbi táblázat tartalmazza:

Tagfüggvény	*Leírás*
`void push(const value_type& v)`	*v* hozzákapcsolása a sor végéhez,
`void pop()`	a sor első elemének levétele,
`value_type& front()`	a sor első elemének elérése,
`const value_type& front() const`	a sor első elemének lekérdezése,
`value_type& back()`	a sor utolsó elemének elérése,
`const value_type& back() const`	a sor utolsó elemének lekérdezése,
`bool empty() const`	**true** értékkel jelzi, ha a sor üres,
`size_type size()const`	a sorban található elemek száma,
`operatror == és operator<`	egyenlőség és lexikografikus kisebb művelet.

A prioritásos sor (***priority_queue***) a ***vector*** vagy a ***deque*** tárolóból adaptálható. Szükségünk van egy összehasonlító objektumra is, hisz az elemek rendezetten helyezkednek el – a legfelső elem a legnagyobb. Az adaptált ***priority_queue***-függvényeket az alábbi táblázatban foglaltuk össze:

Tagfüggvény	*Leírás*
`void push (const value_type& v)`	*v* bevitele a prioritásos sorba,
`void pop ()`	a prioritásos sor felső elemének levétele,
`Const value_type& top () const`	a prioritásos sor felső elemének lekérdezése,
`bool empty () const`	**true** értékkel jelzi, ha a prioritásos sor üres,
`size_type size ()const`	megadja a prioritásos sorban található elemek számát,

F4.2. Iterátorok

A tárolókban iterátorok segítségével „közlekedhetünk". Az iterátorokat öt csoportba sorolhatjuk:

`istream_iterator`	*input*	bemeneti,
`ostream_iterator`	*output*	kimeneti,
`iterator`	*forward*	előre haladó (az előző kettő is ilyen),
`bidirectional_iterator`	*bidirectional*	kétirányú,
`random_access_iterator`	*random access*	közvetlen hozzáférésű.

Az istream_iterator és az ostream_iterator

A kimeneti és a bemeneti iterátorok segítségével a tároló elemei és az I/O adatfolyamok között létesítünk kapcsolatot.

```
#include <iostream>
#include <iterator>
#include <algorithm>
using namespace std;
void main() {
    double x[5]={7, 12, 23, 79, 157};
    ostream_iterator<double> out(cout,"\t");
    for (int i=0; i<5; i++)
      *out=x[i];
    cout<<endl;        // vagy
    copy(x, x+5, out);
    cout<<endl;
```

```
    istream_iterator<double, ptrdiff_t> in(cin);
    for (int i=0; i<5; i++) {
      cout<<"x["<<i<<"]=";
      x[i]=*++in;
    }
    copy(x, x+5, out);
    cin.ignore(100,'\n');
    cin.get();
}
```

A fenti példaprogramból jól láthatjuk az I/O iterátorok használatát. Az ***istream_iterator*** létrehozásakor a ***ptrdiff_t*** típussal jelezzük, hogy értelmezzük az iterátorok közötti távolság fogalmát, például léptetéshez. A bemeneti iterátort az értékadás jobb oldalán használjuk.

Az ***ostream_iterator*** létrehozásakor megadhatunk egy karaktersorozatot, amellyel az elemeket tagolhatjuk – a példában a tabulátort használtuk. A kimenetei iterátor az értékadás bal oldalán állhat.

Iterátor-adapterek

Az iterátorok adaptálásával, az alapiterátorokon túlmenően továbbiakat is használhatunk:

`reverse_iterator`	az elemek fordított sorrendben történő eléréséhez,
`__reverse_bi_iterator`	az elemek fordított, kétirányú sorrendben történő eléréséhez,
`insert_iterator`	beszúró iterátor segítségével az elem átírása helyett, beszúrjuk az új elemet,
`back_insert_iterator`	a végére beszúró iterátor,
`front_insert_iterator`	az elejére beszúró iterátor,

A fordított iterátorra példaként tekintsük az alábbi programot!

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

template <class ForwardIterator, class T>
void csere (ForwardIterator first, ForwardIterator last,
            const T& old_value, const T& new_value){
   while (first != last) {
      if (*first == old_value)
          *first = new_value;
      ++first;
   }
}
```

```
void main() {
    int x[5]={7, 12, 23, 79, 157};
    ostream_iterator<int> out(cout,"\t");
    vector<int> v(x, x+5);
    copy(v.begin(), v.end(), out);
    csere (v.begin(), v.end(), 7, 17);
    copy(v.begin(), v.end(), out);

    vector<int>::reverse_iterator p=v.rbegin();
    copy(p, v.rend(), out);
    cin.get();
}
```

F4.3. Algoritmusok

A különböző algoritmusokat függvénysablonok formájában tartalmazza az STL. Az algoritmusokat négy csoportba sorolhatjuk:

- elemsorrendet nem változtató algoritmusok,
- elemsorrendet megváltoztató algoritmusok,
- numerikus algoritmusok.

Az alábbiakban az egyes algoritmusoknak csak a nevét adjuk meg, hisz teljes ismertetésük külön könyvet igényelne.

Az elemsorrendet nem változtató algoritmusok:

```
accumulate                  includes
adjacent_find               lexicographical_compare
binary_search               lower_bound
count_min                   max
count_if                    max_element
equal                       min
equal_range                 min_element
find                        mismatch
find_end                    nth_element
find_first_of               search
find_if                     search_n
for_each
```

Az elemsorrendet megváltoztató algoritmusok:

```
copy                          remove_if
copy_backward                 replace
fill                          replace_copy
fill_n                        replace_copy_if
generate                      replace_if
generate_n                    reverse
inplace_merge                 reverse_copy
iter_swap                     rotate
make_heap                     rotate_copy
merge                         set_difference
nth_element                   set_symmetric_difference
next_permutation              set_intersection
partial_sort                  set_union
partial_sort_copy             sort
partition                     sort_heap
prev_permutation              stable_partition
push_heap                     stable_sort
pop_heap                      swap
random_shuffle                swap_ranges
remove                        transform
remove_copy                   unique
remove_copy_if                unique_copy
```

Numerikus algoritmusok

```
inner_product
partial_sum
adjacent_difference
```

Az alábbi példaprogramban a különböző algoritmusok tömbre való alkalmazását követhetjük nyomon:

```
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;
void main() {
    int adat[5]={5, 7, 12, 23, 46};
    ostream_iterator<int> out(cout,"\t");

    // az adatok véletlenszerű keverése:
    for (int i=0; i<7; i++) {
       random_shuffle(adat,adat+5);
       copy(adat,adat+5, out);
       cout<<endl;
    }
```

```
    // az adatok rendezése:
    sort(adat,adat+5);
    copy(adat,adat+5, out);
    cout<<endl;

    // adatok felcserélése:
    swap(adat[2],adat[4]);
    copy(adat,adat+5, out);
    cout<<endl;

    // a tömb feltöltése 23 értékekkel:
    fill(adat,adat+5, 23);
    copy(adat,adat+5, out);
    cout<<endl;
    cin.get();
}
```

F4.4. Művelet-objektumok (*functions*)

A különböző műveleteket lefedő művelet-objektumok definiálják az *operator()* függvényt. A művelet-objektumoknak fontos szerepük van az általánosított algoritmusok használatában, hiszen ezen algoritmusok interfésze vagy egy függvénymutatót, vagy pedig egy olyan objektumot fogad, amely definiálja az *operator()* függvényt. A (*functional*) könyvtárban megtalálható művelet-objektumokat három csoportba oszthatjuk:

Aritmetikai műveletek

`plus`	összeadás $x + y$
`minus`	kivonás $x - y$
`multiplies`	szorzás $x * y$
`divides`	osztás x / y
`modulus`	maradékképzés $x \% y$
`negate`	negálás $- x$

Relációs műveletek

`equal_to`	egyenlő $x == y$
`not_equal_to`	nem egyenlő $x != y$
`greater`	nagyobb $x > y$
`less`	kisebb $x < y$
`greater_equal`	nagyobb vagy egyenlő $x >= y$
`less_equal`	kisebb vagy egyenlő $x <= y$
`negate`	negálás $- x$

Logikai műveletek

`logical_and`	logikai és $x \&\& y$
`logical_or`	logikai vagy $x \| \| y$
`logical_not`	logikai tagadás $! x$

Az alábbi példában saját műveleti objektumot definiálunk a faktoriális-számítás elvégzésére:

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

template<class Arg>
class faktorialis : public unary_function<Arg, Arg> {
   public:
     Arg operator()(const Arg& arg) {
       Arg a = 1;
       for(Arg i = 2; i <= arg; a*=i++);
       return a;
     }
};

void main() {
   int init[5] = {3,5,7,11,12};
   deque<int> d(init, init+5);
   // üres vektor létrehozása
   vector<int> v((size_t)5);
   // a kétvégű sorban tárolt elemek átalakítása a factoriális
   // felhasználásával, és tárolása a vektorban
   transform(d.begin(), d.end(), v.begin(), faktorialis<int>());
   // megjelenítés
   ostream_iterator<int> out(cout,"\t");
   copy(d.begin(),d.end(), out);
   cout << endl;
   copy(v.begin(),v.end(),out);
   cin.get();
 }
```

Művelet-adapterek (function adaptors)

A művelet-adapterek segítségével művelet-objektumokat hozhatunk létre adaptáció révén.

`not1`	az átadott egyoperandusú predikátum tagadásával (!) tér vissza
`not2`	az átadott kétoperandusú predikátum tagadásával (!) tér vissza
`binder1st`	a művelet-objektum 1. operandusának kijelölése,
`binder2nd`	a művelet-objektum 2. operandusának megadása,
`ptr_fun`	mutató létrehozása egy-, illetve kétoperandusú művelet-objektumhoz.

Az alábbi példában a ***bind2nd*** adapter segítségével olyan összeadó művelet-objektumot készítünk, amely 23-at ad az első operandusához:

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

void main() {
   double adat[5] = {3, 5, 7, 11, 12};
   transform(adat, adat+5, adat, bind2nd(plus<double>(),23));
   ostream_iterator<double> out(cout,"\t");
   copy(adat, adat+5, out);
   cout << endl;
   cin.get();
}
```

F4.5. Allokátorok (*allocators*)

Az allokátor-objektumok (*memory*) a tárolók számára lefoglalt memória-területet kezelik. Az allakotátorok lehetővé teszik, hogy egy konkrét környezet feltételeihez igazodjunk, méghozzá úgy, hogy megőrizzük a konténerosztály interfészének hordozhatóságát.

Az allokátor-definíciókban megtalálhatjuk a ***value_type***, ***reference***, ***size_type***, ***pointer*** és a ***difference_type*** típusokat. A legfontosabb allokátor-tagfüggvényeket az alábbi táblázatban foglaltuk össze:

Tagfüggvény	*Leírás*

`allocator();`
`allocator(const allocator& a);`
`~allocator()`
	konstruktorok és a destruktor,

`pointer address (reference r) const`	*r* címével tér vissza,
`pointer allocate (size_type n);`	tárfoglalás *n* darab objektum számára,
`void deallocate(pointer p, size_type n);`	a lefoglalt memória felszabadítása,
`size_type max_size () const;`	a *difference_type* legnagyobb értékével tér vissza.

F5. Karaktersorozatok kezelése

A C++ nyelv könyvtárában a nullavégű karaktersorozatok kezelésére szolgáló C-függvényeket (*cstring*) és a sablonnal definiált string-osztályokat (*string*) egyaránt megtaláljuk. A fejezetben először áttekintjük a C-könyvtár ide tartozó függvényeit, majd *basic_string<**char**>* sablonpéldánnyal foglakozunk, melynek neve ***string***.

F5.1. A C-könyvtár függvényei

A C nyelvben a karaktersorozat (*string*) '\0' karakterrel záródó karaktertömböt jelöl. A sztringkezelő függvények prototípusát a *cstring* állomány tartalmazza. Ugyanitt találjuk a függvények széles karakterekkel (**wchar_t**) működő változatait. Példaként tekintsük az sztringmásoló függvény két változatának prototípusát!

```
char *strcpy(char *dest, const char *src);
wchar_t *wcscpy(wchar_t *dest, const wchar_t *src);
```

Az alábbiakban csak a **char** típushoz készített függvényeket tekintjük át:

```
char *strcat(char *str1, const char *str2);
```

Az ***strcat()*** függvény az *str1* karaktersorozathoz fűzi az *str2* sztring tartalmát, és az összefűzött karaktersorozatra mutató pointerrel tér vissza.

```
char *strchr(const char *str1, int c);
```

Az ***strchr()*** függvény megkeresi a *c*-ben tárolt karakter első előfordulásának helyét az *str1* karaktersorozatban, és erre a pozícióra mutató pointerrel tér vissza. (A *NULL* függvényérték azt jelzi, hogy a karakter nincs a sztringben.)

```
int strcmp(const char *str1, const char *str2);
```

Az ***strcmp()*** függvény karakterenként összehasonlít két karaktersorozatot. A visszatérési érték negatív, ha *str1<str2*, nulla, ha *str1==str2* és pozitív, ha *str1>str2*.

```
char *strcpy(char *str1, const char *str2);
```

Az ***strcpy()*** függvény az *str2* sztring karaktereit másolja az *str1* karaktersorozatba a záró *EOS* ('\0') karakterrel együtt.

```
size_t strcspn(const char *s1, const char *s2);
```

Az ***strcspn()*** függvény megadja, hogy az *str1* karaktersorozat első hány karaktere nem található meg az *str2* sztringben.

```
size_t strlen(const char *str);
```

Az ***strlen()*** függvény a karaktersorozat hosszát adja vissza. A visszatérési érték a karakterek számát tartalmazza, a sztringet záró *EOS* karakter nélkül.

```
char *strncat(const char *str1, const char *str2, size_t maxn);
```

Az ***strncat()*** függvény az *str2* karaktersorozatból *maxn* karaktert hozzáfűz az *str1* sztring végéhez.

```
int strncmp(const char *str1, const char *str2, size_t maxn);
```

Az ***strncmp()*** függvény az összehasonlításnál mindkét karaktersorozatból csak *maxn* számú karaktert vesz figyelembe. A visszatérési érték negatív, ha *str1<str2*, nulla, ha *str1==str2* és pozitív, ha *str1>str2*.

```
char *strncpy(const char *str1, const char *str2, size_t maxn);
```

Az ***strncpy()*** függvény az *str2* sztringből *maxn* karaktert másol az *str1* karaktersorozatba (a nulla karakter hozzáadása nélkül).

```
char * strpbrk (const char *str1, const char *str2);
```

Az ***strpbrk()*** függvény az *str1* karaktersorozatban az *str2* sztring bármely karakterének első előfordulását keresi, és az előfordulási helyre mutató pointerrel tér vissza. Ha az *str2* semelyik karaktere sem található meg az *str1*-ben, akkor *NULL* lesz a függvény értéke.

```
char *strrchr(char *str, int c);
```

Az ***strrchr()*** függvény az *str* sztringben a *c* karakter utolsó előfordulási helyét keresi, és az előfordulás helyére mutató pointerrel tér vissza. Ha a karakter nem található, akkor *NULL* lesz a függvény értéke.

```
size_t strspn(const char *s1, const char *s2);
```

Az ***strspn()*** függvény az *str1* sztring első olyan karakterének pozíciójával tér vissza, amely nem található meg az *str2* karaktersorozatban.

```
char *strstr(const char *str1, const char *str2);
```

Az ***strstr()*** függvény az str2 részsorozat első előfordulási helyét keresi az *str1* karaktersorozatban. A visszatérési érték az első előfordulási helyre mutató pointer. Ha az *str2* nem található meg az *str1*-ben, *NULL* értékkel tér vissza a függvény.

```
char *strtok(char *str1, const char *str2);
```

A ***strtok()*** függvény az *str2* karaktersorozatban megadott elválasztójelek alapján – többszöri hívással - feldarabolja az *str1* sztringet. (A függvény második hívásától kezdődően az első argumentumban *NULL* értéket kell megadni.) Minden visszatérési érték egy-egy részsorozatra mutató pointer (vagy *NULL*, ha az eredeti sztring nem darabolható tovább). Az alábbi példában az ***strtok()*** függvény alkalmazásával szavakra bontunk egy mondatot:

```
#include <cstring>
#include <iostream>
using namespace std;
void main(void) {
    char mondat[] = "A boldogsag egy pillango; ha kergetjuk,"
                    " kisiklik a kezunkbol... De ha csendben "
                    "leulunk, az is lehet, hogy leszall rank.";
    char irasjelek[]=" .,;:!?-";
    char *p;
    p = strtok(mondat, irasjelek);
    do {
      cout<<p<<endl;
      p = strtok(NULL, irasjelek);
    } while (p!=NULL);
    cin.get();
}
```

F5.2. A string osztály használata

Mint a bevezetőben említettük, a ***string*** osztály a ***basic_string*** sablonosztály **char** típussal előállított példánya. (A **wchar_t** típussal készített sablonpéldány neve ***wstring***.) A jobb érthetőség kedvéért, az osztálysablon elemei helyett, a megvalósított *string* osztály tagjait tekintjük át.

Konstruktorok

Egy sor konstruktor segíti a karaktersorozatok kényelmes definiálását:

Konstruktor	***Leírás***
`string();`	alapértelmezett, egy üres sztringet hoz létre,
`string(const char * p);`	konverziós konstruktor, amely c-karaktersorozatból állítja elő a sztringet,
`string(const vector<char> &v);`	konverziós konstruktor, amely konténerből állítja elő a sztringet,
`string(const string& s, size_t pos=0, size_t n = npos);`	másoló konstruktor – *npos* értéke általában –*1*, ami azt jelzi, hogy nem volt memória foglalva.
`string(const char *p ,size_t n);`	Konstruálás a *p* C-sztring első *n* karakteréből,
`string(const char c ,size_t n=1);`	*n* darab *c* karakterből álló sztringet készít,

Tagfüggvények

A ***string*** osztály túlterhelt operátorai:

Operátorfüggvények	*Leírás*
`string& operator=(const string& s);`	értékadás,
`string& operator=(const char* p);`	értékadás,
`string& operator=(char c)`	értékadás,
`string& operator+=(const string& s);`	hozzáfűzés,
`string& operator+=(const char* p);`	hozzáfűzés,
`string& operator+=(char c)`	hozzáfűzés,
`operator vector<char>() const;`	vektorrá konvertálás,
`char operator[](size_t pos) const;`	adott pozícióban lévő karakter,
`char& operator[](size_t pos);`	adott pozícióban lévő karakter referenciája.

A további tagfüggvényeknek is több átdefiniált változata létezik a ***string***, a ***char*** * és a **char** típusokhoz. A leírásban csupán a függvénynévre és a működés leírására szorítkozunk:

Tagfüggvények	*Leírás*
`append()`	hozzáfűzés,
`assign()`	értékadás,
`insert()`	beszúrás adott pozíciótól,
`remove()`	adott pozíciótól kezdődően, adott számú karakter törlése,
`replace()`	adott számú karakter adott pozíciótól kezdődő felülírása,
`get_at()`	adott pozíción levő karakter lekérdezése,
`put_at()`	karakter beírása adott pozícióra,
`length()`	a karaktersorozat hosszának lekérdezése,
`c_str()`	konvertálás **char** * típusú karaktersorozattá,
`data()`	a karaktersorozat kezdőcímének lekérdezése,
`rezise()`	a sztring átméretezése,
`copy()`	adott pozíciótól, adott számú karakter másolása **char** * karaktersorozatba,
`substr()`	adott pozíciótól, adott számú karakterből részsorozat készítése,
`compare()`	a tárolt sztring összehasonlítása egy másik karaktersorozattal,
`find()`	egy másik karakter/karaktersorozat keresése a tárolt sztringben,
`rfind()`	keresés visszafelé,
`find_first_of()`	első egyező előfordulás keresése,
`find_last_of()`	utolsó egyező előfordulás keresése,
`find_first_not_of()`	első nem egyező előfordulás keresése,
`find_last_not_of()`	utolsó nem egyező előfordulás keresése.

F5.3. Külső operátorfüggvények

A *sztring* osztályhoz tartozik egy sor, külső függvényként megvalósított operátor is.

Tagfüggvények	*Leírás*
`istream& operator>>(istream& is, string& str);`	adatbeviteli operátor,
`istream& getline(istream& is, string& str, char delim);`	adatbeviteli függvény,
`ostream& operator<<(ostream& os, const string& str);`	adatkiviteli operátor,
`string operator +(const string& s1, const string& s2);`	a megadott két sztring összefűzése,
`bool operator==(const string& s1, const string& s2);`	**true**, ha *s1* lexikografikusan azonos *s2*-vel,
`bool operator!=(const string& s1, const string& s2);`	**true**, ha *s1* lexikografikusan nem azonos *s2*-vel,
`bool operator< (const string& s1, const string& s2);`	**true**, ha *s1* lexikografikusan kisebb *s2*-nél,
`bool operator<=(const string& s1, const string& s2);`	**true**, ha *s1* lexikografikusan kisebb vagy egyenlő mint *s2*,
`bool operator> (const string& s1, const string& s2);`	**true**, ha *s1* lexikografikusan nagyobb *s2*-nél,
`bool operator>=(const string& s1, const string& s2);`	**true**, ha *s1* lexikografikusan nagyobb vagy egyenlő *s2*.

Az összehasonlító és az összefűző operátoroknak további túlterhelt változatai is léteznek a különböző típusokhoz:

```
string operator+(const string& s1, const string& s2);
string operator+(const char* cs, const string& s);
string operator+(char c, const string& s);
string operator+(const string& s, const char* cs);
string operator+(const string& s, char c);

bool operator==(const string& s1, const string& s2);
bool operator==(const char* cs, const string& s);
bool operator==(const string& s, const char* cs);
```

Végezetül nézzünk egy példaprogramot az elmondottak szemléltetésére!

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

void main() {
    string mondat, szavak[19];
    int poz=0, rpoz=0, nszo, i=0;

    mondat =  "A boldogsag egy pillango ha kergetjuk";
    mondat += " kisiklik a kezunkbol de ha csendben"
              " leulunk az is lehet hogy leszall rank";

    // felbontjuk a mondatot szavakra a szóközöknél vágva
    while (poz<mondat.size()) {
       poz = mondat.find(' ', rpoz);
       szavak[i++].assign(mondat, rpoz, poz-rpoz);
       cout<<szavak[i-1]<<endl;
       rpoz = poz+1;
    }

    nszo = i;     // a szavak száma
    // összerakjuk a mondatot szóhelyettesítéssel
    mondat="";
    for (i=0; i<nszo; i++)
       if (!szavak[i].compare("ha"))
          mondat+="amennyiben " ;
       else
          mondat+=szavak[i]+' ';
    mondat+='!';
    cout<<mondat<<endl;
    cout<<"Helyes? [igen/nem]";
    getline(cin, mondat,'\n');
    if (!mondat.compare("igen"))
       cout<<"Helyes!"<<endl;
    else
       cout<<"Nem tudom milyen."<<endl;
    cin.get();
}
```

Irodalomjegyzék

Bjarne Stroustrup

The C++ Programming Language
Special Edition
Addison-Wesley Publishing Co., 2000

Dirk Louis

C/C++, New Reference
Prentice Hall, 2001

Robert Robson

Using the STL
Springer-Verlag New York, Inc., 1988, 200

Victor Shtern

Core C++, A Software Engineer Approach
Prentice Hall Inc., 2000

Herbert Schildt

C/C++: Programmer's Reference
Second Edition
Osborne McGraw Hill, 2000

Johnsonbaugh, Richard

Applications Programming in C++
Prentice Hall Inc, 1999

Herbert Schild:

Teach Yourself C++
Third Edition
Osborne McGraw-Hill, 1998

Benkő Tiborné – Benkő Laszló – Poppe András

Objektum-orientált programozás C++ nyelven
ComputerBooks, 1998

Ira Pohl
Object-Oriented Programming Using C++
Second Edition
Addison-Wesley Longman, 1997

Timothy Budd
Data Structure in C++ Using the STL
Addison Wesley Longman Inc., 1997

Kris Jamsa
C++
Kossuth Kiadó, 1997

Ira Pohl
C++ Distilled
Addison-Wesley, 1997

Dr. Kondorosi Károly - Dr. László Zoltán - Dr. Szirmay-Kalos László
Objektum-orientált szoftverfejlesztés
C és C++ nyelven
ComputerBooks, 1997

P.J. Plauger
The Draft Standrd C++ Library
Prentice Hall, 1995

Stanley B. Lippman
C++ primer (2nd Edition)
Addison-Wesley Publishing Co., 1991

Brian W.Kernighan - Dennis M.Ritchie:
The C programming language (second edition)
Prentice Hall, Inc., New Jersey, 1988

Mark Williams Company:
ANSI C - A Lexical Guide
Prentice Hall, New Jersey, 1988

Tárgymutató

B

C

Cs

D

E,É

F

G

H

I,Í

J

K

L

M

N

Ny

O,Ó

Ö,Ő

P

Q

R

S

Sz

T

Ty

U,Ú

Ü,Ű

V

W

Megrendelőlap
info@computerbooks.hu
1253 Budapest Pf. 71

... *Dr.Kovács T.– Dr.Kovácsné C. J.– Ozsváth M.:*
Adatkezelés az MS ACCESS 2000 alkalmazásával **3.497.-**
... *Kovács L.:* **Adtatbázisok tervezésének és kezelésének módszertana** **4.200.-**
... *Mudri István:* **Digitális videózás Adobe Premiere 6.0**
alkalmazásával - Win/Mac **3.400.-**
... *Pintér M.:* **AutoCAD 2004 felhasználói ismeretek** **4.919.-**
... *Pintér Miklós:* **AutoCAD 2002 + LT, tankönyv és példatár**
1. kötet - síkbeli rajzok **3.900.-**
... *Pintér Miklós:* **AutoCAD 2002 tankönyv és példatár**
2. kötet - térbeli ábrázolás **4.500.-**
... *Móricz Attila:* **Tippek a CD íráshoz - ◉** **2.500.-**
... *Benkő L.–Benkő T.né–Tóth B.:* **Programozzunk C nyelven – ◉** **2.968.-**
... *Tóth B.:* **Programozzunk C++ nyelven ! ◉** **2.994.-**
... *Benkő T.né– Poppe A.*: **Objektum-orientált C++ – ◉**
- Együtt könnyebb a programozás - **3.217.-**
... *Kőhalmi Éva- Kőhalmi Mariann:* **CorelDraw 12** **5.590.-**
... *László József:* **Dinamikus weboldalak programozása - 💻** **4.500.-**
... *Kovalcsik Géza:* **EXCEL 2000 –◉** **3.400.-**
... *Kovalcsikné Pintér Orsolya:* **Az EXCEL függvényei A-tól Z-ig** **3.779.-**
... *László József:* **Hangkártya programozása** Pascal és Assembly nyelven – 🖫 **2.900.-**
... *Szirmay-Kalos.:* **Háromdimenziós grafika, animáció és játékfejlesztés -◉** **5.496.-**
... *Ács Zsolt*: **Linux operációsrendszer-váltás-Linux 9 adminisztrátori ism. ◉** **2.979.-**
... *Czenky M.-Tamás P.- Vágási J.:* **Tanuljuk együtt az informatikát**
ECDL elméleti modul - ◉ **3.495.-**
... *Dr.Kovács T. –Dr.Kovácsné C.J. –Ozsváth M. –G.Nagy J.:* **Mit kell tudni? a PC-ről**
- az OKJ és ECDL vizsgákhoz; Számítástechnikai alapismeretek;
Windows 95-98; Office 97 szoftverek; adatbázis-kezelés; e. levelezés **1.497.-**
... *Dr.Kovács T.-Dr.Kovácsné C.J.-Ozsváth M.-G.Nagy J.:*
Mit kell tudni? a PC-ről – Példatár **2.990.-**
... *László József:* **Mindenkinek az Internetről** **1.999.-**
... *Móricz A.:* **Mobiltelefon a kezekben** **2.990.-**
... *Molnár Mátyás:* **Lotus Notes 6 felhasználóknak** – 6.5 kiegészítéssel **2.997.-**
... *Dr. Kovácsné Cohner J.–Ozsváth M.–G. Nagy J.:* **OFFICE 2000** **2.990.-**
... *Dr. Kovácsné Cohner J.–Ozsváth M.–G. Nagy J.:* **OFFICE XP** **3.498.-**
... *Móricz A.* **Programozási alapfeladatok WAP oldalakhoz és WEB lapokhoz** **3.398.-**
... *Tóth B-. Benkő T.-né és* társai.: **Programozzunk Turbo Pascal nyelven– ◉** **2.992.-**

ISBN 963-618-301-5

9 789636 183011

Ára: 2.994.- Ft
www.computerbooks.hu