ANSI

C++

összefoglaló

ISBN: 963 618 265 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

Tartalomjegyzék

F1. ANSI C++ összefoglaló

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

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

Mielőtt elkezdenénk a szabványos C++ nyelv elemeinek bemutatását, le kell szögeznünk, hogy a C++ nyelv a C nyelv szintakszisára épülő önálló programozási nyelv. Alapvető eltérés a két nyelv között, hogy amíg a C nem típusos nyelv, addig a C++ erősen típusos objektum-orientált nyelv.

A nyelv bemutatását két lépésben végezzük. Először áttekintjük azokat az eszközöket, amelyek a C nyelv természetes kiegészítését jelentik. A lehetőségek felhasználásával hatékonyan készíthetünk nem objektum-orientált programokat. A második részben az objektum-orientált programozást támogató C++ nyelvvel ismerkedünk meg.

A *Borland C++ Builder* rendszer szintén lehetőséget kínál a fenti részek szétválasztására. A *.C* kiterjesztésű fájlok esetén csak bizonyos C-kiegészítések használatát engedélyezi a fordító, míg a második opció választásakor a teljes C++ nyelvdefiníció használható. A *.CPP* kiterjesztésű fájlok esetén mindkét esetben C++ programként fordít a fordítóprogram.

1. A C++ mint egy jobb C nyelv

A C++ nyelv a hagyományos (funkció-orientált) és az objektum-orientált programépítést egyaránt támogatja. Elsőként azokat a nyelvi elemeket fogjuk csokorba, amelyek mindkét módszerrel felhasználhatók hatékonyan működő programok kialakításához. Az F1.2. fejezetben az osztályok által megvalósított objektum-orientált programépítésre helyezzük a hangsúlyt.

1.1. 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>
void main()
{
 char nev[20];
 int  a;
 double b;

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

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 << "Kérem a szöveget: ";
 cin  >> nev;
 cout << "A=";
 cin  >> a;
 cout << "B=";
 cin  >> b;
 cout << nev <<" : A*B=" << a*b;
}
```

Szembeötlő eltérés a C 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 használjuk 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 a *IOSTREAM* deklarációs állomány tartalmazza. (A 16-bites karaktereket tartalmazó szövegek kezelését a fenti objektumok *w* betűvel kezdődő párjaik támogatják: ***wcout***, ***wcin***, ***wcerr***.)Az adatfolyam-osztályok lehetőségeit a későbbiekben részletesen tárgyaljuk. Az itt bemutatott szabványos I/O műveleteket a példaprogramokban kívánjuk felhasználni.

1.2. 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 tokennek 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.

1.2.1. 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.

1.2.2. A C++ nyelv azonosítói

A C++ nyelvű program bizonyos összetevőire (pl. 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, TRUE, FALSE
```

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

```
FelsoSarok, XKoordinata
```

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

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

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

__try, __property, __published

1.2.3. 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.

1.2.3.1. 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:

```
1994,  -1990,   32,   -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 konstans fajtától. 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,   0x12f35e7l
```

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

```
3087007744UL,       0xB8000000LU
```

1.2.3.2. Karakterkonstansok

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

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

Az egyetlen karaktert tartalmazó karakter konstansok 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 megadva	*ooo*	`'\ooo'`
ANSI karakter hexadecimáliskóddal megadva	*hh*	`'\xhh'`

1.2.3.3. Lebegőpontos konstansok

A lebegőpontos konstans olyan decimális szám, amely előjeles valós számot reprezentál. A valós szám általában egész részből, tizedes törtrészből és kitevőből tevődik össze. 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:

```
3.14159265358979323l,          2.71828182845904523541
```

1.2.4. Sztringkonstansok (literálok)

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

```
"Ez egy ANSI sztring konstans!"
```

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

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

melyek esetén csak a nekik megfelelő karakter (egy byte) kerül tárolásra.

Egymás után elhelyezkedő sztring konstansokat szintén egyetlen sztringliterálként tárolja a fordító:

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

A széles karaktereket tartalmazó unicode sztringkonnstansok előtt az L betút kell használnunk:

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

1.2.5. Megjegyzések

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

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

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

1.2.6. 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 tettünk róluk említést, mivel szintén a C++ nyelv alapegységei (tokenjei). A következő táblázatban minden magyarázat nélkül felsoroltuk a C++ nyelv (szabványos) operátorait:

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

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

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

1.3. 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.

1.3.1. Egyetlen modulból felépülő C++ program

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

//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.)

1.3.2. 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 összeg: "<<e<<endl;
}
```

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.
Általában 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

1.4. Alaptípusok, változók, konstansok

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

1.4.1.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 (a négy alapművelet elvégezhető rajtuk).	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.

1.4.1.1. 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 a hatásukat kétféle módon fejtik ki. A **short** és a **long** módosítók a tárolási hosszat, míg a **signed** és az **unsigned** az előjel értelmezését szabályozzák.

A **short int** típus egy rövidebb (2 byte-on tárolt), míg a **long int** típus egy hosszabb (4 byte-on tárolt) 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

1.4.1.2. 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 megváltoztathat. 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

*1.4.1.3. 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;
```

1.4.2. 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 az objektum tárolásával kapcsolatban tartalmaznak előírásokat. Az előírások, amiket tárolási osztálynak nevezünk, meghatározzák az objektum 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 (futás idejű) kifejezéssel inicializálhatjuk. Ezzel szemben a globális elérésű változók kezdőértékeként csak fordítási idejű kifejezéseket adhatunk meg.

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!

1.4.5. 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.

1.4.6 Értékek, címek, mutatók és referenciák

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

```
változó = érték;
```

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

1.4.6.1. 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 jobboldalon megadott konstans értéket be kell másolni. A második értékadás során az *a* változó az

értékadás mindkét oldalán szerepel. A baloldalon álló *a* ugyancsak a tárolót jelöli ki a memóriában (*lvalue*), míg a jobboldalon álló *a* egy jobbérték kifejezésben szerepel, melynek értékét (13) a fordító meghatározza az értékadás elvégzése előtt.

1.4.6.2. 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! A definíció hatására a memóriában létrejön egy (**int** típusú) tároló, melybe bemásolódik a kezdőérték.

```
int x=7;
```

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

```
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.) A

```
*p = x +13;
```

kifejezés feldolgozása során az *x* értéke *20*-ra módosul.

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

```
típus & azonosító = vátozó;
```

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

```
int x = 7;
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 *20* lesz az x változó értéke:

```
r = x +13;
```

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 = 2001;
int & r = b;
r++;  // b nem változik!
```

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

```
int x = 7;
typedef int & tri;
typedef int * tpi;
tri r = x;
tpi p = &x;
```

1.4.6.3.A **void** ** típusú általános mutatók*

Az előző példákban pointerekkel mutattunk **int** típusú változókra. 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 fordítónak. 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=5;
```

1.4.6.4. 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

```
typedef int *  iptr;   // egészre mutató pointer típusa
typedef iptr * ipptr;  // iptr típusú objektumra mutató pointer típusa
iptr p;
ipptr q;
```

A definíciók megadása után kijelenthetjük, hogy az

```
x = 7;
p = &x;
q = &p;
x = x + *p + **q;
```

utasítások lefutását követően az *x* változó értéke *21* lesz.

1.5. 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 csoportosítottuk:

1. Legmagasabb		***asszociativitás: balról->jobbra***
	()	függvényhívás
	[]	tömbindexelés
	->	közvetett tagkiválasztó operátor.
	::	érvényességi kör op.
	.	közvetlen tagkiválasztó operátor
2. Egyoperandusú		***asszociativitás: jobbról->balra***
	!	logikai tagadás (NEM)
	~	bitenkénti negálás
	+	+ előjel
	-	- előjel
	++	léptetés előre
	--	léptetés vissza
	&	a címe operátor
	*	indirektség operátor

	typeid	futásidejű típusazonosítás
	(*típus*)	explicit típuskonverzió
	dynamic_cast	futásidejű ellenőrzött típusátalakítás
	static_cast	fordításidejű ellenőrzött típusátalakítás
	reinterpret_cast	ellenőrizetlen típusátalakítás
	const_cast	konstans típusátalakítás
	sizeof	az operandus bájtban kifejezett méretét adja meg
	new	tárterület dinamikus foglalása
	delete	tárterület dinamikus felszabadítása
3. Osztálytagok elérése		***asszociativitás: balról->jobbra***
	.*	osztálytagra történő indirekt hivatkozás operátora
	->*	mutatóval megadott osztályobjektum tagjára való indirekt hivatkozás operátora
4. Multiplikatív		***asszociativitás: balról->jobbra***
	*	szorzás
	/	osztás
	%	maradék
5. Additív		***asszociativitás: balról->jobbra***
	+	összeadás
	-	kivonás
6. Biteltolás		***asszociativitás: balról->jobbra***
	<<	eltolás balra
	>>	eltolás jobbra
7. Összehasonlító		***asszociativitás: balról->jobbra***
	<	kisebb
	<=	kisebb vagy egyenlő
	>	nagyobb
	>=	nagyobb vagy egyenlő
8. Egyelőség		***asszociativitás: balról->jobbra***
	==	egyenlő
	!=	nem egyenlő
9. - 13.		***asszociativitás: balról->jobbra***
9.	&	bitenkénti ÉS
10.	^	bitenkénti VAGY
11.	|	bitenkénti kizáró VAGY
12.	&&	logikai ÉS
13.	||	logikai VAGY
14. Feltételes		***asszociativitás: jobbról->balra***
	?:	a ? x : y jelentése: "ha a akkor x, különben y"
15. Értékadás		***asszociativitás: jobbról->balra***
	=	egyszerű értékadás
	*=	szorzat megfeleltetése
	/=	hányados megfeleltetése
	%=	maradék megfeleltetése
	+=	összeg megfeleltetése
	-=	különbség megfeleltetése

	&=	bitenkénti ÉS megfeleltetése
	^=	bitenkénti kizáró VAGY megfeleltetése
	|=	bitenkénti VAGY megfeleltetése
	<<=	balra eltolt megfeleltetése
	>>=	jobbra eltolt megfeleltetése
16. Vessző		***asszociativitás: balról->jobbra***
	,	kiértékelés
A előfordító műveletei		
	#	sztringbe illesztés
	##	szövegbe illesztés
	defined	makró létezésének vizsgálata

A felhasználói típushoz kapcsolódóan a C++ lehetőséget kínál az operátorok többségének átdefiniálására (*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.

1.5.1.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 csoporjai az azonos precedenciával rendelkező operátorokat tartalmazzák. A csoportok mellet külön jeleztük az azonos precedenciájú operátorokat tartalmazó kifejezésben a kiértékelés irányát, amit asszociativitásnak (csoportosításnak) hívunk. A táblázat első sora a legnagyobb precedenciával rendelkező műveleteket tartalmazza.

1.5.1.1.Az elsőbbségi (precedencia) szabály

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

```
5 + 3 * 4
```

kifejezés értéke 17 lesz. Ha az összeget zárójelbe tesszük, megváltoztatva ezzel a kiértékelés sorrendjét

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

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

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

akkor ismét 17-et kapunk eredményként, hisz ebben az esetben a zárójelezés csak kiemelte, de nem változtatta meg a számítás menetét.

1.5.1.2. A csoportosítási (asszociativitás) szabály

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

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

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

1.5.2. 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 sorrendjétő, például:

```
a[i] = i++;
```

A művelet-táblázatból 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 baloldali operandus 0 értéke esetén a jobboldali 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.

1.5.3. Elsődleges operátorok

Az első csoportba azok az operátorok tartoznak (`()`, `[]`, `->`, `.`), amelyekkel a későbbiek során ismerkedünk meg részletesen, hiszen hozzájuk speciális C++ nyelvi szerkezetek kapcsolódnak. Nevezetesen a függvényhívás (`fv(argumentumok)`), a tömbindexelés (`tomb[index]`) és a osztálytagra való hivatkozás (`str.tag` vagy `pstr->tag`) operátorokról van szó. A zárójel () operátor azonban kettős szereppel rendelkezik. Mint már említettük, zárójelek segítségével a kifejezések kiértékelése megváltoztatható. A zárójelbe helyezett kifejezés típusa és értéke megegyezik a zárójel nélküli kifejezés típusával és értékével.

1.5.4. 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 egészosztást jelöl.

Megjegyezzük, hogy a + és a - operátorok egyik vagy mindkét operandusa mutató is lehet, ekkor pointerarit-metikáról beszélünk. A megengedett pointeraritmetikai 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

1.5.5. Összehasonlító és logikai operátorok

A C++ nyelvben a logikai típus is az egész típusok közé tartozik (**false** értéke 0, **true** értéke 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 igaz.

Ahhoz, hogy bonyolultabb feltételeket is össze tudjunk hasonlítani, 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

1.5.6. 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 fenti programban bemutatott módon használjuk, nem látszik különbség az előrevetett és hátravetett forma között, hisz 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.

1.5.7. Bitműveletek

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

1.5.7.1. 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. 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;`	`10717 (0x29dd)`
Bitek törlése	1101 1111 1110 1111	`a = a & 0xdfef;`	`2509 (0x09cd)`
Bitek negálása	0010 0000 0001 0000	`a = a ^ x2010;` `a = a ^ 0x2010;`	`10701 (0x29cd)` `2525 (0x09dd)`
Az összes bit negálása		`a = ~a;`	`-2526 (0xf622)`

1.5.7.2. Biteltoló műveletek

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

```
short int a;
```

Értékadás	*Bináris érték*	*Művelet*	*Eredmény*	*Bináris eredmény*
`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 *ax* értéke kétszeresére (2^1) nőtt, míg két lépéssel jobbra eltolva, *ax* 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.

1.5.8. Értékadó operátorok

Már említettük, hogy C++ nyelvben az értékadás egy olyan kifejezés, amely a baloldali operandus által kijelölt tárolónaknak adja a jobboldalon 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 = 13;
b = (a+4)*7-30;
```

során az *a* változó értéke *13*, míg a *b* változóé *89* 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 baloldali kifejezés kiértékelése csak egyszer történik meg. Operátorként (*op*) az eddig megismert kétoperandusú műveletek használhatók:

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`

1.5.9. 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, *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 jelenti.

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

```
int *p, *q, h;
...
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:

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

1.5.10. 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.

1.5.11. A vessző operátor

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

Példaként tekintsük az

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

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

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

```
int x, y;
double z;

x = 5, y = 0, z = 1.2345 ;
```

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

```
int a=13, b=26, 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 operátor. Ezért ezekben az esetekben nem garantált a balról-jobbra haladó kiértékelési sorrend.

1.5.12. A feltételes operátor

A feltételes operátor (?:) három operandussal rendelkezik:

```
kif1 ? kif2 : kif3
```

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;
```

1.5.13. 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) hatókörrel rendelkező nevekre.

```
int i = 126;
void main() {
   double i = 3.14;
   {
      long i = 5,a;
      a=i*::i; // az a változó értéke 5*126=630
      ::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 használjuk, 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();
}
```

1.5.14. 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. 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;
```

Nézzünk néhány példát a **new** használatára!

1. Helyfoglalás 10 elemű egész tömb számára:

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

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

```
delete[] ap;
```

2. Helyfoglalás 10 elemű egészre mutató pointertömb számára:

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

3. Memóriafoglalás ellenőrzés beépítésével:

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

int main()
{
   long * data;
   long size;

   cout << "\nKérem a tömb méretét: ";
   cin >> size;

   // Memóriafoglalás
   data = new long [size];
   // A foglalas sikerességének ellenőrzése
   if ( !data )
   {
      cerr << "\nNincs elég memória !\n" << endl;
      return -1;
   }

   for (long i=0; i<size; i++)
      data[i]=i+1;
   for (long i=0; i<size; i++)
     cout << data[i] << "\t";
   cout << "\n\n";

   // A lefoglalt memória felszabadítása
   delete[] data;
   return 0;
}
```

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()*** függvényel kell az új kezelőfüggvényt definiálnunk.

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

// A new operátor új hibakezelője:
void mem_kezelo() {
   cerr << "\nNincs elég memória!";
   exit(1);        // kilépés a programból
}

void main(void)
{
  // Az új hibakezelő beállítása
  set_new_handler(mem_kezelo);
  char *ptr = new char[10000000];
  cout << "\nElső memóriafoglalás: ptr = " << hex << long(ptr);
  ptr = new char[10000000];
  cout << "\nMásodik memóriafoglalás: ptr = " << hex << long(ptr);

  // Az eredei hibakezelés visszaállítása
  set_new_handler(0);
}
```

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 kiléphetünk a programból.

1.5.15. 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 (*RTTI*).

```
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 és B típusa: " << typeid(A).name();
```

1.5.16. Típuskonverziók

A kifejezések kiértékelése során előfordulhat, hogy valamely kétoperandusú operátor különböző típusú operandusokkal rendelkezik. Ahhoz azonban, hogy a műveletet 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.

1.5.16.1. Explicit típusátalakítások

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

```
(típusnév) kifejezés                    // A C nyelvből származó forma.
típusnév (kifejezés)                    // A korai C++ nyelvből származó forma
static_cast<típusnév>(kifejezés)        // A szabványos alak.
```

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 *Típus* és *arg* azonos típusúak kell legyenek . Az alábbi példában egy nem konstans **int** argumentumot váró függvény is meghívható **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 típus-átalakítás 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.

```
void __fastcall TForml::ButtonlClick(TObject *Sender)
{
    if (dynamic_cast <TButton*>(Sender))
       dynamic_cast <TButton*>(Sender)->Caption="Nyomógomb"
}
```

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 A **reinterpret_cast** két inkompatibilis mutatótípus közötti konverzió elvégzésére is használható.

```
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 *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.

1.5.16.2. Implicit típuskonverziók

Általánosságban elmondható, hogy az automatikus konverziók során a "szűkebb" operandus információvesztés 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=5, j;
float f=3.65;
i + f;
```

Az implicit konverziók azonban nem minden esetben mennek végbe információvesztés 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

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

akkor bizony adatvesztés lép fel, hiszen az összeg törtrésze elvész, és *8* lesz a *j* változó 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 az 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;
```

1.5.17. 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 = 1970;
int * pi;

ci  = 1993;  // Hibás C-ben és C++-ban
pi  = &ci;   // csak C++-ban hibás
*pi = 1993;
```

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

```
const double pi=3.14.159265; // szabályos konstansdefiníció
```

Konstansra csak megfelelő mutatóval hivatkozhatunk:

```
const double ac=1.26;
double * pd = &ac;           // Hiba!

const double *pdc;           // double konstansra mutató pointer
const double dc=1.26;
double   d;
pdc  = &dc;                  // a pdc pointer a dc-re mutat
pdc  = &d;                   // a pdc pointert a d-re állítjuk
*pdc = 6.28;                 // Hiba! A pointer segítségével a d
                             // változó sem változtatható meg.
```

Konstans értékű pointer:

```
int ev;
int * const aktev=&ev;       // Az aktev pointer értéke nem változtatható meg,
*aktev=2001;                 // de a *aktev módosítható.
aktev = &ev;                 // Hiba!
```

Konstansra mutató konstans értékű pointer:

```
const int ev=1970;
const int ae=2001;
const int * const aktev=&ev; // int típusú konstansra mutató konstans pointer;
aktev = &ae;                 // Hiba!
*aktev= 1993;                // Hiba!
```

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

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

1.6. 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**, **throw**
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.

1.6.1. Utasítások és blokkok

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

```
kifejezés;
```

A *kifejezés utasítás* végrehajtása a kifejezésnek az 1.5. 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 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.

1.6.2. Az if utasítás

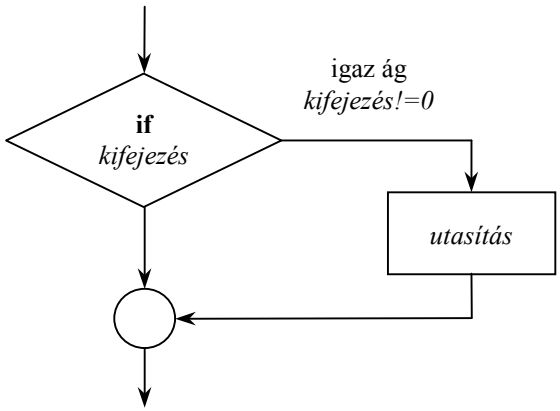
A C++ nyelv két lehetőséget biztosít a program kódjának feltételhez kötött végrehajtására - az **if** és a **switch** utasításokat. Az **if** utasítás segítségével valamely tevékenység (*utasítás*) végrehajtását egy *kifejezés* (feltétel) értékétől tehetjük függővé. Az **if** alábbi formájában az *utasítás*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

```
if (kifejezés)
      utasítás
```

A következő példaprogram egyetlen karaktert olvas a billentyűzetről *Enter*rel lezárva, ha a karakter az *escape* (<*Esc*>) karakter, akkor kilépés előtt hangjelzést ad. Ha nem az <*Esc* > billentyűt nyomjuk le, a program egyszerűen befejezi a futását.

```
#include <iostream>
using namespace std;
#define ESC 27

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

A különböző vezérlési szerkezetek működésének grafikus szemléltetésére a 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)
```

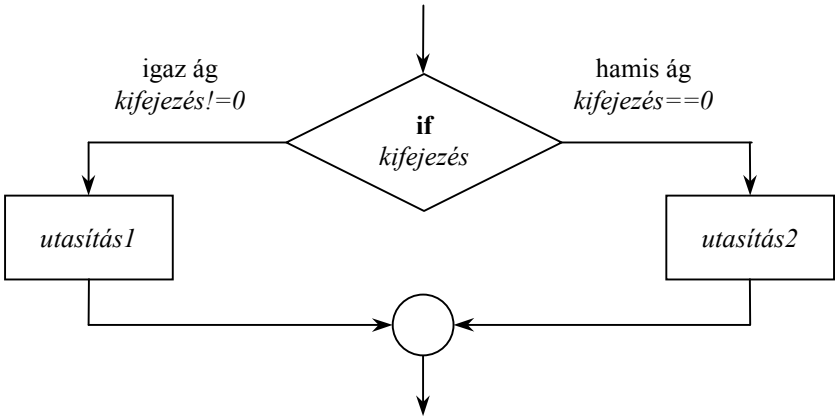
használjuk. Ez a megoldás általában világos, de bizonyos esetekben rejtélyesnek tűnhet. Külön felhívjuk a figyelmet arra, hogy a feltétel kifejezést körülvevő zárójelet mindig ki kell tenni.

1.6.2.1. 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 az páros vagy páratlan:

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

void main()
{
  cout<<"Kérek egy egész számot: ";
  int n;
  cin>>n;
  if (n % 2 == 0)
     cout<<"A szám páros!"<<endl;
  else
     cout<<"A szám páratlan!"<<endl;
  (cin>>skipws).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<<"Negatív páros szám."<<endl;
else
     cout<<"Nem negatív szám."<<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<<"Negatív páros szám."<<endl;
  else ;
else
  cout<<"Nem negatív szám."<<endl;
```

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

```
if (n < 0) {
    if (n % 2 == 0)
       cout<<"Negatív páros szám."<<endl;
}
else
   cout<<"Nem negatív szám."<<endl;
```

1.6.2.2. 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 **elsi-if** programrészlet olvashatósága. Az alábbi példában az *n* számról eldöntjük, hogy az negatív, nulla vagy pozitív:

```
if (n > 0)
    cout<<"Pozitív szám"<<endl;
else if (n==0)
    cout<<"Nulla"<<endl;
else
    cout<<"Negatív szám"<<endl;
```

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

```
12.45+34.55
```

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
  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<<"Hibás művelet!"<<endl;
       return;                    // kilépés a programból
  }
  cout<<a<<op<<b<<'='<<e<<endl;
  system("PAUSE");
}
```

1.6.3. 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** címkére (esetre), amelyben a *konstan_ 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 } utáni utasításra adódik.

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

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;
  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 művelet!"<<endl;
       return;
  } // a switch blokkjának vége
  cout<<a<<op<<b<<'='<<e<<endl;
}
```

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.

Akövetkező 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 válasz [I/N]?";
  char valasz=cin.get();

  switch (valasz) {
    case 'i':  case 'I':
       cout<<"A válasz IGEN."<<endl;
       break;
    case 'n':
    case 'N':
       cout<<"A válasz NEM."<<endl;
       break;
    default:
       cout<<"Hibás válasz!"<<endl;
       break;
  }
  system("PAUSE");
}
```

1.6.4. 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 (kifejezés1opt ; kifejezés2opt ; kifejezés3opt) 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 (1);
```

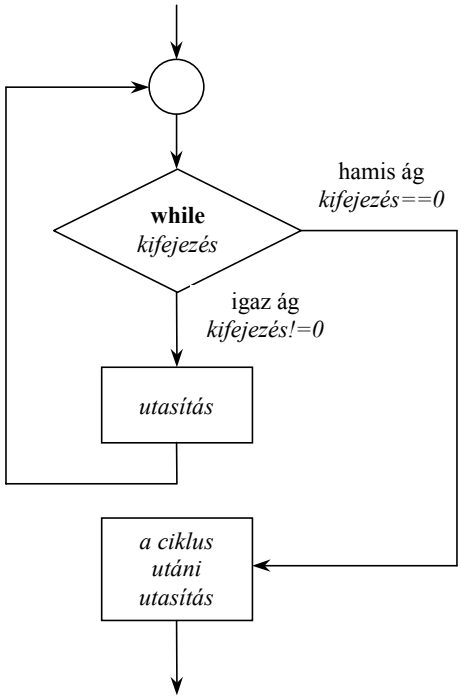
A ciklusokból a vezérlőfeltétel hamis értékének bekövetkezte előtt is ki lehet ugrani (a végtelen ciklusból is). Erre a célra további utasításokat biztosít a C++ nyelv, mint a **break**, a **return**, illetve a ciklus törzsén kívülre irányuló **goto**. A ciklus törzsének biznyos utasításait átugorhatjuk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

1.6.4.1. 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 = 2001;

  cout<<"Az első "<<n<<" egész szám ";
  sum=0;
  while (n>0) {              // vagy : while (n>0)
     sum += n;               //            sum += n--;
     n--;                    //
  }                          //
  cout<<"összege: "<<sum<<endl;
  system("PAUSE");
}
```

1.6.4.2. A for ciklus

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

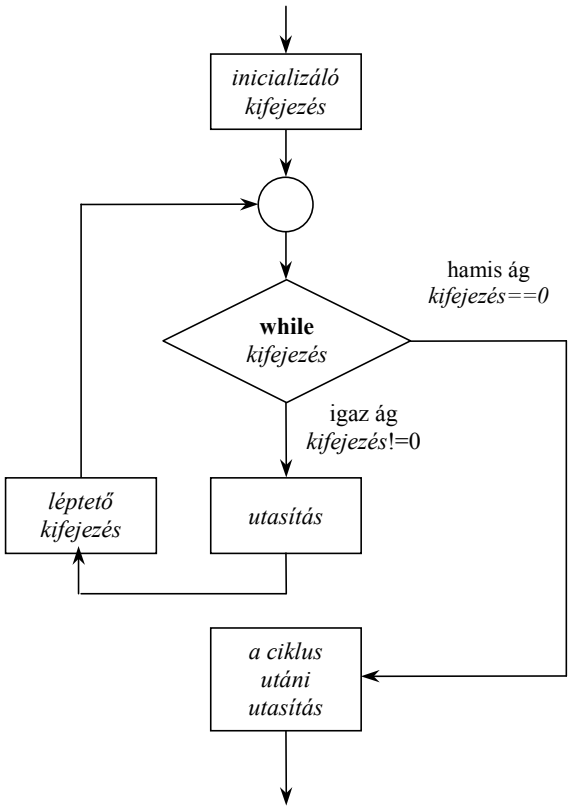
```
for (init_kif ; feltétel_kif ; léptető_kif)
```

```
    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;
}
```

A **for** ciklus működési vázlatát az alábbi ábra tartalmazza:



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), 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), 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 = 2001;

  cout<<"Az első "<<n<<" egész szám ";
  for (i=1, sum=0 ; i<=n ; i++)
      sum += i;
  cout<<"összege: "<<sum<<endl;
  system("PAUSE");
}
```

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:

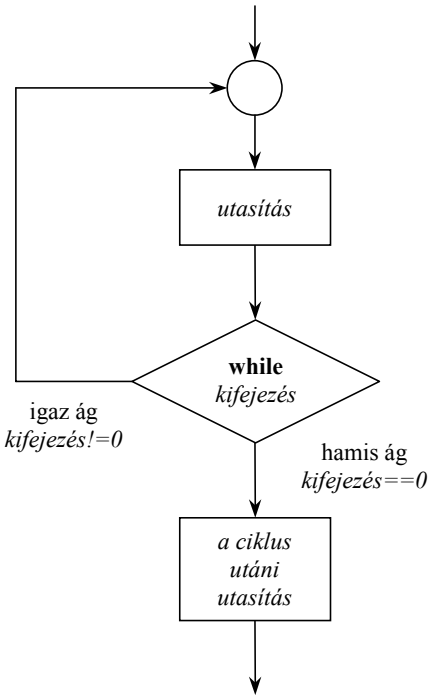
```
#include <iostream>
using namespace std;

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

1.6.4.3. 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. Amennyiben 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 az alábbi ábrán blokkdiagram segítségével ábrázoltuk:



Első példaként az egész számokat összegző programnak írjuk meg 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 = 2001;

  cout<<"Az első "<<n<<" egész szám ";
  sum=0;
  do {                          // vagy : do
     sum += n;                  //            sum += n--;
     n--;                       //        while (n>0);
  } while (n>0);                //
  cout<<"összege: "<<sum<<endl;
  system("PAUSE");
}
```

1.6.5. 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.

1.6.5.1. 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 1.6.3. fejezetben, most csak a ciklusból való kiugrásra mutatunk példákat.

Az alábbi ciklus akkor lép ki, ha megtalálja a legnagyobb olyan 2001-nél kisebb egész számot, amely 123-mal osztható:

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

void main()
{
  int n=2001;
  while (n>0) {
    if (!(n % 123))
       break;
    n--;
  }
  if (n)
     cout<<n<<endl;
  system("PAUSE");
}
```

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;
  system("PAUSE");
}
```

```
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
```

1.6.5.2. 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;
  }
  system("PAUSE");
}
```

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

1.6.6. A goto utasítás

A strukturált, jól áttekinthető (tehát 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;
```

1.6.7. 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 a programunk befejezi a futását, hisz a ***main()*** függvény az ("őt hívó") operációs rendszernek adja vissza a vezérlést

A **return** utasítást

```
return ;
```

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

```
return kifejezés ;
```

A **return** utasítással részletesen a függvényeket tárgyaló fejezetben foglalkozunk.

1.6.8. 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 kivételkezelést tehát 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:

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

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

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

A beérkező kivételek a típusuk alapján a megfelelő kezelőhöz irányítódnak:

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

catch (...)          // minden 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:

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

void beolvas(double & a, double &b, char &op)  throw(bool, char*)
{
   char p[100];
   cout<<":";
   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<<"Viszlát!"<<endl;
          break;   // while
      }
      catch (int) {
          cout<<"Számolási hiba."<<endl;
      }
      catch (char *s) {
          cout<<s<<endl;
      }
      catch (...) {
          cout<<"Ismeretlen kivétel"<<endl;
      }
   }
   system("Pause");
}
```

1.6.9. 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 számot: ";
  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-definiciót feltételt használó utasításokba is bevihetjük. Ennek feltétele, hogy a változót inicializáljuk, például véletlen számmal:

```
if (int s=rand()%13) {
    cout<<s<<endl;
}

while (int n=rand()%26) {
    cout<<n<<endl;
}
```

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

1.7. Származtatott adattípusok

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

1.7.1. Tömbök, sztringek és mutatók

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

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

1.7.1.1. 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:

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

A tömb elemeire pedig az indexelés operátorának (*[]*) segítségével hivatkozunk.

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

```
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[24];
nap[-1] = 12.34;          // hiba!
nap[24] = 356.23;         // hiba!
```

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)};
```

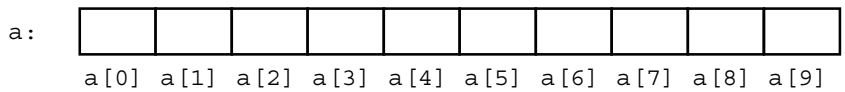
1.7.1.2. Mutatók és a tömbök

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];
```

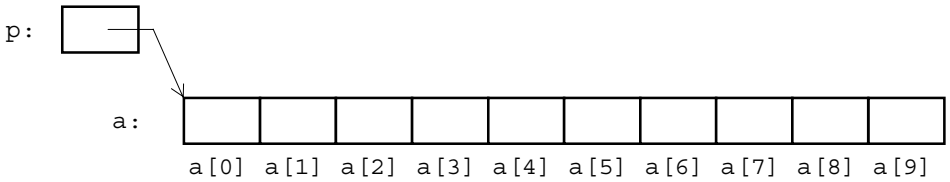
A vektor elemei a memóriában adott címtől kezdve folytonosan helyezkednek el. Mindegyik elemre *a[i]* formában hivatkozhatunk:



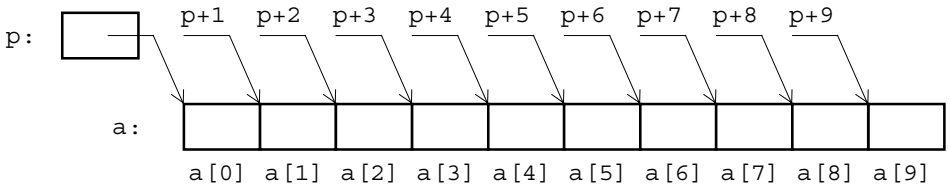
Most vegyünk fel egy *p* egészre mutató pointert, majd a „címe” operátor segítségével állítsuk az *a* tömb elejére (a 0. elemére):

```
int *p;

p = &a[0];
```

Ezek után, ha hivatkozunk a *p* mutató által kijelölt *(*p)* objektumra, akkor valójában az *a[0]* elemre hivatkozunk:



Ha *p* memóriaobjektumra mutat, akkor a mutatóaritmetika szabályai alapján a *p+1*, a *p+2* stb. címek az adott objektum után elhelyezkedő objektumokat 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 konvertálja, majd ezt a pointeres alakot lefordítja. Az analógia azonban visszafelé is igaz, vagyis az indirektség (*) operátora helyett mindig használhatjuk az indexelés ([ ]) operátorát.

1.7.1.3. Sztringek

Az egydimenziós tömböket leggyakrabban karaktersztringek tárolására használjuk. A C++ nyelv nem rendelkezik önálló sztringtípussal, ezért a karaktertömböket használja a sztringek tárolására. 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 sztring számára, akkor a sztring végét jelző bájtot is figyelembe kell venni. Ha az *str* tömbben maximálisan 80 karakteres szöveget szeretnénk tárolni, akkor a tömb méretét 80+1=81-nek kell megadnunk:

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

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

```
char st1[10] = { 'A', 'L', 'M', 'A', '\0' };
char st2[] = { 'A', 'L', 'M', 'A', '\0' };
```

Az *st1* sztring számára 10 byte 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 (sztring-konstansok) felhasználásával:

```
char st1[10] = "ALMA";
char st2[] = "ALMA";
```

A sztringek kezelésére karaktermutatókat is szoktunk használni, azonban a mutatókkal óvatosan kell bánnunk. Tekintsük az alábbi gyakran használt definícióka! Első esetben a fordító létrehozza a 16-elemű *s* tömböt, majd oda bemásolja az inicializáló sztring karaktereit és a 0-ás karaktert. A második esetben a fordító az inicializáló sztringet 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 sztring konstanscímét. 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 sztringek kezelésére szolgáló könyvtári függvények sokkal több lehetőséget biztosítanak a programozónak, mint más nyelvek sztringműveletei. Nézzünk néhány sztringekre vonatkozó alapműveletet elvégzésére szolgáló függvényt:

M ű v e l e t	***F ü g g v é n y***
sztring beolvasása	***scanf(), gets()***
sztring kiírása	***printf(), puts()***
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 végig kell lépkedni karakterenként, akkor választhatunk a tömbös és a pointeres megközelítés között. A következő példaprogramban a beolvasott sztringet 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<<"Kérek egy szöveget   : ";
  cin.getline(s,80);

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

  p=s;
  while (*p)                     // visszaállítás
     *p++ ^= KULCS;
  cout<<"Az eredeti szöveg : "<<s<<endl;
  system("pause");
}
```

Mind a két esetben a ciklusok leállási feltétele a sztringet záró 0-ás bájt elérése volt.

1.7.1.4. 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 használunk.

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

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

$$\begin{bmatrix} 4 & 26 & 90 & 14 & 11 \\ 13 & 30 & 70 & 63 & 9 \\ 7 & 87 & 60 & 19 & 12 \end{bmatrix}$$

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

```
int matrix[3][5] = { { 4, 26, 90, 14, 11 },
                     {13, 30, 70, 63,  9 },
                     { 7, 87, 60, 19, 12 } };
```

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

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

hivatkozással a *2*. sor *3*. sorszámú elemét (*19*) jelöljük ki. Nem szabad megfeledkeznünk arról, hogy az indexek értéke minden dimenzióban *0*-val kezdődik.

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)
```

Az alábbi 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;
}
```

1.7.1.5. 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];
```

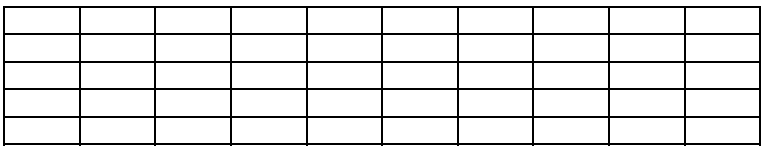
Formailag az *a[2][5]* és a *b[2][5]* hivatkozások egyaránt helyesek, hiszen mindkét esetben egyetlen **int** típusú elemet jelölnek. Azonban az *a* egy igazi kétdimenziós tömb, amely számára a fordító 50 **int** típusú elem tárolására alkalmas területet foglal le a memóriában. Az elemek helyének meghatározására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

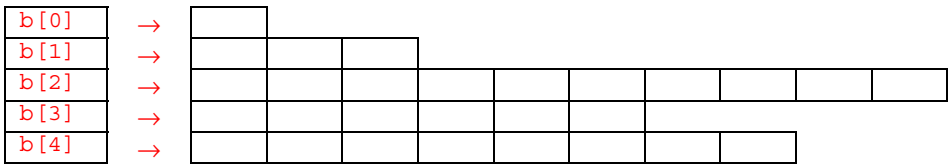
```
int s1[10], s2[10], s3[10], s4[10], s5[10];
int *b[5] = { s1, s2, s3, s4, s5 };
```

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:

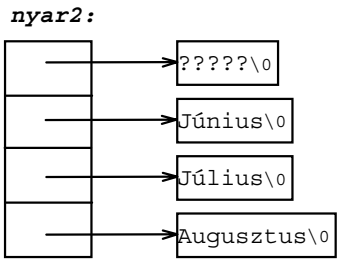
nyar1:

?????\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 feltöltött a fordító:



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

1.7.1.6. Dinamikus helyfoglalású tömbök

A tömböket használó program 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 alapelve az, hogy az objektum számára csak akkor foglalunk helyet, amikor szükségünk van rá, illetve ha már nincs szükségünk az objektumra, akkor az általa elfoglalt memóriaterületet 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 memóriaterületet. A memóriafoglalás elvégzéséhez a **new** operátort, míg a felszabadításához a **delete**[] operátort használjuk.

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.

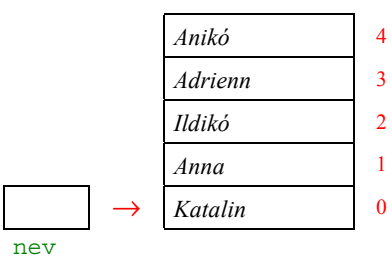
Megoldás statikus kétdimenziós karaktertömb használatával

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

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<<"Hány nevet kíván rendezni: ";
  cin>>db;  cin.get();
  if (db>maxn || db<1) return;  // Kilépés

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

  // Rendezés
  char sv[51]; // Segédváltozó
  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);
     }

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

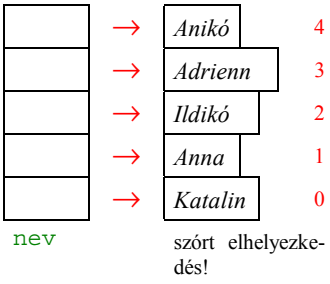
Megoldás egydimenziós 51-karakteres szöveges elemeket tartalmazó dinamikus tömbbel



A sorok száma tetszőleges, azonban a nevek legfeljebb 50 karakterek lehetnek. Valójában egydimenziós dinamikus tömböt használunk, azonban a tömbelemek maguk is egydimenziós karaktertömbök. A megoldásban aláhúztuk a lényeges programsorokat.

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

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

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

  // Dinamikus memóriafoglalás
  nev=new tsor[db];

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

  // Rendezés
  char * sv=new char[51]; // Segédváltozó
  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;

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

  // A lefoglalt memória felszabadítása
  delete [] nev;
}
```

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. A megoldásban aláhúztuk a lényeges programsorokat.

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

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

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

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

```
// A nevek beolvasása (mérethatár az input puffer mérete)
  char sv[251];
  for (int i=0; i<db; i++) {
    cout<<i<<". név: ";
    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;
}
```

1.7.2. Felhasználó által definiált adattípusok

A C++ nyelv lehetővé teszi, 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.

1.7.2.1. A struct *struktúratípus*

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

C++ nyelven a struktúra (**struct**) típus több tetszőleges típusú (kivéve a **void** és a függvény típust) adatok együttese. Ezek az adatelemek önálló, a struktúrán belül érvényes nevekkel rendelkeznek. Az objektumok szokásos elnevezése struktúraelem vagy adattag (*member*). (A 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;
                    típus2 tag2;
                       . . .
                    típusN tagN;
                    };
```

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

```
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 az alábbi 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
};
```

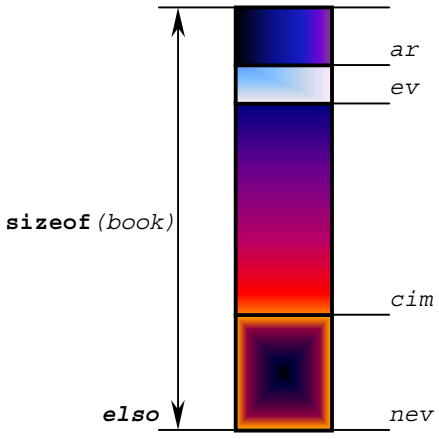
A típusdeklaráció után változókat is létrehozhatunk:

```
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.

Hivatkozás a struktúra adattagjaira

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

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

Az *s1* és az *s2 book* típusú statikus helyfoglalású változók, amelyek tárolásához szükséges memóriaterület 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 memóriafoglalás használatát jelenti. Az alábbi programrészletben memóriát foglalunk a *book* struktúra számára, majd pedig felszabadítjuk azt:

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

A megfelelő definíciók elvégzése után van három struktúránk, az *s1, s2* és a **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 = 2000;
s1.ar = 34.95;
```

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

A pont operátort a *ps* által mutatott objektumra alkalmazva a precedencia szabályok miatt zárójelek között kell megadni a **ps* kifejezést:

```
strcpy( (*ps).nev, "Bjarne Stroustrup");
strcpy( (*ps).cim, "The C++ Programming Language");
(*ps).ev = 2000;
(*ps).ar = 34.95;
```

Mivel a C++ nyelvben (főleg a C nyelv első változatában) gyakran használunk mutató által kijelölt struktúrákat (például függvények argumentumaként), a C++ nyelv ezekben az esetekben egy önálló operátort - a nyíl (->) operátort - biztosít az adattag-hivatkozások megadására. (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 = 2000;
ps->ar = 34.95;
```

A nyíl operátor bal oldali operandusa a struktúra-változóra mutató pointer, míg a jobb oldali 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, mint például:

```
(&s1)->ev=1988;
```

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 struktúra-változókra vonatkozó értékadás (=) műveletét:

```
 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 megoldás közül is választhatunk. Egyrészt az adattagonkénti értékadás módszerét használva magunknak kell kiküszöbölni a problémát, másrészt pedig a másolás műveletének átdefiniálásával (*operátor overloadig*) saját értékadó operátort definiálhatunk a struktúrához.

Kezdőérték-adás a struktúrának

A tömbökhöz hasonlóan a struktúra definíciójában is szerepelhet kezdőérték-adás. 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",
            2000, 34.95 };
```

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:

```
typedef struct {
          int nelem;
          int v[20];
} Vektor;

Vektor a = {5, {1, 2, 3, 4, 5} };

Vektor b = {4, 10, 20, 30, 40 };
```

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 beépíthető 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.

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[13].ar = 123.23;
```

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[13]).ar = 123.23;
```

A struktúratömböt a definiálásakor a szokásos módon inicializálhatjuk. A áttekinthetőség érdekében ajánlott az egyes struktúrák kezdőértékét kapcsos zárójelben elkülöníteni:

```
book mlib[] ={ { "0. Író" , "0. Könyv", 1999, 1000 },
               { "1. Író" , "1. Könyv", 2000, 2000 },
               { "2. Író" , "2. Könyv", 2001, 3000 } };
```

Amennyiben dinamikusan kívánjuk a „könyvtárat” létrehozni, mutatót kell használnunk az azonosításra:

```
book * plib;
```

A struktúraelemek számára a new operátorral foglalhatunk helyet a dinamikusan kezelt memóriaterületen:

```
plib = new book[100];
```

A tömbelemben tárolt struktúrára a pont operátor segítségével hivatkozhatunk:

```
plib[14].ar = 25.54;
```

Ha már nincs szükségünk a struktúra elemeire, akkor a **delete**[] operátorral felszabadítjuk a lefoglalt memóriaterületet:

```
delete[] plib;
```

1.7.2.2. 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 kezdőérték-adás kérdésével foglalkozunk. Általánosan is elmondhatjuk, hogy (**struct** vagy **class**) osztálytípusú változok 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 2. fejezetben részletesen foglalkozunk, hiszen ezeken alapulnak a C++ nyelv objektum-orientált lehetőségei.

1.7.2.3. 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 memóriakezelés. Nézzük meg, miben áll a következő két fejezetben bemutatásra kerülő megoldások lényege:

- Helyet takarítunk meg, ha ugyanazt a memóriaterületet 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 objektumokat, amelyek értéke 1 bájtnál kisebb területen is elfér, egyetlen bájtban helyezzük 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 az 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}; // kezdőérték-adás az első mező szerint
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<<"Hibás adattípus!"<<endl;
}
```

Névtelen union-ok 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<<"Név        :  "<<vr1.nev<<endl;
switch (vr1.tipus) {
   case 'd' :
         cout<<"Szám       :  "<<vr1.szam<<endl;
         break;
   case 'c' :
         cout<<"Szöveg     :  "<<vr1.szoveg<<endl;
         break;
   default  :
         cout<<"Hibás adattípus!"<<endl;
}
```

1.7.2.4. A bitmezők használata

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 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",
                      2000, 34.95 },
                      20010820L, 1, 0, 8};
```

majd pedig adjunk értéket a bitmezőknek!

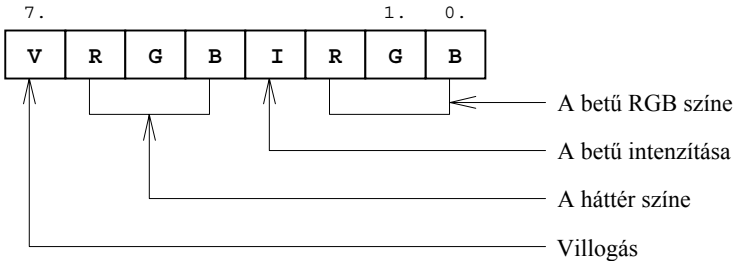
```
cppprog.kikolcsonozve = 1;
cppprog.kolcsonozheto = 1;
cppprog.het           = 3;
```

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 hátrányai!

- 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.)

1.8. 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 meghívott függvénynek. A már bemutatott **return** utasítás végrehajtásakor, illetve a függvény fizikai végének elérésekor a hívott függvény visszatér a hívás helyére, és a **return** utasításban szereplő kifejezés mint függvényérték (visszatérési érték) jelenik meg. A visszatérési érték nem más, mint a függvényhívás kifejezés értéke.

1.8.1. Függvények definíciója és deklarációja

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

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

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

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

```
⟨visszatérési típus⟩ függvénynév (⟨paraméter-deklarációs lista⟩)
{
   // 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. Ffüggvények 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 (⟨paraméterdeklarációs 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 konverzió 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 );
```

Azon függvények prototípusát, amelyek nem rendelkeznek paraméterrel, 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, ...);
```

1.8.2. 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 mint a függvény lokális változói 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 tömb típusok 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 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 függvényt 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. 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 alkalamzott függvényhívási konvenciók.

1.8.3. A függvényhívás

A függvényhívás olyan kifejezés, amely átadja a vezérlést és az argumentumokat (amennyiben 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
```

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 írni, amely felcseréli két egész típusú változó értékét? Az érték szerint argumentumátadás 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 értéke, hanem a változók címe (*&x* és *&y*) adódik át, amit 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 paraméterátadás 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 forrásprogramban ez nem látható.)

Nézzünk néhány érdekes példát mutatók és a referenciák paramétersorban való használatára! Az elsőben **void** 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 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 = 471330;
  int *ap;
  cime( &ap , &a );  // Mint az ap = &a
  *ap += 26;         // Az a értéke 471356 lesz!
}
```

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=471330;
   *kozvetit1(&a) += 26;  // Mint az a += 26;
  // Az a értéke 471356 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=471330;
   kozvetit2(a) += 26;  // Mint az a += 26;
  // Az a értéke 471356 lesz!
}
```

Felhívjuk a figyelmet arra, hogy függvényen belül definiált (*auto*) lokális tárolók 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.

1.8.4. 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.

1.8.4.1. 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 függvény értékét 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;
```

1.8.4.2. 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 referencia típus 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
}
```

1.8.4.3. 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.

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 megmaradnak.

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 sztringek 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 elemű 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 pointerműveletekkel 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 meghívni:

```
double b1 = atlag1 (a, 10);

double b2 = atlag2 (a, 10);
```

A vektor címét és méretét más módon is megadhatjuk, például egy struktúrában is 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 meghívás 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);
  system("pause");
}
```

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 azonban írjuk meg a függvénynek azt a változatát, amely csak 3x4-es tömbök megjelenítésére alkalmas:

```
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;
  }
}
```

1.8.4.4.Sztringargumentumok

A sztringargumentumok 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 index-változóra, 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!

```
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
}
```

1.8.4.5. 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.

*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 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:

```
faktfv fakt;                      // prototípus

faktfv fakt                       // definíció
{
   unsigned long f = 1;
   for ( ; n > 0 ; n--) f *= n;
   return f;
}
```

Függvényre mutató pointerek

A C++ nyelvben a függvénynevek kétféle módon használhatók. A függvénynevet a függvényhívás operátor baloldali 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* nemcsak 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, mivel a függvényhívás operátora erősebb precedenciájú az indirekt hivatkozás operátoránál. Természetesen az *fptr* mutató 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;
}
```

1.8.4.6. 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 *, ... );
```

A három pont azt jelenti a fordítóprogram számára, hogy „még lehetnek további argumentumok”. A ***printf()*** esetén legalább egy argumentumnak kell szerepelnie, amelyet tetszőleges 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 formátumsztring 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. (A szabvány javaslata alapján csak a ***va_arg*** és a ***va_start*** rutinok makrók, míg a ***va_end*** rutint könyvtári függvényként kell megvalósítani.)

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.

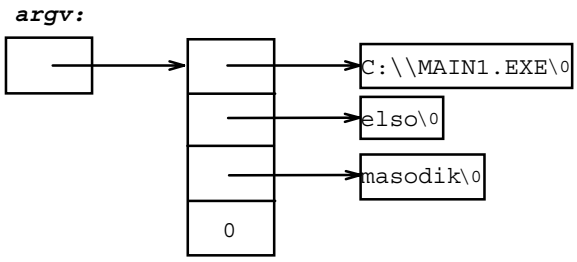
```
int osszeg(int elso, ...) {          // Egész számok összegzése 0-ig
    va_list  ap;
    int      s = elso, ertek;
    va_start( ap, elso);             // Az első argumentum átlépése
    while (ertek = va_arg(ap, int)) // Az int argumentumok
           s += ertek;
    va_end(ap);
    return s;
}

double atlag(int n, ...) {           // Adott számú double érték átlagolása
  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);       // A double argumentumok
  va_end(ap);
  return s / n;
}
```

1.8.4.7. A main() függvény paraméterei és visszatérési értéke

A ***main()*** függvény különlegességét nem csak 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.

1.8.5. Rekurzív függvények használata

A matematikában lehetőség van bizonyos adatok és műveletek rekurzív definiálására. Minden r e k u r z í v problémának létezik i t e r a t í v (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:

$a_0 = 0$

$a_1 = 1$

$a_n = a_{n-1} + a_{n-2}, \quad n = 2,3,4,...$

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;
}
```

1.8.6. 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 meghatározható.) 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.

1.8.7.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 vesz.

Az **inline** megoldás 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.

1.8.8. 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.

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 konverzió, illetve a **float** -> **double** konverzió felhasználásával.

4. Keresés a szabványos típuskonverziók alkalmazásával. Ezek a konverziók alapvetően a szokásos aritmetikai (**int** → **double**, **unsigned** → **signed**), és a mutatókonverziókat jelentik. A pointerkonverziókat az alábbiakban foglaltuk össze. (A referenciatípusok konverziója a mutató típusok konverziójának megfelelően megy végbe.)

Típusról	*Típusra*
tetszőleges mutatótípus	**void ***
leszármaztatott osztály mutatója	az alaposztályra mutató pointer,
0 konstans	nulla 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 a következő részben foglalkozunk részletesen.

1.8.9. Általánosított függvények (template)

A függvények átdefiniálásánál láttuk, hogy sok esetben ugyanazt a függvényfelépítést használjuk a túlterhelt függvényváltozatokban, csupán néhány típus 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();
}
```

1.8.10. 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! 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>
}
```

1.9. Tárolási osztályok

Ahhoz, hogy igazán megértsük a C++ program 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 tárolási osztály egyrészt meghatározza, hogy a változó hol jön létre a memóriában (regiszterben, statikus vagy dinamikus területen), 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 rátérünk a tárolási osztályok részletes tárgyalására tisztáznunk kell néhány, a témával szoros kapcsolatban álló fogalmat:

- élettartam (*lifetime*),
- láthatóság (*visibility*),
- érvényességi tartomány (hatókör, *scope*),
- kapcsolódás (*linkage*),
- névterület (*namespace*).

1.9.1. 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 objektumok.

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 memóriaterület (és a benne tárolt adatok) a program futásának teljes időtartama alatt megmarad. A globális változók inicializálása egyetlen egyszer - a program indításakor - megy végbe.

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 memóriaterülete (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 memóriaterületek, amelyeket a **new** operátorral lefoglalunk, illetve a **delete** operátorral felszabadítunk.

1.9.2 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), függvényszintű vagy prototípusszintű.

- 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.

1.9.3. 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). A szabvány háromféle kapcsolódást különböztet 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 memóriaterülettel. C++ nyelven az alábbi azonosítók nem rendelkeznek kapcsolódással:
  - minden olyan azonosító, amely nem változó vagy függvényt jelöl (például az **enum** konstansok, a címkék stb.),
  - a függvényparaméterek,
  - olyan blokkszintú hatókörbeli változó-azonosítók, melynek deklarációjában nem szerepel az **extern** kulcsszó.

1.9.4. Névterületek

A fordító a programban használt neveket (azok felhasználási módjától függően) különböző területeken (névterület - *namespace*) tárolja. Valamely névterületen belül tárolt neveknek egyedinek kell lenni, 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 felhasználhatunk. A tagok elérése csak 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 felhasználni.

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 *::alma*.)

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 szereplenek, az azokhoz tartozó definíciókat a névtéren kívül is megadhatjuk, használva az érvényességi kör 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, 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 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();
}
```

1.9.5. A tárolási osztályok használata

A tárolási osztály megadásával lehetőségünk van az alapértelmezéstől eltérő élettartam és érvényességi tartomány 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 deklarációjának elhelyezkedése a forrásállományban 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, ahol a definíció helyétől a fájl végéig
	változó-deklaráció	**extern**	globális	a definíció helyétől a fájl végéig
	prototípus vagy függvénydefiníció	**static**	globális	korlátozva az adott fájl ra
	prototípus	**extern**	globális	a definíció helyétől a fájl végéig
blokkszintű 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.

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.) Az **auto** változók inicializálása minden esetben végbemegy, amikor a vezérlés a blokkhoz kerül. Azonban csak azok a változók kapnak kezdőértéket, amelyek definíciójában szerepel kezdőérték-adás. (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++ verzió lehetővé teszi az automatikus tömb és struktúra típusú változók tetszőleges kifejezéssel történő inicializálását

```
double b[3]={sin(12), cos(12), sqrt(12)};
```

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** változó 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, azonban az **extern** tárolási osztályt követően. Az elmondottakat jól szemlélteti az alábbi példa:

```
// modul1.cpp
double KorKerulet(double r);

double KorTerulet1(double r)
{
   return r*KorKerulet(r)/2;
}

extern double pi;

double KorTerulet2(double r)
{
   return r*r*pi;
}
```

```
// modul2.cpp
double pi=asin(1)*2;

double KorKerulet(double r)
{
  return 2*r*pi;
}
```

A szabványos C++ lehetővé teszi, hogy a statikus élettartamú változók kezdőértékeként futás idejű kifejezést adjunk kezdőértékként.

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 belső szintű nevek előtt adjuk meg, a nevek élettartamát automatikusról globálisra módosítja.

Amikor több modulból álló programot írunk, a moduláris programozás elvének megvalósításához nem elegendő az, hogy vannak közös változóink. Szükségünk lehet olyan modul szinten definiált változókra és függvényekre is, amelyek elérését a modulra korlátozhatjuk (információrejtés). Ezért az **extern** és a **static** tárolási osztályú azonosítókat egyaránt használunk a megfelelő modulszerkezet kialakításához. A C++ szabvány a statikus külső változók 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 modul szintű 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, amely kockadobás szimulációjához használja a fenti *random()* függvényt:

```
// 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(2001);    // A generátor inicializálása
  cout<<"Ötször dobunk a kockával: "<<endl;
  for (int i=0; i<5; i++)
   cout<<i<<". dobás \t"<<random()%6+1<<endl;
}
```

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 s*random()* 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). Statikus változó kezdőértékét tetszőleges kifejezéssel definiálhatjuk.

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 két megállapítás is következik. Az első, hogy nem használhatjuk a „címe” (&) operátort **register** tárolási osztályú operandussal. A másik, 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 a **char**, az **int,** a mutató- és a referenciatípusokat tárolja regiszterben.

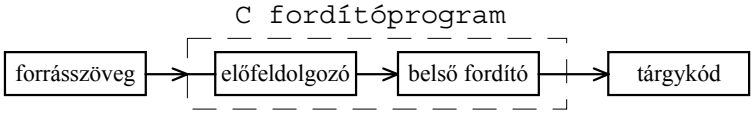
Az alábbi *csere* függvény a korábban bemutatott megoldásnál potenciálisan gyorsabb működésű. (A potenciális szót azért használjuk, mivel nem ismert, hogy a **register** előírások közül hányat teljesít a fordító.)

```
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.

1.10. 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 a **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.

1.10.1. 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,

1.10.2. 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 szimbólum megfelelő értékkel való definiálását a program elején végezzük el:

```
#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)            #ifdef DEBUG
. . .                         . . .
#endif                        #endif
```

illetve a fordított megoldás:

```
#if !defined(DEBUG)           #ifndef DEBUG
. . .                         . . .
#endif                        #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 a fent vázolt problémát! A következő megoldást minden makrókat definiáló fejállományban ajánlott alkalmazni. A deklarációs fájl első beépítésekor létrejön egy szimbólum (például *fqjlnevH*), melynek létezését vizsgálva elkerülhető az ismételt beépítés:

```
#ifndef fajlnevH
#define fajlnevH

   makró-definíciók és egyéb deklarációk

#endif
```

1.10.3. 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.

Szimbolikus konstans makrók készítése

Ebben az esetben a ***#define*** direktíva egyszerűbb alakját használjuk:

```
#define  azonosító  helyettesítő szöveg
```

Nézzünk néhány példát szimbolikus konstans definiálására!

```
#define EOS    '\0'            #define URES
#define TRUE   1               #define STR    ""
#define FALSE  0               #define UDV    "Üdvözöllek dicső lovag ....!"
#define NOT    !               #define PMERET 1024
#define BOOL   int             #define HA     if
#define IGEN   1               #define KIIR   cout
#define NEM    !IGEN
```

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)
```

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 használunk a négyzetre emelés elvégzésére:

```
inline int square(int 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);
}
```

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" változó értéke: "<<n<<endl;

void main()
{
   int     a = 13;
   char   *p = "C++ nyelv";
   double pi = 3.14259265;
   megjelenit (a);
   megjelenit (p);
   megjelenit (pi);
}
```

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*** használata:

```
#undef azonosító
```

Az ***#undef*** utasítás nem jelez hibát, ha az azonosító nincs definiálva.

Előre definiált makrók

Az ANSI C++ szabvány az alábbi előre definiált makrókat tartalmazza:

Makró	***Leírás***	***Példa***
`__DATE__`	A fordítás dátumát tartalmazó sztringliterál.	*"Aug 15 2001"*
`__TIME__`	A fordítás időpontját tartalmazó sztringliterál.	*"07:30:13"*
`__FILE__`	A forrásfile nevét tartalmazó sztringliterál.	*"preproc.c"*
`__LINE__`	A forrásállomány aktuális sorának sorszámát tartalmazó számkonstans (1-től sorszámoz).	*26*
`__STDC__`	A értéke 1, ha a fordító ANSI C++ fordítóként működik, különben nem definiált.	
`__cplusplus`	A értéke 1, ha C++ forrásállományban tetszteljük az értékét, különben nem definiált.	

(Mindegyik azonosító két-két aláhúzáskarakterrel kezdődik és végződik.) Az előre definiált makrók nevét nem lehet sem a ***#define*** sem pedig az ***#undef*** utasításokban szerepeltetni. Az előre definiált makrók értéke beépíthetjük a program szövegébe, de a feltételes fordítás feltételeként is felhasználhatjuk.

1.10.4. 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 kezdõsorszám "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 a Turbo C++ rendszerben nem a *small* memóriamodellt állítjuk be:

```
#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:

```
#
```

2. A C++ mint objektum-orientált nyelv

Az objektum-orientált programozás (OOP) a gondolkozást, cselekvést közelítő programozási mód. Az objektum-orientált programozási nyelv sokkal strukturáltabb, modulárisabb és absztraktabb, mint egy hagyományos programozási nyelv.

Ellentétben a hagyományos programozási nyelvekkel, mint például a C, nem a műveletek (funkciók) megalkotása áll a programozás központjában, hanem az egymással kapcsolatban álló programegységek 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()
{
  //a Vektor típusú objektumpéldány létrehozása
  Vektor v1;

  // hivatkozás az objektum tagjaira
  v1.init(3,4);  // a public elérésű init() tagfüggvény hívása - OK.
  v1.x=4;        // HIBA! Az x és y adattagok kívülről nem érhetők el.
}
```

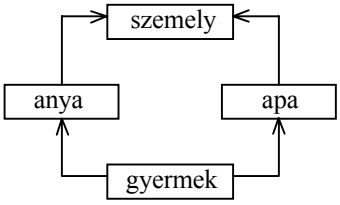
A C++ nyelvben definiált osztálytípus a nyelv szerves részévé tehető, az ún. *operator overloading* (operátor átdefiniálás, túlterhelés) mechanizmusának felhasználásával. Ennek segítségével az általunk létrehozott típushoz definiálhatunk operátorokat, konverziókat sőt akár specifikus I/O műveleteket is. Ugyancsak ezt a célt szolgálják az objektum automatikus inicializálását elvégző konstruktorok, illetve az objektum lebontásában fontos szerepet játszó destruktor használatának 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 öröklődés (*inheritance*) lehetősége. Az öröklődés 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, 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 Alaposztály"; }
};

class  uj : public alap {
public:
  void f2() { cout << "\n Származtatott osztály"; }
};

void main() {
  alap bo;
  uj   uo;
  bo.f1();
  uo.f1();
}
```

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:

```
Alaposztály
Alaposztály
```

Az alaposztályban az *f2()* tagfüggvényt virtuálisnak definiálva

```
class alap {
public:
  void f1() { f2(); }
  virtual void f2() { cout << "\n Alaposztály"; }
};
```

a program kimenete a kívánt eredményt tartalmazza:

```
Alaposztály
Származtatott osztály
```

A továbbiakban az objektum-orientált C++ programozás eszközkészletével ismerkedünk meg.

2.1. Osztályok definiálása

A C++ **struct** deklaráció a C **struct** típus kiterjesztését tartalmazza. Ez 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** egyaránt tartalmazhatnak adatmezőket (adattag), és ezekhez a mezőkhöz definiált műveleteket, függvénymezők (tagfüggvény, metódus) formájában. A **struct** és a **class** adattípusok egyaránt használhatók osztályok definiálására, azonban a tagok alapértelmezés szerinti hozzáférésének következtében a **class** definíció áll közelebb az objektum-orientált gondolkodásmódhoz.

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. Általában az osztálydefiníció az adattagokon és tagfüggvényeken kívül a tagokhoz való hozzáférést szabályzó **public**, **private** és **protected** 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);
};
```

2.1.1. Adattagok

Az osztályban az adattagokat a C és a C++ változóihoz hasonlóan deklaráljuk, egyetlen különbség, hogy az adattagok nem tartalmazhatnak inicializációs listát. Ha egy osztályban valamely más osztály objektumát kívánjuk elhelyezni, akkor a másik osztály definíciójának 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 *to;
 ...
};
```

Osztályon belül nem lehet közvetlenül saját osztálydefinícióval rendelkező objektumot adattagként elhelyezni. Ehhez a másik osztályt előzőleg definiálni kell.

2.1.2. Tagfüggvények

A tagfüggvényeket az osztály törzsén belül deklaráljuk. Ez a deklaráció vagy 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 tartományában definiáltak, ellentétben a közönséges függvényekkel, amelyek fájl szintű vagy névtér szintű érvényességi körrel 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 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");
```

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-hivatkozások automatikusan

```
this->adattag
```

formában kerülnek be a kódba. A példánkban szereplő osztály tagfüggvényeinek ***valódi*** (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.

2.1.2.1. 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 függvény-fejet 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:
   Osztaly() { f=num=szamlalo=0; ptr=NULL;}             // Alapértelmezett konstruktor
   void set (int i, float f, char *p=0);
   int getint(void) const { szamlalo++; return num; } // Konstans tagfüggvény
   float getfloat(void) const { return f; }            // Konstans tagfüggvény
};
```

A **mutable** kulcsszó segítségével jelölt osztálytagok minden esetben megváltoztathatók, még akkor is, ha konstansként definiáljuk az osztály példányát. (A **mutable** előírás nem használható **static** és **const** nevekre.)

```
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;
}
```

2.1.3. Az osztály tagjainak elérése

A C++ az osztály koncepcióban megvalósítja az információrejtés elvét. A **public** (nyilvános), **private** (privát) és a **protected** (védett) alapszavakkal az egyes tagok elérését szabályozhatjuk.

```
class Clppp {
   // Itt helyezkednek el a private elérésű tagok.
   // Természetesen a private: is használható a tagok előtt megadva.
 protected:
   // protected adattagok és tagfüggvények
 public:
   // korlátozás nélkül elérhető adattagok és tagfüggvények
};
```

Az osztály deklarációjában több **public**, **private** és **protected** rész is szerepeltethető. 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 egyes elérést szabályzó 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 privát**,** 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 leszármaztatott osztály tagfüggvényei (habár a privát tagok is öröklődnek), 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 osztály származtat belőle.

Mivel az adattagok általában nem elérhetők az osztály felhasználói számára, ún. nyilvánost elérési tagfüggvényeket szokás definiálni az adattagokhoz (*setxxx()*, *getxxx()*). Az elérési tagfüggvények hívásával az adattagokhoz való hozzáférés az ellenérzésünk mellet megy végbe.

2.1.4. 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 Fclass {
  friend Sclass;  // Az Sclass osztály barát, így minden tagfüggvényére vonatkozik
                  //a baráti viszony. Ez a kapcsolat azonban nem kölcsönös!

  friend int Tclass::szamlal(int x);
                  // A Tclass osztálynak csak a szamlal() tagfüggvénye barát.

  friend long sum(void);
                  // A sum függvény barát.
                  // ...
};
```

2.1.5. Az osztály objektumai

C++ nyelvben az objektumok az osztály példányait (*instance*) jelölik, melyeket a szokásos változó-definícióval hozzuk 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.
```

A dianmikusan létrehozott objektumok megszüntetéséről magunknak kell gondoskodnunk 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 mindkét objektumban a *ptr* mutató 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 el, amelyet szintén átdefiniálhatunk.

Az osztályok tagjait a felhasználói típusoknál megismert operátorokkal érhetjük el. A pont (.) operátort statikus, míg a nyíl (->) operátort dinamikusan létrehozott objektumok esetén használjuk:

```
class RosszOsztaly // Mivel az adattag nyilvános elérésű!
{
 public:
   int a;
   void init(int x) { a = x; }
};

void main()
{
   RosszOsztaly x;
   RosszOsztaly *px;
   px = new RosszOsztaly;
   x.a = 26;              px->a=70;
   x.init(26);            px->init(70);
   *px = x;   // 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.

2.1.6. 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):

Az incializálás függetlenül a statikus adattag elérhetőségétől, mindig elvégezhető. Ha a statikus adattag **public** elérésű, a programban bárhonnan hozzáférhetünk az osztály neve és a hatókör (*scope)* operátor felhasználásával.

Az alábbi példában a statikus tagok használatának bemutatásán túlmenően, szemléltetjük 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:
    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; }
};

const double Math::Pi=M_PI;              // A statikus adattagok kezdőértékének
double Math::dFokRad = Math::Pi/180;     // beállítása
bool Math::eRadian = Math::rad;
```

Az osztály lehetséges alkalmazását az alábbiakban láthatjuk:

```
{
   double 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 adattag kezelésére statikus tagfüggvényeket használhatunk. A statikus tagfüggvényekkel a normál adattagokat nem érhetjük el, hiszen a paraméterek között nem szerepel a **this** mutató. Megjegyezzük, hogy az osztályon belül **typedef** tárolási osztállyal definiált típust is elhelyezhetünk.

2.1.7. 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 adatmezőkhöz rendelt mutatók esetén is. A mutató helyes definiálásakor az osztály nevét és a *scope* 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, amelyet argumentum
                           // nélkül hívunk és int értéket ad vissza
```

A következő példában bemutatjuk az osztálytagokra mutató pointerek használatát. Ilyen mutatók használata 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;           // Mindhárom tagot mutató segítségével érjük el
   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)*=11;               // Az o2->a szorzása 11-el
   int iv=(o2->*gp)();          // Az o2->get() meghívása
   delete o2;
}
```

2.2. Konstruktorok és destruktorok

Ebben a részben két speciális tagfüggvénnyel ismerkedünk meg, melyek feladata az osztály példányainak inicializálása (konstruktor), illetve megszüntetés előtti „takarítása” (destruktor).

2.2.1. Konstruktorok

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 objektumok 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, amely elvégzi az osztály objektumainak inicializálását. A konstruktor nevének meg kell egyeznie az őt tartalmazó osztály nevével. 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 is átdefiniálható, így adott a lehetőség többféle inicializálás megvalósítására.

Fontos megértenünk, hogy a konstruktor nem foglal memóriát 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 *default* (*X::X()*) és a másoló (*X::X(const &X))* konstruktorral. A másoló konstruktort általában akkor definiáljuk, ha valamilyen dinamikus terület tartozik az osztály példányaihoz (mint a példánkban).

Ha a fenti osztálydefiníciót a *vektor.h* fejállományban tároltuk, 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];
}
```

Mind a négy konstruktor helyet foglal a memóriából, a vektor elemei számára. Nézzünk néhány példát a *Vektor* típusú objektumok definiálására!

```
// Az első (default) konstruktor működik - 10 elemű vektorok:
   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 - 20 elemű vektorok
   Vektor f = Vektor (20);
   Vektor g(20);
   Vektor h = 20;
   Vektor *i = new Vektor(20);
   delete i;
   // 3-elemű vektortömb létrehozása, melynek elemei 3, 7, 9 elemű Vektorok:
   Vektor j[]={3,7,9};

// A harmadik (másoló) konstruktor használata:
   Vektor k;
   Vektor l=k;
   Vektor m(k);
   Vektor n[3]={k,l,m};
   Vektor *o=new Vektor(k);
   delete o;

// 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 konstruktorok egyaránt lehet **public**, **private** és **protected** elérésűek. A csak **private** konstruktorokat tartalmazó osztályt privát 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 származtathatunk.

2.2.1.1. A konstruktorok explicit paraméterezése

Általában az egyparaméteres konstruktorral rendelkező osztályok példányainak olyan kifejezést adunk kezdőértékül, amely típusával illeszkedik a konstruktor paraméterének típusához. A fordító ekkor implicit konverziókat használ a megfelelő konstruktor kiválasztásához. Az **explicit** kulcsszó megadásával megtilthatjuk implicit konverziók használatát 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};
```

2.2.2. 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 terület felszabadításáról. A destruktor nevét a hullám karakterrel (~) egybeépített osztálynévként kell megadni. A destruktor, akárcsak a konstruktor, szintén nem rendelkezik visszatérési típussal.

A destruktort tartalmazó *Vektor* osztály, amelyben 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 destruktorait 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();*)

2.2.3. 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.

2.3. Operátorok átdefiniálása (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 **.** , a **.*** , a **::** , a **?:,** a **sizeof** és a **typeid** operátorok. Az operátorok átdefiniálásával nem változtatható meg az operátorok elsőbbsége és csoportosítása.

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++ operátorokat két csoportra oszthatjuk, az operandusok száma alapján. 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 átdefiniáltuk 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.

```
// Vektor.h
class Vektor {
  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;
}
```

2.3.1. A new és a delete operátorok átdefiniálása

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 definiálnunk. A második paramétert, amennyiben megadtuk, a fordító használja az első paramétert által megcímzett objektum méretének átadására.

Az átdefiniált **new** és **delete** operátorok mellett, a érvényességi kör operátorának megadásával mindig elérhetjük az eredeti **new** és **delete** operátorokat:

```
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.

2.3.2. 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 alapértelemzett paramétereket használunk:

```
cplx(double a=0, double b=0) {re=a; im=b;}
```

2.3.3. Az osztályok bővítése input/output műveletekkel

A C++ nyelv lehetővé teszi, hogy az osztályokon alapuló adatfolyamoknak „megtanítsuk” a 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: 13.7+26.4i, illetve 13.7-26.4i
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: 13.7+26.4i, illetve 13.7-26.4i
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<<"Kérek egy komlex számot: ";
   cin >>a;
   cout<<"A komplex szám: "<<a<<endl;
   cin.get();
}
```

2.4. Az öröklés (öröklődés) mechanizmusa

Az öröklés (*inheritance*) az objektum-orientált C++ nyelv legfőbb sajátossága. Ez a mechanizmus 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 származtathatunk.

2.4.1. 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. 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, függvényargumentum,...) 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 öröklött adattagok 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 az érvényességi kör operátort kell használnunk a hivatkozáshoz:

```
Alap::tag
```

Nézzünk egy példát az elmondottak szemléltetésére!

```
class B { // az alaposztály
   int x, y;
 public:
   int b1, b2;
   int Bfunc1(void) { return x;}
   int Bfunc2(void) { return y;}
};
```

```
class D: private B  {  // a származtatott osztály
   int b2;   // az öröklött b2 elfedése
   int d;
 public:
   B::b1;             // a private származtatással öröklött b1 public elérésű lesz
   void Bfunc1(void); // az öröklött Bfunc1() elfedése
};

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=13;
}
```

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()`

2.4.2. Az alaposztály inicializálása

Az alaposztály(ok) inicializálására a tag-inicializációs lista kibővített változatát használjuk. Példaként tekintsük az *Alap1* és *Alap2* osztályokból származtatott *Szarmaztatott* osztály konstruktorát:

```
Szarmaztatott::Szarmaztatott(int ct, char *ptr, long val, char st,
                             bool tp) : Alap1(ct, val), Alap2(tp, ptr) {
 // a származtatott osztály konstruktora
}
```

Az alaposztályok konstuktorai az inicializációs lista szerinti sorrendben hívódnak meg, 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 objektummal, vagy a származtatott osztály objektumával, ha a származtatást **public** módon végeztük. Az inicializálást a másoló konstruktor végzi (*X::X(const &X)*):

```
Szarmaztatott:: Szarmaztatott (int ct, Szarmaztatott & dc) : Alap(dc)
{
 // a származtatott osztály konstruktora
}
```

2.4.3. Virtuális tagfüggvények

A virtuális függvény olyan **public** vagy **protected** tagfüggvénye a **public** alaposztálynak, amelyet a származtatott osztályban újradefiniálhatunk az osztály tulajdonságainak megváltoztatása érdekében. A virtuális függvény általában a **public** alaposztály referenciáján vagy mutatóján keresztül kerül meghívásra, melynek aktuális értéke a program futása során alakul ki (dinamikus kapcsolás). 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 szerepjen. Ebben az esetben ún. tiszta virtuális függvénnyel (*pure virtual function*) van dolgunk:

```
class Pelda {
  public:
    virtual int pf()=0;
};
```

Egy vagy több tiszta virtuális függvényt tartalmazó osztállyal (*absztrakt osztállyal*) nem hozhatunk létre objektumpéldányt. Az absztrakt osztály csak alaposztályként használható.

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 az átdefiniálás (*overloading*) mechanizmusa érvényesül.

Az alábbi példaprogramban mindegyik alakzat saját maga számolja ki a területét és a kerületét, azonban megjelenítést az absztrakt alaposztályuk végzi:

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

class Alakzat { // Absztrakt alaposztály
  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<<"\tTerület: "<<terulet();
          cout<<"\tKerület: "<<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);}
};

void main()
{
     Negyzet n(13,7,10);
     cout<<"Négyzet: ";
     n.megjelenit();
     Teglalap t(4,7,10,20);
     cout<<"Téglalap: ";
     t.megjelenit();
     cin.get();
}
```

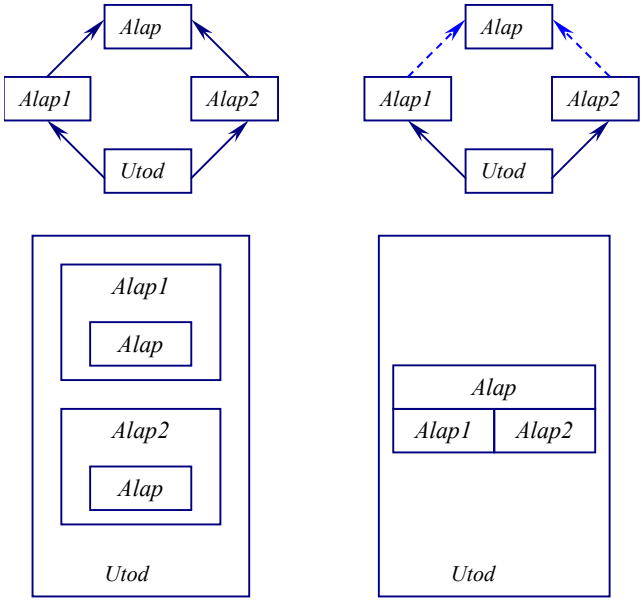
Virtuális tagfüggvényeket tartalmazó osztályok esetén a fordítóprogram az osztályhoz kapcsolódóan egy ún. virtuális táblát (VT) épít fel, amely az aktuális újradefiniált virtuális függvények címeit tartalmazza. Az osztályhierarchiában található azonos nevű virtuális függvények azonos indexszel szerepelnek ezekben a táblákban, ami lehetővé teszi a virtuális tagfüggvények teljes lecserélését.

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 kerül meghívásra, amikor az objektumot megszünik.

2.4.4. 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;
};

class Alap1 : virtual public Alap { // az Alap virtuális alaposztály
   int x;
 public:
   Alap1(int i) { x=i;}
};

class Alap2: public virtual Alap { // az Alap virtuális alaposztály
   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** definíció 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 a *base1* és a *base2* alaposztályok. A virtualitás miatt a *base* alaposztály egyetlen példányban szerepel, így a *base1::q* és a *base2::q* ugyanarra az adattagra hivatkoznak. A **virtual** szó használata nélkül a *base1::q* és a *base2::q* különböző adattagokat jelölnek.

2.5. Általánosított osztályok (templates)

Az paraméterezett (általánostott) 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.

Készítsük el a 2.2. és 2.3. fejezetekben használt Vektor osztály általánosított változatát!

```
// Vektor.h
#ifndef VektorH
#define VektorH
template <class TTip>
class Vektor {
  public:
    Vektor();
    Vektor(int n);
    Vektor(const Vektor& v);
    Vektor(const TTip a[],int n);
    ~Vektor() {delete[] p; }
    int fh() const { return meret; }
    TTip& operator[] (int i);
    Vektor& operator= (const Vektor& v);
    Vektor  operator+= (const Vektor& v);
private:
    TTip *p;
    int meret;
};
#endif

// Vektor.cpp
#include "Vektor.h"

template <class TTip> Vektor<TTip>::Vektor(void) {
    p = new TTip[meret = 10];
}

template <class TTip> Vektor<TTip>::Vektor(int n) {
    p = new TTip[meret=n];
}

template <class TTip> Vektor<TTip>::Vektor(const Vektor& v) {
     p = new TTip[meret=v.meret];
     for (int i = 0; i < meret; ++i)
        p[i] = v.p[i];
}

template <class TTip> Vektor<TTip>::Vektor(const TTip a[], int n) {
    p = new TTip[meret=n];
    for (int i = 0; i < meret; ++i)
       p[i] = a[i];
}

template <class TTip> TTip& Vektor<TTip>::operator[](int i) {
    if (i < 0 || i > meret-1)
       throw i;
    return p[i];
}

template <class TTip> Vektor<TTip>& Vektor<TTip>::operator=(const Vektor<TTip>& v) {
    delete []p;
    p=new TTip [meret=v.meret];
    for (int i = 0; i < meret; ++i)
       p[i] = v.p[i];
    return *this;
}
```

```
template <class TTip> Vektor<TTip> Vektor<TTip>::operator+=(const Vektor<TTip>& v) {
    int m = (meret < v.meret) ? meret : v.meret;
    for (int i = 0; i < m; ++i)
       p[i] += v.p[i];
    return *this;
}

// Külső függvény
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 specializálását, vagyis konkrét típus behelyettesítésével való megvalósítását:

```
#include "Vektor.cpp"
void main() {
   int a[5]={1,2,3,4,5};
   int b[3]={10,20,30};
   typedef Vektor<int> iVektor;       // Specializáció typedef-ben
   iVektor v1(a,5), v2(b,3), v3;
   v3=v1;
   v3+=v2+v1;
   Vektor<double> x,y,z;            // Lokális specilaizáció double típussal
   for (int i=0; i<x.fh(); i++)
     x[i]=123.12/(5.3+i);
   y=x;
   z=x+y;
}
```

Az osztálysablonnak is lehetnek barátai, melyek többféleképpen viselkedhetnek. Azok amelyek nem tartalmaznak template előírást, minden specializált osztály közös barátai lesznek. Ellenkező esetben a külső fügvény csak az adott specializált változat *friend* függvényeként használható. Az általánosított osztályban definiált statikus tagok specializációnként jönnek létre.

A következő egyszerű osztálysablont a típuson kívül értékkel is paraméterezzük:

```
#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());
}
```

2.5.1. 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);
}
```

2.6. Futás közbeni típusinformációk (RTTI) osztályok esetén

A különböző vizuális fejlesztőrendszerek futás közbeni típusinformációkat 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 elérni.

```
#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<<" lába van."<<endl;
      }
};

class Hal : public Allat {
  protected:
     string fajta() {return "hal";}
  public:
     Hal(int n=0) : Allat(n) {}
     void Uszik() {cout<<"Úszik"<<endl;}
};

class Madar : public Allat {
  protected:
     string fajta() {return "madár";}
  public:
     Madar(int n=2) : Allat(n) {}
     void Repul() {cout<<"Repül"<<endl;}
};

class Emlos : public Allat {
  protected:
     string fajta() {return "emlős";}
  public:
     Emlos(int n=4) : Allat(n) {}
     void Fut() {cout<<"Fút"<<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]))  // Emlősl?
             dynamic_cast<Emlos*>(p[i])->Fut();

    for (int i=0; i<db; i++)
        delete p[i];
    cin.get();
}
```

3. 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.

AC++ 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 ossztályok	<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> <vektor>
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>
Sztring-adatfolyamok	<sstream> <cstdlib>
Fájl-adatfolyamok	<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 neveit 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>