Herbert Schildt

C/C++

Referenciakönyv

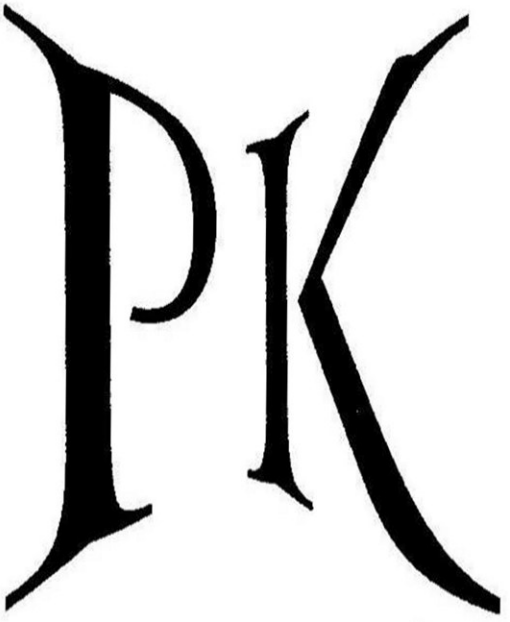
PK

PROGRAMOZÓK KÖNYVTÁRA

Herbert Schildt

C/C++

Referenciakönyv



PROGRAMOZÓK
KÖNYVTÁRA

Herbert Schildt

C/C++

Referenciakönyv

PANEM

A mű eredeti címe: C/C++: Programmer's Reference. Osborne McGraw-Hill,
2600 Tenth Street, Berkley, California 94710 U.S.A.

Panem Kft.
1385 Budapest, Pf. 809
Hungary

ISBN 963 545 178 4
ISSN 1417-0906

A kiadásért felel a Panem Kft. ügyvezetője, Budapest, 1998

Fordította: Morvay Gábor
Lektorálta: Csányi Kornél
Grafika és borítóterv: Érdi Júlia
Műszaki szerkesztő: Kelemen András

A Panem könyvek megrendelhetők a 06-30/488-488 hívószámú telefonon,
illetve a 1385 Budapest, Pf. 809 levélcímen.
panem@mail.datanet.hu
http://www.datanet.hu/panem

Tartalomjegyzék

Bevezetés

A C és a C++ a világ legfontosabb programozási nyelvei. Valóban, egy mai profi programozónak létszükséglete e két nyelv mélyreható ismerete. Ezek a modern programozás alapkövei.

A C-t Dennis Ritchie alkotta meg a 70-es években. A C egy középszintű programozási nyelv. Ötvözi a magas szintű nyelvek vezérlési szerkezeteit és a bitek, byte-ok, pointerek (címek) közvetlen manipulálásának lehetőségeit, ami az alacsony rendű programozási nyelvek sajátja. Ez az oka a rendkívüli flexibilitásnak, amely szinte teljhatalmat ad a programozó kezébe. A C nyelvet 1989 végén szabványosították az American National Standards Institute (ANSI = Amerikai Nemzeti Szabványügyi Hivatal) jóváhagyásával. Ezt a szabványt később – 1990-ben – az ISO (International Standards Organization = Nemzetközi Szabványügyi Hivatal) is elfogadta. 1996-ban a szabvány néhány új elemmel bővült.

A C++ nyelvet Bjarne Stroustrup fejlesztette ki az 1980-as években. Kezdetben nem volt más, mint a C objektumorientált irányban történő kiterjesztése, de hamarosan különálló programozási nyelvvé nőtte ki magát. A C++ ma is a C leszármazottjának tekinthető, azonban nyelvi lehetőségei megduplázódtak elődjéhez képest. Nem szükséges mondani, hogy a C++ – mind a mai napig – egyike a leghatékonyabb programozási eszközöknek. Nemsokára várható egy ANSI/ISO standard elfogadása, így a C++ minden szempontból szabványosítva lesz.

Ezen könyv tárgya az ANSI-szabvány C és az ANSI/ISO szabványügyi bizottság által készített legfrissebb C++ szabvány.

Bizonyára az olvasó is tisztában van vele, hogy a C és C++ teljes körű ismertetése nem kis feladat. A könyv – érthető módon – nem is tekintheti céljának e két nyelv minden fontos részletének tárgyalását. Ehelyett arra törekedtünk, hogy a leglényegesebb jellemzőket kiszűrjük és azokat egy kényelmes, könnyen használható formában tálaljuk az olvasónak.

1. FEJEZET

Adattípusok, változók és konstansok

C és C++ beépített adattípusok gazdag választékát nyújtja a programozónak. Felhasználói adatszerkezetek létrehozásával pedig gyakorlatilag minden igényt ki lehet elégíteni. Bármely érvényes adatszerkezetből változókat hozhatunk létre. Konstansok definiálhatók minden beépített C/C++ típusból. Ebben a fejezetben a különböző adattípusokkal, változókkal és konstansokkal kapcsolatos tudnivalókat foglaljuk össze.

■ *Elemi típusok*

A C nyelvben a következő öt elemi típus áll a programozó rendelkezésére:

Típus	*Kulcsszó*
karakter	char
egész	int
lebegőpontos	float
dupla lebegőpontos	double
érték nélküli	void

A C++-ban ezek a további kettővel bővülnek:

Típus	*Kulcsszó*
logikai (igaz/hamis)	bool
Hosszú karakter	wchar_t

Az elemi típusok közül számos olyan van, amelyek módosíthatóak az alábbi kulcsszavak használatával:

signed (előjeles)
unsigned (előjel nélküli)
short (rövid)
long (hosszú)

A típusmódosítókat mindig a módosítandó típus neve elé tesszük. A C és a C++ összes lehetséges módosítókkal kombinált predefinit adattípusát az alábbi táblázatban láthatjuk, amely egyúttal megadja ezek értéktartományát is. A legtöbb fordító egyes típusoknál szélesebb tartományokat használ. Ugyanakkor jó, ha szem előtt tarjuk azt is, hogy a kettes komplemens ábrázolásmódot használó gépeknél (a legtöbb ilyen) a legkisebb tárolható negatív szám az előjeles egész típusok esetén eggyel kisebb a táblázatban megadottnál. Például az **int** típus a legtöbb számítógépen –32 768 és 32 767 közötti egész értékeket foglalja magában. A **char** típus előjeles vagy előjel nélküli mivolta implementációfüggő.

Típus	*Minimális értéktartomány*
bool	true/false
char	–127-től 127-ig vagy 0-tól 255-ig
unsigned char	0-tól 255-ig
signed char	–127-től 127-ig
int	–32 767-től 32 767
unsigned int	0-tól 65 535-ig
signed int	ugyanaz, mint az int
short int	ugyanaz, mint az int
unsigned short int	0-tól 65 535-ig
signed short int	ugyanaz, mint a short int
long int	–2 147 483 647-től 2 147 483 647-ig
signed long int	ugyanaz, mint a long int
unsigned long int	0-tól 4 294 967 295-ig
float	6 tizedesjegy pontosság
double	10 tizedesjegy pontosság
long double	10 tizedesjegy pontosság
wchar_t	ugyanaz, mint az unsigned int

Ha egy típusmódosítót önmagában használunk, azt a fordító úgy értelmezi, mintha mögötte az int kulcsszó szerepelne. Például megadhatunk egy előjel nélküli egészet egyszerűen az **unsigned** kulcsszó segítségével. Így az alábbi két deklaráció ekvivalens:

```
unsigned int i; // itt az int-et kiírtuk
unsigned i; // itt az int odaképzelendő
```

■ *Változók deklarálása*

Minden változót deklarálnunk kell még az első hivatkozás előtt. A deklaráció általános alakja így néz ki:

típus változó_név;

Például, ha létre akarunk hozni egy **x** nevű lebegőpontos változót, egy **y** nevű egészt és egy **ch** azonosítójú karaktert, akkor tegyük a következőt:

```
float x;
int y;
char ch;
```

Vesszővel elválasztott lista formájában egyszerre több változót is deklarálhatunk egy adott típusból. Az alábbi példa három változót hoz létre:

```
int a, b, c;
```

Változók inicializálása

Egy változó kezdeti értékének megadása úgy történhet, hogy a deklarációban a változónév után egy egyenlőségjelet követően leírjuk a kezdeti értéket. Az alábbi deklaráció a **count** nevű változó értékét 100-ra állítja be:

```
int count = 100;
```

Inicializálóként, azaz az egyenlőség jobb oldalán bármilyen kifejezés szerepelhet, amely a deklaráció megtételekor érvényes. Ez magába foglalja a függvényhívásokat és más változók használatát. A C-ben azonban a globális változókat és a **statikus** lokális változókat csak konstans kifejezések segítségével inicializálhatunk.

■ *Azonosítók*

A változó, függvény és felhasználó által definiált függvénynevek mind jó példák az *azonosítókra*. A C/C++-ban az azonosítók betűk sorozatából, számjegyekből és aláhúzás jelekből állnak, hosszuk egy vagy több karakter lehet. (Számjeggyel nem kezdődhet azonosító.) Az azonosítók bármilyen hosszúak lehetnek, de csak az első 31 karakter figyelembevétele garantált. Az aláhúzást a jobb érthetőség kedvéért szokták használni, mint pl. a **havi_fizetés** azonosítóban vagy esetleg a név első karaktereként, mint a **_count** azonosítóban. A kis- és nagybetűk között a fordító különbséget tesz. Ezért például a **test** és a **TEST** két különböző változó. Fontos megjegyeznünk, hogy a C++ lefoglal minden olyan változónevet, amely két aláhúzással vagy egy aláhúzással és egy nagybetűvel kezdődik.

■ *Osztályok*

Az OOP-beli (objektumorientált programozás) zártság elvét a C++-ban az osztály képviseli. Osztályokat a **class** kulcsszó segítségével definiálhatunk. Az osztályok nem tartoznak a C nyelvhez. Az osztály alapvetően nem más, mint bizonyos változók és az ezeket manipuláló függvények gyűjteménye. Az osztályt alkotó függvényeket és változókat közös néven *tagoknak* is hívják. Egy osztály általános szintaktikája így néz ki:

```
class osztálynév: öröklődési_lista {
   //privát tagok
 protected:
   //olyan privát tagok, amelyekből leszármazottak hozhatók létre
```

```
  public:
    //nyilvános tagok
} objektumlista;
```

Az *osztálynév* nem más, mint az osztály típusának a neve. Az osztálydeklaráció lefordítása után az *osztálynév* egy új adattípus neve lesz, amelynek segítségével példányokat (objektumokat) hozhatunk létre az osztályból. Az *objektumlista* egy olyan objektumazonosító lista, amelyben a szomszédos azonosítókat vessző választja el. Az objektumlista segítségével az osztálydefinícióval egy időben deklarálhatunk példányokat. Ez elhagyható, mert a deklarációk a program későbbi részeiben is megtehetők a szokott módon, az osztálynév feltüntetésével. Az öröklődési_lista szintén opcionális. Megadásával azon alaposztályokat tüntetjük fel, amelyekből az új osztály származik (l. lejjebb az *Öröklődés* c. fejezetet).

Az osztály tartalmazhat két speciális függvényt az ún. *konstruktort* és a *destruktort* (mindkettő opcionális). A konstruktor automatikusan végrehajtódik, amikor az osztály egy új példányát hozzuk létre. A destruktor az objektum megszűnésekor kerül meghívásra. A konstruktor az osztály nevét viseli. A destruktor nevét szintén az osztálynévből képezzük, de azt meg kell hogy előzze egy tilde (~) jel. Sem a destruktoroknak, sem a konstruktoroknak nincsen visszatérési értékük. Egy öröklődési hierarchiában a konstruktorok mindig a származtatás sorrendjében hajtódnak végre, míg ez destruktorok esetén éppen fordítva történik.

Alapértelmezésben az osztály minden tagja privát, azaz csak ezen osztály más tagjai számára látszanak. Ahhoz, hogy egy adott tag olyan függvények által is hozzáférhető legyen, amelyek nem tagjai az osztálynak, deklarálásukat megelőzően fel kell tüntetnünk a **public** kulcsszót. Például:

```
class myclass {
  int a, b; // Csak ez az osztaly lathatja
public:
  // Elerheto kívülrol is
  void setab(int i, int j) { a = i; b = j; }
  void showab() { cout << a << '' << b << endl; }
{} ;

myclass ob1, ob2;
```

Ez a deklaráció létrehoz egy osztálytípust: a **myclass**-t, amely két privát változót tartalmaz: **a**-t és **b**-t. Az osztályban két publikus függvény is található a **setab( )** és a **showab( )**. A fenti részlet egyúttal két **myclass** típusú példányt is deklarál: az **ob1**-et és az **ob2**-t. Előfordulhat, hogy egy privát tagot szeretnénk tovább örökíteni, ekkor a **protected** kulcsszóra van szükségünk, ezt is a tagok deklarálása előtt kell feltüntetnünk. Ha egy osztály adott objektumán akarunk műveletet végezni, akkor a „." operátoron keresztül érhetjük el. A „->" operátort használjuk, ha az objektumra egy mutatón keresztül kívánunk hivatkozni. Az alábbi kódrészletben példát láthatunk mindkét operátor használatára, az **ob putinfo( )** függvényét a pont operátoron keresztül, míg a **show( )** függvényt a nyíl operátoron keresztül hívjuk meg.

```
struct c1_type {
  int x;
  float f;
public:
  void putinfo(int a, float t) { x = a; f = t; }
  void show() { cout << a << ' ' << f << endl; }
} ;

c1_type ob, *p;
// . . .
ob.putinfo(10, 0.23);
p = &ob; // ob címét bemásolja p-be
p->show() ; // ob adatait kiírja
```

Lehetőség van osztálysablonok vagy ún. generic-ek létrehozására a **template** kulcsszó használatával. (Lásd a „**template**" kulcsszó alatt a Kulcsszavak összefoglalása c. fejezetben.)

■ *Öröklődés*

A C++-ban az osztályok átvehetnek más, korábban definiált osztályok bizonyos jellemzőit; ez az öröklődés. Azt az osztályt, amelyből származtatunk, *ős-* vagy *alaposztálynak* nevezzük. Az ősosztályból származtatott

osztályra a *gyermekosztály* vagy a *származtatott osztály* elnevezést használja a szakirodalom. Ha egy osztályból másikat származtatunk, *osztályhierarchiáról* beszélhetünk. A származtatás általános szintaxisa a következő:

class *osztálynév*: *láthatóság alaposztálynév* {
    //...
};

A *láthatóság* meghatározza, hogy milyen módon öröklődik az alaposztály – lehet **private**, **public** vagy **protected**. (Amennyiben a *láthatóság* helyét üresen hagyjuk, a fordító automatikusan **public**-ot tételez fel, ha az alaposztály **struct** címkével van ellátva, **class** megjelölés esetén **private** az alapértelmezés.) Ha több osztályt szeretnénk egyszerre származtatni, akkor az ősök neveit vesszővel elválasztva kell felsorolnunk.

Ha *láthatóság*ként **public**-ot adunk meg, akkor az alaposztály minden **public** és **protected** tagja a származtatott osztálynak is **public**, illetve **protected** tagja lesz. Ha a *láthatóság* **private**, akkor az alaposztály valamennyi **private** és **protected** tagja az új osztályban **privateként** jelenik meg. Ha a *láthatóság* **protected** értéket kap, akkor az alaposztály **public** és **protected** tagjai **védetté** válnak a származtatott osztályban.

Az alábbi példában egy egyszerű osztályhierarchiát hoztunk létre. A *származtatott* nevű osztály az *alap* nevű osztály gyermeke, a származtatás a **private** módon történt. Ez azt fogja jelenteni, hogy az *i* változó a *származtatott* privát tagja lesz.

```
Class alap {
public:
  int i;
};
class szarmaztatott: private base {
  int j;
public:
  szarmaztatott (int a) {j = i = a;}
  int getj() {return j;}
  int geti() {return i;} //jó, mert a származtatott-
                         //ból látszik i
};
```

```
szarmaztatott ob (9); // itt jön létre a származta-
                      //tott egy példánya

cout << ob.geti() << ""<< ob.getj(); //jó

//ob.i = 10; // HIBA, i privát tagja a származta-
             //tottnak
```

■ *Struktúrák*

Struktúrákat a **struct** kulcsszó segítségével hozhatunk létre. A C++ -ban a struktúrák egyben osztályokat is definiálnak. Az egyedüli különbség a **struct** és a **class** között, hogy alapértelmezésben a struktúra minden tagja nyilvános. Ahhoz, hogy egy tagot priváttá tegyünk, a **private** kulcsszót kell használnunk. Egy struktúradeklaráció általános formája így néz ki:

struct strukt-név: öröklődési_lista {
  //nyilvános tagok, ez az alapértelmezés, nem kell külön megjelölni
protected:
  //privát tagok, amelyek azonban öröklődhetnek
private:
  //privát tagok
} *objektumlista;*

A C-ben számos megszorítás vonatkozik a struktúrákra. Először is csak adattagokat tartalmazhatnak, azaz függvénytagok nem megengedettek. C-beli struktúrák nem támogatják az öröklődést. Továbbá minden tag nyilvános (**public**), és a **protected**, illetve a **private** kulcsszavak nem használhatók.

■ *Unionok*

Az *union* egy olyan osztálytípus, amelyben minden adattag egyazon memóriaszeleten osztozik. A C++-ban az union tartalmazhat mind adattagokat, mind függvényeket. Alapértelmezésben minden tag nyilvános.

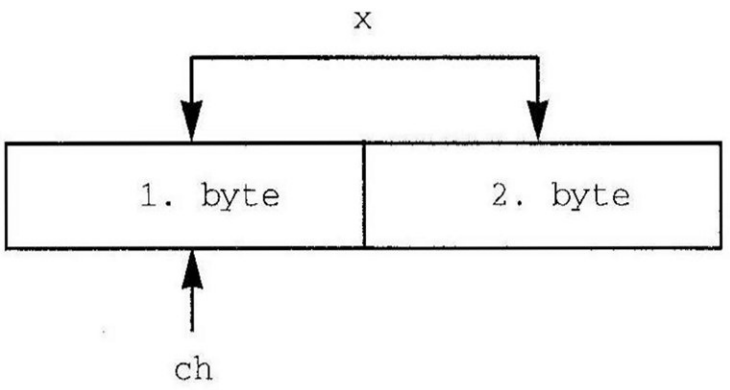
Privát elemek létrehozása a jól ismert **private** kulcsszóval történik. Az union deklarálásának általános formája:

```
union osztálynév {
  // nyilvános tagok, ez az alapértelmezés, nem kell külön megjelölni
private:
  // privát tagok
} objektumlista
```

C-ben az unionok kizárólag adattagokat foghatnak össze és a **private** kulcsszó használata nem megengedett. Az union elemei átfedik egymást. Például a

```
union tom {
  char ch;
  int x;
} t;
```

részlet a **tom** azonosítójú uniont deklarálja, amelynek memóriafelhasználását (2-byte-os egészeket feltételezve) az alábbi ábra illusztrálja:



Hasonlóan, mint ahogy azt osztályok esetén már megszokhattuk, az union egyes komponenseire a pont („.") operátor segítségével hivatkozhatunk. A nyíloperátort akkor használjuk, ha egy mutatón keresztül kívánjuk elérni az adott elemet.

Az unionokra számos megkötés vonatkozik. Az unionok nem származtathatóak semmiféle osztályból. Union nem lehet alaposztály. Az unionoknak nem lehetnek virtuális függvénytagjai. Semelyik tag sem deklarál-

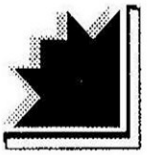
ható statikusként. Az union nem tartalmazhat tagként olyan objektumot, amely túlterheli az = operátort. Végezetül nem szerepelhet tagként olyan objektum, amelynek osztályában explicit módon definiált konstruktor vagy destruktor található. (Objektumok, amelyek az alapértelmezett konstruktorral, illetve destruktorral rendelkeznek, nem esnek ezen korlátozás alá.)

A C++ támogat egy speciális uniontípust, az ún. *anonim uniont*. Az anonim union típusokban nem szerepel osztálynév és nem deklarálható semmilyen példánya az unionnak. Ehelyett az történik, hogy a deklaráció jelzi a fordítónak, hogy a tagként felsorolt változókat egyazon tárterületen tárolja. A változókra közvetlenül, nevükön keresztül hivatkozhatunk, a szokásos pont és a nyíl használatának nincs értelme. Az anonim uniont alkotó változók hatásköre megegyezik bármely, ugyanabban a blokkban deklarált változó hatáskörével. Ebből következik, hogy az anonim unionban szereplő változóneveknek nem szabad ütközniük a blokkban a hatáskörükben érvényes más változónevekkel. Lássunk most egy példát az anonim unionra:

```
union { // anonim union
  int a; // a és f
  float f; // átfedi egymást a memóriában
};
// . . .
a = 10; // a-t manipulálja
cout << f; // f-et kérdezi le
```

Az **a** és az **f** változók közösen használnak egy tárterületet. Látható, hogy az union változóira közvetlenül hivatkozhatunk, nincs szükség a pont-, illetve a nyíloperátor használatára.

Minden korlátozás, ami az unionra vonatkozik, fennáll az anonim unionok esetében is. Ehhez jön még, hogy az anonim unionban csak adattag szerepelhet – nem tartalmazhat függvénytagot. Anonim unionokban nem használhatók a **private** és a **protected** kulcsszavak. Végül a névtér (namespace) hatáskörrel rendelkező anonim unionokat statikusként kell deklarálni.



Programozási tipp
A C++-ban bevett szokás, hogy a **struct***-ot akkor használjuk, amikor olyan C stílusú struktúrára van szükségünk, amely csak adattagokat tartalmaz. A* **class** *használata inkább olyankor célszerű, amikor függvénytagokat is szerepeltetni akarunk. A szakirodalom gyakran használja a POD mozaikszót a C stílusú struktúrák megnevezésére, ez a Plain Old Data (magyarul: egyszerű régi adatok) kifejezésből jön.*

■ *Felsorolt típus – enumeráció*

Az adattípusok egy gyakran használt osztályát alkotják a felsorolt típusok vagy enumerációk. Egy felsorolt adattípus tulajdonképpen nem más, mint névvel ellátott egészkonstansokból álló lista. Az enumeráció megadásához egyszerűen elegendő felsorolni a használni kívánt neveket.

Az felsorolt típus definiálása az **enum** kulcsszó segítségével történik, általános formában így:

enum *enum_név* {*névlista*} *változólista;*

Az *enum_név* a felsorolt típus azonosítója. A *névlista* vesszővel elválasztott azonosítókat jelent, ezek a típus tagjai.

Az alábbi programrészlet például egy városokból álló enumerációt definiál, melynek neve **városok**. A **v** változó a „Kecskemét" értéket kapja.

```
enum varosok {Budapest, Kecskemet, Debrecen} v;
v=Kecskemet;
```

Egy felsorolt típusban a legelőször szereplő elem értéke 0, a másodiké 1, a harmadiké 2 és így tovább. Általában minden elem számértéke eggyel nagyobb, mint az előtte lévő elemé. Lehetőségünk van azonban az alapértelmezett hozzárendelésen változtatni. Egy névhez rendelt számérték tetszés szerint inicializálható. Az alábbi enumerációban Kecskemétnek az értéke 10 lesz.

```
enum varosok {Budapest, Kecskemet=10, Debrecen} ;
```

Ebben a példában **Debrecen** értéke 11 lesz, mert a fent leírtak alapján eggyel nagyobb kell, hogy legyen, mint az őt megelőző **Kecskemété**.

A C++-ban egy felsorolt típusú változónak szigorúan csak a típusdefinícióban felsorolt elemeket adhatjuk értékül. C-ben egy enumerációs objektum egész értéket is felvehet.

■ *Címkék a C-ben*

C-ben nem igaz, hogy a struktúrák, unionok és enumerációk nevei egy teljes típusnevet is definiálnának. A C++-ban igaz. Az alábbi példa helyes C++-ban, de hibás C-ben.

```
struct s_type {
  int I;
  double d;
};
//...
s_type x; // C++-ban működik, de helytelen C-ben
```

C++-ban a fenti kód egy **s_type** azonosítójú típust hoz létre és ettől kezdve módunkban áll ilyen típusú változókat deklarálnunk. C-ben az **s_type** csak egy *címke* (nem egy teljes típusnév). Objektumok deklarálásakor a címkét meg kell hogy előzze a **struct**, **union** vagy **enum** kulcsszavak valamelyike. Például:

```
struct s_type x; //így már C-ben is működik
```

Az előző szintaxis használata C++-ban is megengedett, de ritkán használják.

■ *A Tárolásiosztály-specifikátorokról*

Tárolásiosztály-specifikátorok **extern**, **auto**, **register**, **static** és **mutable** segítségével utasíthatjuk a fordítót, hogy egy adott változó esetében térjen el a szokásos tárolási módtól. A specifikátorok mindig a típusazonosító előtt állnak a deklarációban.

extern

Ha egy változónevet az **extern** specifikátor előz meg, akkor a fordító tudni fogja, hogy egy külső linkből származó változóról van szó, vagyis, hogy a változó egy másik forrásállományban van definiálva, de láthatósági köre túlnő azon. Az **extern** kulcsszó feladata, hogy jelezze a fordítónak a változó típusát anélkül, hogy újabb tárterületet foglaltatnák le számára. Legtöbbször olyan esetben használjuk, amikor ugyanarra a globális változóra több forrásállományban is szükségünk van.

auto

Az **auto** specifikátor jelzi a fordítónak, hogy az őt követő lokális változó létrehozása az aktuális blokkba való belépéskor történjen meg, majd amikor a végrehajtás elhagyja a blokkot, szabadítsa fel a lefoglalt memóriát. Mivel a függvényeken belül definiált változók alapértelmezésben így viselkednek, az **auto** kulcsszót ritkán (ha egyáltalán) használják.

register

A C nyelv alkonyán a **register** specifikátor csak egész vagy karakter típusú változókra vonatkozhatott, akkor a **register** használatával arra adhattunk utasítást a fordítónak, hogy kísérelje meg a változó értékét a processzor egy regiszterében tárolni, ahelyett, hogy ezt a memóriában tenné. Az adott változó elérése így nagyságrendekkel gyorsabb lett. A **register** definícióját azóta kiterjesztették. Ma már minden változó ellátható a **register**módosítóval, a fordító feladata, hogy optimalizálja a hozzáférés sebességét ezen változókra. Karakterek és egészek esetén most is a CPU-regiszterek használata jelenti a leggyorsabb elérés biztosítását, más adattípusok esetén viszont a gyorsítás jelentheti például a cache memória (gyorstár) használatát.

Fontos megjegyeznünk, hogy a **register** használata csupán csak egy kérést jelent. A fordítónak jogában áll ezt figyelmen kívül hagyni. Ennek magyarázata, hogy a sebességet csak korlátozott számú változóra lehet optimalizálni. Amint a korlátot átlépjük, a kérések teljesítetlenek maradnak.

Static

A **static** specifikátor utasítja a fordítót, hogy az utána álló lokális változó élettartamát a program futásának befejeztéig biztosítsa. Így a változót nem kell minden alkalommal létrehozni, amikor a végrehajtás az érvényességi körén belülre kerül és megszüntetni, amikor kilép onnan. Ezért egy változó statikussá tétele azt is jelenti, hogy értéke megőrződik két függvényhívás között.

A **static** módosító globális változókra is alkalmazható. Ez a változó érvényességi körét arra az állományra szűkíti le, amelyben deklaráltuk, azaz kizárjuk a változó exportálásának lehetőségét.

A C++-ban, ha a **static**-ot egy osztály adattagjára alkalmazzuk, akkor az közössé válik az osztály minden példánya számára, azaz a tagot nem az egyes objektumokhoz kötjük, hanem magához az osztályhoz.

mutable

A **mutable** specifikátor csak a C++-ban létezik, egy objektum tagját mentesíti a **const** hatálya alól. Egy konstans objektum **mutable** specifikátorral megjelölt tagja módosíthatóvá válik.

■ *A típuskvalifikátorokról*

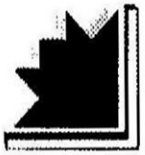
A **const** és **volatile** típuskvalifikátorok további információval szolgálnak az őket követő változóról.

const

Azok az objektumok, amelyeket a **const** kulcsszóval deklarálnak nem változtathatók meg a program futása során. Hasonlóan, egy **const** minősítővel ellátott mutató által referált objektumot nem módosíthatunk. Ezen kvalifikátor használata felhatalmazza a fordítót, hogy az ilyen objektumokat a csak-olvasható (ROM) tárba helyezze el. A konstans változók értéküket vagy inicializálással kapják vagy a hardver egy jellemzőjét hordozzák. Például a

```
const int a = 10;
```

sor egy **a** nevű egészt állít elő, amelynek értéke 10 és amelyet a program nem módosíthat. Kifejezésekben természetesen felhasználható.



Programozási tipp
Egy osztály függvénytagjának ***const*** *kvalifikátorral történő deklarálása megtiltja a függvénynek, hogy a hívó objektumot módosítsa. Egy ilyen függvénytag deklarálásakor a* ***const*** *szót a paraméterlista mögé kell beszúrnunk. Például:*

```
class MyClass {
  int i;
public:
  //egy const típusú függvény
  void wrong(int a) const {
    i = a; //Hiba! Nem  módosíthatjuk a hívó
           //objektumot
  }
  void right (int a) {
    i = a; //Helyes, ez nem const típusú függvény
  }
};
```

Ahogyan a megjegyzésből is kitűnik a ***wrong()*** *egy* ***const*** *függvény, tehát nem módosíthatja a* ***MyClass a*** *változóját.*

volatile

A **volatile** minősítő azt jelzi a fordítónak, hogy a mögötte szereplő változó értéke úgy is megváltozhat futás közben, hogy explicit módon arra nem ad utasítást egyetlen programsor sem. Ilyen lehet például, amikor az operációs rendszer óra rutinjának adódik át egy globális változó címe, amelynek tartalmát aztán minden ciklusban frissíti. Ebben az esetben a változó tartalma explicit értékadás nélkül változik meg. A **volatile** feltüntetése ilyenkor azért fontos, mert a fordító bizonyos kifejezések optima-

lizálása során feltételezi, hogy az egyes változók értéke nem változik a kifejezésen belül. Ezt a jobb hatásfok elérése érdekében teszi. A **volatile** módosító megakadályozza ezt a fajta optimalizálást azokban a ritka esetekben, amikor ez a feltevés nem helytálló.

■ *Tömbök*

Minden lehetséges adattípusból készíthetünk tömböt. Az egydimenziós tömbdeklaráció általános formája a következő:

típus *változónév[méret];*

Ahol a *típus* a tömb elemeinek közös típusát jelenti, a *méret* az elemek számát adja meg. Az alábbi példában egy 100 elemű egydimenziós egészekből álló tömb deklarációját láthatjuk:

```
int x[100];
```

x-ben egy olyan 100 hosszú tömböt hoztunk létre, melynek első eleme a 0, utolsó eleme a 99 indexet kapja. Az alábbi programrészlet az **x** tömböt feltölti a 0 és 99 közti egész számokkal.

```
for (t=0; t<100; t++) x[t] = t;
```

Tömböt minden érvényes adattípusból létrehozhatunk, beleértve a programozó által definiált osztályokat is.

Hasonlóképpen többdimenziós tömböket is definiálhatunk, ezt a megfelelő számú szögletes zárójel hozzáadásával végezhetjük. Például egy 10x20-as egésztömb deklarálása így néz ki:

```
int x [10][20];
```

Tömbök inicializálása történhet az elemek kezdő értékeinek kapcsos zárójelpáron belüli felsorolásával. Például:

```
int count[5] = {1,2,3,4,5};
```

■ *Új típusnevek definiálása a typedef kulcsszóval*

Programunkban egy már meglévő típust szerepeltethetünk új néven. Erre lehetőséget a typedef kulcsszó biztosít. Használata általános esetben:

typedef *típus újnév;*

Az alábbi kódrészlet közli a fordítóval, hogy ezentúl a **meter** azonosító jelentse ugyanazt, mint az **int** típusnév.

```
typedef int meter;
```

Így értelmet nyer már a következő deklaráció is, amely egy **tavolsag** nevű egész változót hoz létre.

```
meter tavolsag;
```

■ *Konstansok*

Konstansok vagy más néven *literálok* rögzített értékeket képviselnek, amelyeket a program nem változtathat meg. Minden alaptípusból létezik konstans. Az egyes konstansok reprezentálása típusfüggő. A karakter konstansokat macskakörmök közé tesszük. Például az 'a' és a '+' karakter konstans. Az egész-konstansokat a törtrész nélküli számok jelentik. Például a 10 és a –100 egész konstans. Lebegőpontos konstansoknál szükséges a tizedespont használata és adott esetben a törtrész feltüntetése. Például 11.123 egy lebegőpontos konstans. Lebegőpontos konstansok megadáskor használható még a normál alak is.

A C-ben két lebegőpontos típus létezik a **float** és **double**. Ezek kívánság szerint tovább finomíthatók a módosítók használatával. Alapértelmezésben a fordító a lehető legszűkebb kompatíbilis típust használja, ami tartalmazza numerikus konstansként az adott literált. Ez alól az egyedüli kivételt a lebegőpontos konstansok jelentik, amelyeket a fordító mindig **double**-ként értelmez. A legtöbb program esetén az alapértelmezés teljesen jól működik. Lehetősége van azonban a programozónak a konstans típusának precíz megjelölésére.

A pontos típusmegjelölés úgy történik, hogy a konstanshoz a típusnak megfelelő szuffixumot (utótagot) illesztünk. Ha egy lebegőpontos konstanst **F** betű követ, akkor a szám **float**-ként fog viselkedni. Ha **L** betű áll a szám után, akkor azt a fordító **long double**-ként kezeli. Egész típusokra az **U** szuffixum az **unsigned**-ot, az **L** utótag a **long**-ot jelöli. Néhány példát összegyűjtöttünk:

Adattípus	*Példák konstansokra*
int	1 123 21000–234
long int	35000L–34L
unsigned int	10000U 987U
float	123.23F 4.34e–3F
double	123.23–0.9876324
long double	1001.2L

Hexadecimális és oktális konstansok

Gyakran megkönnyíti a dolgunkat, ha a tízes számrendszer helyett a nyolcas vagy tizenhatos számrendszert használjuk. A nyolcas alapú számrendszert *oktálisnak* hívjuk. Itt a számokat a 0, 1, ..., 7 számjegyekkel írjuk le. Az oktális rendszerben a 10 a decimális 8-nak felel meg. A tizenhatos alapú számrendszert *hexadecimálisnak* nevezzük. Itt nem elegendő a 0, 1, ..., 9 számjegyek használata, ezeket betűkkel egészítjük ki A-tól egészen F-ig. Az A 10-nek, a B 11-nek, a C 12-nek, ..., az F 15-nek felel meg. Például a hexadecimális 10 ugyanaz, mint a decimális 16. E két számrendszer praktikussága miatt, a C/C++ támogatja az oktális, illetve hexadecimális konstansok használatát, megkönnyítve azok dolgát, akik bizonyos esetekben előnyben részesítik ezeket. A hexadecimális konstansokat 0x vagy 0X előtag különbözteti meg a többitől, amit közvetlenül a konstans hexadecimális reprezentációja követ. Az oktális konstansok 0-val kezdődnek. Használatukra két példát mutatunk:

```
int hex = 0x80; //decimális 128
int oct = 012; //decimális 10
```

Stringkonstansok

A C/C++ támogatja – a predefinit típusú konstansok mellett – a *stringkonstansokat* vagy más néven stringliterálokat is. A *string* (karakterlánc) vagy *stringkonstans* egy idéző jelek közé zárt karaktersorozat. Például az „Ez egy példa" egy érvényes string. Ne keverjük össze a karaktereket a stringekkel. A karakterkonstanst macskakörmök közé írjuk, mint pl. az 'a'-t. Az „a" literál viszont egy egyetlen karakterből álló stringet jelent. A stringkonstansokat a fordító nulla-végűekként tárolja, azaz a memóriában a string végét egy 0 tartalmú byte jelzi. C++ rendelkezik egy **string** nevű osztállyal, amelyről bővebben a könyv egy későbbi fejezetében lesz szó.

Logikai konstansok

C++ lehetővé teszi két Boolean konstans használatát is a **true**-ét (igaz) és a **false**-ét (hamis).

Vezérlőkarakterek

Kód	*Jelentés*
\b	Törlés (backspace)
\f	Lapdobás
\n	Újsor
\r	Kocsivissza
\t	Vízszintes tabulátor
\"	Idézőjel
\'	Macskaköröm
\\	Backslash
\v	Függőleges tabulátor
\a	Figyelmeztetés
\N	Oktális karakterkód (ahol N oktális konstans)
\xN	Hexadecimális karakterkód (ahol N hexadecimális konstans)
\?	Kérdőjel

A legtöbb karakter közvetlenül a billentyűzet által bevihető a forráskódba. Vannak azonban speciális karakterek, mint például a kocsivissza, amelyeket nincs módunk beírni a programkódba. Ezeket vezérlő karaktereknek vagy escape szekvenciáknak nevezzük, szükségünk lehet rájuk string- és karakterkonstansokban. A C és a C++ nyelvekben ezen speciális karakterkonstansokra a fordított törtvonal (backslash) segítségével utalhatunk (l. táblázat az előző oldalon).

A vezérlőkaraktereket leíró konstansok bárhol használhatók, ahol karaktereket lehet használni. Az alábbi utasítás egy új sort kezd, majd egy tabulátor beiktatása után kiírja az „Ez egy próba" stringet:

```
cout << "\n\tEz egy próba";
```

2. FEJEZET

Függvények, érvényességi tartományok, névterek és fejállományok

A C-ben és C++-ban a programok építőkövei a függvények. A programelemek – köztük a függvények – láthatósága több kevesebb blokkra korlátozódhat a programban. A C++-ban létezik egy speciális érvényességi tartományfajta, az ún. névtér. A függvények viselkedésére a deklarációban (a prototípus megadásakor) szereplő különböző függvényfejek utalnak. Ezen témák kifejtése képezi a fejezet tárgyát.

■ *Függvények*

A C/C++ programok szívét a függvények alkotják. Ezekben történnek a program lényegi tevékenységei. A függvények általános alakját az alábbi szintaktikai séma adja meg:

visszatérési_típus függvénynév (paraméterlista)
{
    függvénytörzs
}

A visszatérési érték típusát a *visszatérési_típus*ban közöljük. A *paraméterlista* azon változók neveinek vesszővel való elsorolása, amelyekkel az átadott paraméterekre hivatkozhatunk a függvényen belül. Az alábbi példafüggvénynek két egész típusú paramétere van: **i** és **j** és egy **double** típusú, amelyik a **count** azonosítót viseli.

```
void f(int i, int j, double count)
{ ...
```

Vegyük észre, hogy minden paramétert külön meg kell jelölnünk. C++-ban ha egy függvénynek nincsenek paraméterei, akkor a paraméterlistáját üresen hagyjuk. C-ben ennek az felel meg, ha a paraméterlista helyére a **void** szót írjuk. Például:

```
int f(void)
{ ...
```

C++-ban is használhatnánk ilyenkor a **void**-ot, de felesleges.

C-ben, ha egy függvény visszatérési értékének típusát nem tüntetjük fel, akkor a fordító automatikusan egészt feltételez. A standard C++-ban ez az elv nem kötelező jellegű, ennek ellenére a legtöbb C++ fordítóban átvették. Függvényeken belül a végrehajtás végét az utolsó zárójel jelenti, ekkor a hívó eljárás visszakapja a vezérlést. Ennél korábbi visszatérést a **return** utasítással kényszeríthetünk ki.

Minden függvénynek, kivéve a **void**-ként deklaráltaknak, van visszatérési értéke. A visszaadott érték típusának meg kell felelnie a deklarációban megjelölt típusnak. Egy nem **void**-ként deklarált függvény visszatérési értékként a **return** utasítás mögött álló kifejezés értékét kapja meg.

C++-ban lehetőségünk van ún. generic függvények megadására a **template** kulcsszó segítségével. (l. a Kulcsszavak összefoglalása c. fejezet „**template**" címszavát.)

■ *Rekurzió*

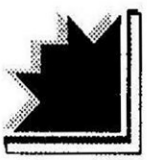
C-ben és C++-ban megengedett, hogy a függvények saját magukat hívják. Ezt a tevékenységet *rekurziónak* nevezzük, az ezt megvalósító függvényre a *rekurzív* jelzőt használjuk. Például tekintsük az alábbi **factr( )** függvényt, amely egy egész szám faktoriálisát számolja ki. Egy N egész faktoriálisán az 1-től N-ig terjedő egész számok szorzatát értjük (jel: N!). Például 3! = 1×2×3 = 6.

```
// egy szám faktoriálisát rekurzióval számoljuk ki
int factr (int n)
{
  int answer;
```

```
    if(n==1) return 1;
    answer = factr(n-1)*n;
    return answer;
}
```

Amikor a **factr( )**-nak 1-et adunk át paraméterként, akkor a függvény 1-et ad vissza; minden más esetben a **factr(n–1)*n** lesz a visszatérési érték, ennek a kifejezésnek a kiértékelése csak a **factr(n–1)** függvényhívás után lehetséges, ami pedig **factr( )** ismételt meghívását jelenti egy eggyel kisebb értékkel. Könnyen látható, hogy ez az önhívási folyamat egészen addig tart, amíg az argumentum értéke 1 nem lesz, ekkor visszafelé egyesével értékelődhetnek ki az egymást követő hívások eredményei. Amikor a **factr( )** végül visszatér az első hívás helyére, visszatérési értéke az argumentum faktoriálisát fogja tartalmazni.

Minden alkalommal, amikor egy függvény meghívja önmagát, újabb tárterület lefoglalása szükséges a lokális változók és a paraméterek számára (a tárolás adatszerkezetileg egy veremben történik), majd a függvénykód ezen új változók segítségével újra végrehajtódik. Egy rekurzív hívás nem készít újabb változatot a függvényről. Kizárólag az argumentumok cserélődnek. Ahogy az egyes rekurzív hívások befejeződnek, a régi lokális változók és paraméterek visszamásolódnak a veremből, és a végrehajtás a függvény belsejében lévő rekurzív hívás után folytatódik. A rekurzív függvények viselkedése egy teleszkóp kihúzásához és betolásához hasonlítható.



Programozási tipp

Nem árt csínján bánni a rekurzív függvények használatával. Sok alprogram rekurzív változata lassabban fut, mint ezekkel ekvivalens iteratív változatok, az ismételt függvényhívás fokozott tárigénye miatt. Sok egymás utáni rekurzív lépés a verem túlcsordulásához vezethet. Mivel a függvény lokális változói és paraméterei a veremben tárolódnak és minden új hívás ezek újabb példányainak tárolását jelenti, a verem betelhet. Ekkor kapjuk a stack overflow *(veremtúlcsordulás) üzenetet. Ha egy működő, hibáktól mentes függvény esetén ez történne, próbáljuk meg növelni a verem számára lefoglalt helyet.*

Rekurzív függvények írásakor mindig ügyeljünk arra, hogy a függvény tartalmazzon valahol egy olyan feltételt, amely újabb önhívás nélkül tér vissza. Ellenkező esetben ugyanis a rekurzió egészen a verem beteltéig fog

tartani. Ez gyakori hiba a rekurzív függvények készítésekor. Célszerű a képernyőn nyomon követni (a kiviteli könyvtár használatával) a rekurzió menetét fejlesztés közben, és félbeszakítani a futást, ha valamilyen hibát észlelünk.

■ *Függvénynevek túlterhelése*

C++-ban a függvényneveket *túlterhelhetjük.* Túlterhelésről akkor beszélünk, amikor kettő vagy több függvény ugyanazt a nevet viseli. A „névrokonok" azonban különböző számú vagy más típusú paraméterekkel kell hogy rendelkezzenek. (A visszatérési értékek is különbözhetnek, de ez nem kötelező.) Amikor egy túlterhelt függvényt hívunk, a fordító el tudja dönteni – a típus és az aritás (paraméterszám) alapján –, hogy melyik verziónak szól valójában a hívás. Példaként tekintsük az alábbi három függvényt, amelyeket egy közös név alatt deklaráltunk.

```
void myfunc(int a) {
    cout << "a is " << a << endl;
}
// myfunc túlterhelése
void myfunc(int a, int b) {
    cout << "a is " << a << endl;
    cout << "b is " << b << endl;
}
// további myfunc túlterhelés
void myfunc(int a, double b) {
    cout << "a is " << a << endl;
    cout << "b is " << b << endl;
}
```

Az alábbi hívások értelmesek:

```
myfunc(10); // a myfunc(int)-et hívja
myfunc(12, 24); // myfunc(int, int)-et hívja
myfunc(99, 123.23); // myfunc(int, double)-et hívja
```

Mindhárom esetben az argumentumok típusa és száma dönti el, melyik változat hívódik meg.

A függvénynevek túlterhelését a C nem támogatja.

■ *Argumentumok alapértelmezett (default) értékei*

A C++ lehetőséget biztosít arra, hogy a függvényhívások bizonyos paraméterek megadása nélkül is értelmesek maradjanak. Ekkor a programozónak kell gondoskodni arról, hogy ezen argumentumok ún. alapértelmezett értékeket kapjanak. Ez szintaktikailag hasonlóan történik a változók inicializálásához. Az alábbi példában szereplő függvény mindkét paramétere default értéket kap:

```
void myfunc(int a = 0, int b = 10)
{ // ...
```

Alapértelmezett változók használatával a következő három hívásmód mindegyike értelmes:

```
myfunc(~); // a automatikusan 0-át, b 10-et vesz fel
myfunc(-1); // a értéke a hívás miatt -1, b-be bemáso-
            //lódik az alapértelmezésként megadott 10
myfunc(-1,99); // a-nak -1 adódik át, b-nek 99
```

Az alapértelmezett argumentumértékeket egyszer kell megadnunk, vagy a függvény prototípusában, vagy magában a definícióban. Az előbbi módszer közkedveltebb. Alapértelmezett argumentumértékeket tartalmazó függvényekben mindig azokat az argumentumokat kell feltüntetnünk elsőként, amelyeknek nincs alapértéke, azaz mindenképpen a híváskor adódnak át. A default értékekkel felruházott argumentumok közé nem ágyazhatunk más paramétereket.

A C nem támogatja az argumentumok alapértelmezését.

■ *Prototípusok*

C++-ban minden függvénynek fel kell tüntetni a prototípusát. C-ben a prototípus használata nem kötelező, azonban melegen ajánlott. A prototípusok megadásának általános alakja:

visszatérési_típus függvénynév (paraméterlista);

A prototípus lényegében nem más, mint a függvény definíciójából a visszatérési érték típusának, a névnek és a paraméterlistának a közlése, amelyet egy pontosvessző zár le.

A következő példában az **fn( )** függvény prototípusát és definícióját láthatjuk.

```
float fn(float x); // prototípus
  .
  .
  .

// függvénydefiníció
float fn(float x)
{
 // ...
}
```

Olyan függvények prototípusának megadásakor, amelyeknek változó számú argumentumot adhatunk át, a paraméterlistában használjunk „..."-ot (felsorolásjelet) ott, ahol a változó számú paraméterek következnek. Például a **printf( )** függvény prototípusa így nézne ki:

```
int printf(const char *format, ...);
```

Egy túlterhelt függvénynévhez tartozó összes függvényverzió prototípusát szerepeltetnünk kell. Az osztályok függvénytagjainak deklarálása egyben a prototípus megadását is jelenti.

C-ben egy paraméter nélküli függvény prototípusában is szükséges a **void** kulcsszó használata a paraméterlista helyén.

Programozási tipp

Két alapfogalmat gyakran kevernek a C/C++ programozásban: a deklarációt *és a* definíciót. *Valójában ezek a következőket jelentik.* A deklarációval *megadjuk az adott objektum típusát és nevét.* A definícióval *helyet foglalunk számára a memóriában. Ezek a meghatározások függvényekre is állnak. Egy függvény deklarációja (prototípusa) a függvény visszatérési érték típusát, nevét és paramétereit rögzíti. Maga a függvény (azaz amelyik a törzset is tartalmazza) a definíció.*

Sok esetben a deklaráció egyben definíció is. Ilyen például egy nem **extern** *változó deklarálása. Egy olyan függvénydefiníció, amely még első használata előtt történik, deklarációként is szolgál.*

■ *Változók érvényességi tartománya és élettartama*

A C-ben és C++-ban jól meghatározott szabályok írják le a különböző programkontextusban deklarált változók hatáskörét és élettartamát. A számos finomság ellenére általánosságban mondhatjuk, hogy kétféle hatáskör különböztethető meg élesen egymástól: *globális és lokális.*

A globális láthatóság minden láthatósági kört tartalmaz. Egy globális hatáskörű névre a program egészében hivatkozhatunk. Például egy globális változót a program összes függvénye használhat. Élettartamuk a program futásának idejéig terjed.

A lokális hatáskör egy blokkon belüli láthatóságot jelent. Azaz egy nyitó utasítászáró-jellel „{" kezdődik és annak a párjánál ér véget. Ha egy utasításblokkban lokális változót deklarálunk, érvényességi tartománya arra az utasításblokkra korlátozódik. Az utasításblokkokhoz hasonlóan az érvényességi tartományok is egymásba ágyazhatók. A leggyakoribb példát a lokális érvényességi tartományra természetes módon a függvényeken belül deklarált változók szolgáltatják. Lokális változókat a program akkor hozza létre, amikor a végrehajtás a változót tartalmazó blokkba kerül. A számukra lefoglalt memória felszabadítása a blokk végrehajtása után történik. Ez azt jelenti, hogy a lokális változók nem őrzik meg értékeiket a függvényhívások között. A **static** módosító használatával a változó élettartamát kiterjeszthetjük a program futásának idejére.

C++-ban egy utasításblokk tetszőleges részében deklarálhatunk lokális változókat. C-ben ezt a blokk elején kell megtennünk, a blokk lényegi (az aktív utasításokat tartalmazó) része előtt. Például az alábbi kód értelmes C++-ban, de hibás C-ben:

```
void f(int a)
{
  int a;
  a=10;
  int b; //jó C++-ban, hibás C-ben
```

Globális változókat az összes függvényen (beleértve a **main( )** függvényt is) kívül kell deklarálni. Globális változóinkat általában a forrásfájl tetejére helyezzük a **main( )**-t megelőzően, a jó átláthatóság kedvéért és mert a változókat használatuk előtt deklarálnunk kell.

A függvények formális paraméterei szintén lokális változók, amelyek a főfunkciójukon, azaz az átadott értékek tárolásán kívül úgy is használhatók, mint bármilyen más lokális változó.

■ *Névterek*

C++-ban a **namesspace** kulcsszó segítségével megoldhatóvá válik, hogy olyan tartományokat definiáljunk változóinkhoz, amelyekre azok közvetlen hozzáférhetősége korlátozható. Célja a nevek lokalizálása. A *névterek* segítségével tehát deklaratív tartományokat hozhatunk létre. Általános szintaxisuk:

```
namespace név{
    // ...
}
```

Ahol a *név* a névtér neve. Példa:

```
namespace MyNameSpace {
    int count;
}
```

Ez egy **MyNameSpace** nevű névteret definiál, melynek belsejében egy **count** nevű változót deklaráltunk. A névtéren belül más utasítások közvetlenül hivatkozhatnak az ott deklarált nevekre. A névtéren kívül kétféleképpen érhetjük el a belső deklarációk azonosítóit. Egyrészt használhatjuk a hatáskörfelbontó operátort (::). Feltételezve, hogy a **MyNameSpace** éppen látható, használható például az alábbi utasítássor:

```
MyNameSpace::count = 10;
```

A **using** utasítás használatával a megadott névtér-azonosítókat az aktuális blokkban láthatóvá teszi. Az alábbi példa ezt szemlélteti:

```
using namespace MyNameSpace;
count = 100;
```

Ebben az esetben a **count**-ra közvetlenül hivatkozhatunk, mert láthatóvá tettük a **MyNameSpace**-ben szereplő változókat.

Hagyományosan a C++-könyvtárban deklarált objektumok a globális (név nélküli) névtérben szerepeltek. A jelenlegi specifikáció, azonban az **std** névtérbe helyezi őket.

■ *A* **main( )** *függvény*

C/C++ programokban a végrehajtás a **main( )** függvénnyel kezdődik. (Windowsos programok a **WinMain( )**-t hívják, de ez egy speciális eset.) Csak egyetlen **main( )** függvényünk lehet a programban. A **main( )** véget érése egyben a program futásának végét és az irányítás visszatérését az operációs rendszerhez is jelenti.

A **main( )** függvényből nem hozunk létre prototípust. Így többféle **main( )** is használható. C/C++-ban például az alábbi **main( )** változatok érvényesek. (Más formák is megengedettek.)

```
int main()
int main(int argc, char *argv[])
```

A második sorban látható, hogy legalább két paramétert támogat a **main( )**: az **argc**-t és **argv**-t. (Számos fordító más paraméterek használatát is megengedi.) Ez a két változó hordozza a parancssor-argumentumok számát, illetve a rájuk mutató pointert. Az **argc** egy egész típusú paraméter, melynek értéke legalább egy, mert első argumentumként mindenképpen a programneve jelenik meg. Az **argv** paramétert karaktermutatókból álló tömbként definiáljuk. Mindegyik mutató egy-egy parancssor-argumentumra mutat. Használatukat az alábbi rövid programmal mutatjuk be, amely a megadott nevet írja ki a képernyőre.

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

int main(int argc, char *argv[])
{
  if(argc<2)
    cout << "Írja be a nevét.\n";
  else
    cout << "Szia " << argv[1];

  return 0;
}
```

■ *Függvényargumentumok*

Ha függvényeinknek paramétereket szeretnénk átadni, akkor deklarálnunk kell azok értékét elfogadó és hordozó változókat. Ezeket a változókat a függvény *formális paramétereinek* nevezzük. Ezek úgy viselkednek, mint a függvényen belüli többi lokális változó. Létrehozásuk a függvénybe való belépéskor, megszüntetésük pedig kilépéskor történik. A lokális változókhoz hasonlóan szerepelhetnek értékadás bal oldalán, és használhatjuk őket bármilyen érvényes C/C++-kifejezésben. Habár ezek fő feladata az átadott értékek tárolása, gyakran használjuk őket a szokásos lokális változó szerepben.

Általában a szubrutinoknak kétféleképpen adhatunk át argumentumokat. Az egyik az ún. *érték szerinti paraméterátadás*. Ennek a módszernek a lényege

abban áll, hogy az argumentumok értékei átmásolódnak az alprogram megfelelő formális paraméterébe. A paramétereken végzett semmilyen változtatás sem befolyásolja a szubrutin hívásakor használt változók értékét.

A másik módszert, amellyel az argumentumokat eljuttathatjuk az alprogramhoz, *cím szerinti paraméterátadásnak* nevezzük. Az ilyen hívásoknál az argumentum címe adódik át paraméterként. A szubrutinon belül ezt a címet használja a program az aktuális argumentum kiolvasására és manipulálására. Ez azt jelenti, hogy minden, a paraméteren végzett változtatás egyben a híváskor használt változók értékét is megváltoztatja.

Alapértelmezés szerint a C és a C++ az érték szerinti paraméterátadást támogatják. Így a függvények általában nem tudják megváltoztatni a hívó változók értékét. Tekintsük az alábbi függvényt:

```
int sqr(int x)
{
  x = x*x;
  return x;
}
```

A fenti példában az **x = x * x** sor végrehajtásakor csak az **b** lokális változó értéke módosul. Az **sqr( )** hívására használt változó megőrzi az eredeti értékét.

Ne feledkezzünk meg arról, hogy valójában csak az argumentumként megadott változó másolata adódik át a függvénynek. Semminek, ami a függvényen belül történik, nincs hatása magára az eredeti változóra.

Mutatók átadása

C-ben, illetve C++-ban az alapértelmezésként használt érték szerinti paraméterátadás mellett kis trükkel, manuálisan a cím szerinti paraméterátadás is megvalósítható, ekkor egy, az argumentumra mutató pointert adunk át. Ezzel tehát az argumentum címe jut el a függvényhez, amelyen keresztül lehetőség nyílik a híváskor használt argumentum értékének a megváltoztatására.

Pointereket úgy adunk át függvényeknek, mint bármely más értéket. Természetesen ekkor szükséges, hogy a paramétereket mutatótípusként

deklaráljuk. Tekintsük példaként az alábbi **swap( )** függvényt, amely két egész argumentumának értékét cseréli ki:

```
//Mutatóparaméterek használata
void swap(int *x, int *y)
{
  int temp;

  temp = *x; // az x címének tartalmának eltárolása
  *x = *y; // y által mutatott értéknek x címre másolása
  *y = temp; // az x által hivatkozott eredeti érték y
             //címre másolása
}
```

Semmiképpen se feledkezzünk meg arról, hogy a **swap( )** függvényt (vagy bármely más függvényt, amelyik mutatóparamétert használ) az argumentumként átadni kívánt értékek *címével* hívjuk meg. Az alábbi programrészlet a **swap( )** helyes hívását mutatja be:

```
int a, b;

a = 10;
b = 20;
swap(&a, &b);
```

A példában jól látható, hogy a **swap( )**-et az **a** és a **b** címével hívjuk meg. A & unáris (=egy paraméteres) operátor segítségével kaphatjuk meg a változók címét. Így végeredményben nem maguk a változó értékek, hanem az **a**-ra és a **b**-re mutató pointerek adódnak át a **swap( )** függvénynek. A hívás után **a** értéke 20, míg **b**-é 10 lesz.

Referenciaparaméterek

C++-ban lehetőség van a manuális út megkerülésére azaz az automatikus címátadásra. Ezt az ún. *referenciaparaméterekkel* végezhetjük el. Referenciaparaméterek használatakor az argumentum címe adódik át a függ-

vénynek és a függvény ezen argumentummal dolgozik, nem pedig annak egy másolatával.

A referenciaparaméter nevét a függvényfejben a „&" jel előzi meg. A függvény belsejében a paramétert a szokványos módon használhatjuk – nincs szükség a „*" operátorra. A fordító automatikusan a megfelelő pointermanipulációt használja majd. Az alábbi példában a **swap( )** egy referencia paraméteres változatát mutatjuk be. A két referenciaparaméter segítségével cseréljük ki az argumentumok értékét.

```
//Referenciaparaméterek használata
void swap(int &x, int &y)
{
  int temp;

temp = x; // x címén lévő érték elmentése
x = y; // y x-be másolása
y = temp; // az eredeti x y-ba kerül
}
```

A **swap( )** meghívásakor is a szokásos szintaxist használjuk. Ezt illusztrálja a példa:

```
int a, b;


a =10;
b = 20;
swap(a, b);
```

Mivel **x** és **y** most referenciaparaméterek, a fordító automatikusan képezi **a** és **b** címét és azokat adja át a függvénynek. A függvényen belül a **x**, **y** paraméterneveket a „*" operátor nélkül használjuk, mert a fordító a műveleteket automatikusan az ezek által mutatott memóriacímek tartalmával végzi, vagyis az **a** és **b** változóval.

A referenciaparamétereket csak a C++ támogatja.

■ *Konstruktorok és destruktorok*

C++-ban az osztályok tartalmazhatnak két speciális függvényt: a *konstruktort* és a *destruktort*. A konstruktor akkor hajtódik végre, amikor az osztály egy példányát létrehozzuk, míg a destruktor az osztály megszűnésekor fut le. A konstruktor azonosítója meg kell, hogy egyezzen az őt tartalmazó osztály nevével. Ugyanez érvényes a destruktorra is azzal a módosítással, hogy azonosítója elé egy „~" (tilde) jelet is be kell szúrnunk. Sem a konstruktornak, sem a destruktornak nincs visszatérési értéke.

Konstruktorfüggvényeknek lehetnek paraméterei. Ezeken a paramétereken keresztül olyan lényeges adatokat kaphat meg a konstruktor, amelyek szükségesek az objektum inicializálásához. A paraméterekbe kerülő értékek megadása az egyes objektumok létrehozásakor történik. Az alábbi példán követhetjük szemmel, hogy egy konkrét esetben hogyan adhatók át argumentumok a konstruktornak:

```
class myclass {
  int a;
public:
  myclass(int i) {a = i; } // Ez a konstruktor
  ~myclass() { cout << "Megszüntetés ...";}
};

// ...

myclass ob(3); // i-nek 3-at adunk át.
```

Amikor az **ob**-t deklaráljuk, egyúttal a konstruktor **i** paraméterének 3-at adunk értékül, ez másolódik be majd az **a** vátozótagba.

■ *Függvényspecifikátorok*

C++-ban a következő függvényspecifikátorok használhatók: **inline**, **virtual** és **explicit**. Az **inline** specifikátor azt a kérést tolmácsolja a fordító programnak, hogy a függvény kódját építse be a hívás helyére, ahelyett, hogy valóban elvégezné a hívást. Ha a fordító ezt valamilyen oknál fogva

nem teheti meg, akkor jogában áll a kérést figyelmen kívül hagyni. Mind közönséges függvények, mind pedig osztályok függvénytagjai megjelölhetők az **inline** specifikátorral.

Azok az alaposztályban deklarált függvények, amelyeket **virtual** specifikátorral látunk el, a származtatott osztályban felüldefiniálhatók. C++-ban ez jelenti az eszközt a polimorfizmus (többalakúság) megvalósítására.

Az **explicit** specifikátornak csak konstruktorokban van értelme. Az **explicit** kulcsszóval megjelölt konstruktorok használata garantálja, hogy az inicializálás csak akkor megy végbe, ha az formailag megfelel a konstruktorban megadottaknak. Automatikus konverzió ilyenkor nem történik. („Nem konvertáló konstruktort" hoz létre.)

■ *Szerkesztési specifikáció*

Gyakran előfordul, hogy egy C++ függvényt más programnyelven (pl. C-ben) írt függvénnyel szeretnénk összeszerkeszteni („összelinkelni"). Ezért a C++ megengedi, hogy megadjuk az összeszerkesztéshez szükséges információkat, amely alapján a fordító be tudja építeni a külső függvényt a kódba. Ennek általános alakja:

extern *"programnyelv" függvény-prototípus*

Látható, hogy a szerkesztési specifikációt is az **extern** kulcsszó vezeti be. A *programnyelv* azt mondja meg, hogy a beszerkesztendő függvény milyen nyelven készült. C és C++ linkelés támogatása garantált. A fordítótól függően más szerkesztések is elképzelhetőek. Egy kalap alatt több külső függvényt is deklarálhatunk az alábbi szerkesztési specifikáció mintájára:

```
extern "programnyelv" {
  függvény-prototípusok
}
```

A szerkesztési specifikáció csak a C++-ban használható, a C nem támogatja.

■ *A C és a C++ standard könyvtárai*

Sem a C-ben sem a C++-ban nincsenek olyan kulcsszavak, amelyekkel be- és kiviteli (I/O) műveleteket végezhetnénk, stringeket manipulálhatnánk. Hiányoznak bizonyos matematikai műveleteket megvalósító utasítások is, és sok egyéb kívánatos funkcióhoz sincsen megfelelő kulcsszó. Ezek megvalósítását egy sor predefinit a fordítóhoz automatikusan mellékelt könyvtári függvénykészleten keresztül oldották meg. Két alapvető könyvtárfajta létezik: a C függvénykönyvtár, amelyet mindkét nyelv fordítójához mellékelnek, és a C++ osztálykönyvtár, amely egy C++-os gyűjtemény. Utóbbiakat a könyv későbbi részében foglaljuk össze.

Ahhoz, hogy programunk valamelyik könyvtári függvényhez hozzáférjen, a megfelelő könyvtár specifikációját (szabványos fej információját) tartalmazó állománynak a nevét fel kell tüntetnünk, a C-ben ezek .h kiterjesztésűek. Az ANSI C szabványkönyvtárak fejét tartalmazó állományokat alább közöljük:

C fejállomány	*Tartalma*
assert.h	**assert( )** makró
ctype.h	Karakterkezelés
errno.h	Hibajelentés
float.h	Implementációfüggő lebegőpontos értékek
iso646.h	Makrók bizonyos operátorok helyettesítésére, úgymint **not** vagy **xor** a !, ill. a ^ helyett.
limits.h	Különböző implementációfüggő korlátok
locale.h	**setlocal( )** függvény
math.h	Különböző matematikai definíciók
setjmp.h	Nem-lokális hívások
signal.h	Szignálértékek
stdarg.h	Változó hosszú argumentumlisták
stddef.h	Gyakran használt konstansok
stdio.h	Állománykezelés
stdlib.h	Vegyes deklarációk
string.h	Sztringkezelő függvények
time.h	Rendszeridő- és dátumkezelő függvények
wctype.h	Hosszúkarakter-kezelés
wchar.h	Hosszú karakterek használatát támogató függvények

A C++ újabb specifikációjában a könyvtárak importálása (az #include direktívával) standard fejneveken keresztül történik. Ezek nem feltétlenül egyeznek meg a könyvtárak specifikációit tartalmazó állománynevekkel – így nem is végződnek .h-ra – csupán egyszerű szabványosított azonosítók, melyeket a fordítók belátásuk szerint kezelnek. Lehet, hogy az azonosító ténylegesen egy állománynevet jelöl, ez azonban nem szükségszerű. Ezeket az új stílusú C++ könyvtárazonosítókat sorolja fel az alábbi táblázat. Külön megjelöltük (STL-lel) azokat, amelyek a Standard Template Library-hoz köthetők.

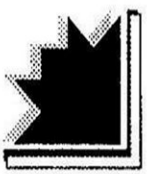
C++ könyvtárazonosító	*Tartalma*
<algorithm>	Konténerekkel kapcsolatos műveletek (STL)
<bitset>	Bithalmazok (STL)
<complex>	Komplex számok
<deque>	Kétvégű sorok (STL)
<exception>	Kivételkezelés
<fstream>	Csatorna (stream) alapú állománykezelés
<functional>	Hasznos függvények (STL)
<iomanip>	I/O manipulációk
<ios>	Alacsonyszintű I/O-kezelés
<iosfwd>	Feloldatlan I/O-deklarációk
<iostream>	Standard I/O-osztályok
<istream>	Bemeneti streamek
<iterator>	Hozzáférés konténerek tartalmához (STL)
<limits>	Különböző implementációfüggő korlátok
<list>	Lineáris listák (STL)
<locale>	Helyfüggő információk kezelése
<map>	Kulcsérték-táblázatok (STL)
<memory>	Memóriakiosztás (STL)
<new>	Memóriakiosztás a **new** használatával
<numeric>	Általános célú numerikus műveletek
<ostream>	Kimeneti streamek
<queue>	Sorok (STL)
<set>	Halmazok (STL)
<sstream>	String streamek
<stack>	Vermek (STL)
<stdexcept>	Szabvány kivételek
<streambuf>	Pufferelt streamek
<string>	Szabvány **string**-osztály (STL)
<typeinfo>	Futásidejű típusinformáció
<utility>	Általános célú sablonok (template-ek) (STL)
<valarray>	Különböző értékekből alkotott tömb kezelése
<vector>	Vektorok (dinamikus tömbök) (STL)

C++ továbbá az alábbi C könyvtáraknak megfelelő új stílusú könyvtárazonosítókat is bevezette:

<cassert>	<cctype>	<cerrno>
<cfloat>	<ciso646>	<climits>
<clocale>	<cmath>	<csetjmp>
<csignal>	<cstdarg>	<cstddef>
<cstdio>	<cstdlib>	<cstring>
<ctime>	<cwchar>	<cwctype>

A szabvány C++-ban minden, a standard könyvtárban definiált objektum az **std** névtér alatt található. Így, ha ezekhez közvetlenül kívánunk hozzáférni, szükséges a **using** utasítás használata az alábbi módon:

```
using namespace std;
```



Programozási tipp

Ha régebbi C++ fordítót használunk, akkor előfordulhat, hogy az nem támogatja az új típusú C++ könyvtár-azonosítókat vagy a ***namespace*** *utasítást. Ebben az esetben a hagyományos jelölésmódot kell használnunk, azaz olyan neveket, amelyek .h-ra végződnek (a C stílusú könyvtárnevekhez hasonlóan). Az alábbi sor az* ***<iostream>*** *importálására példa a hagyományos stílusban:*

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

A régebbi típusú könyvtárfejeket használva, minden könyvtári szolgáltatás a globális névtér alá helyeződik és nem az ***std*** *által definiáltba. Ezért a* ***using*** *utasítás felesleges.*

3. FEJEZET

Műveletek

C és C++ nyelvek bőséges műveletkínálattal rendelkeznek, amelyeket az alábbi csoportokba oszthatunk: aritmetikai, relációs, logikai, bitenkénti, mutatókon végzett, értékadás, I/O (be- és kimeneti) és az egyéb.

■ *Aritmetikai műveletek*

A C-ben és a C++-ban az alábbi hét aritmetikai művelet áll rendelkezésünkre:

Művelet	*Hatása*
–	Kivonás, negatívszám-képzés
+	Összeadás
*	Szorzás
/	Osztás
%	Modulusképzés
– –	Dekrementálás
++	Inkrementálás

A +, –, * és / műveletek a vártnak megfelelően működnek. A % operátor az adott egésszel való osztás maradékát adja meg. Az inkrementáló és dekrementáló operátorok az operandust rendre eggyel növelik, illetve csökkentik.

A fenti operátorok precedencia (végrehajtási) sorrendjét táblázatban adjuk meg:

Precedencia	*Operátor*
Legmagasabb	++ -- - (unáris mínusz)
	* / %
Legalacsonyabb	+ -

Az azonos precedenciaszinten található műveletek balról jobbra értékelődnek ki.

■ *Relációs és logikai műveletek*

Relációs és logikai operátorok igaz vagy hamis értéket adnak eredményül, gyakran a kettő kombinációját használjuk. C/C++-ban bármilyen nullától különböző számnak igaz logikai értéke van, a nullához hamis rendelődik. C++-ban relációs és logikai műveletek eredményei **bool** típusúak. C-ben az eredmény nulla vagy nem nulla egész szám. A relációs operátorokat az alábbi táblázat foglalja össze:

Operátor	*Funkció*
>	Nagyobb, mint
>=	Nagyobb vagy egyenlő, mint
<	Kisebb, mint
<=	Kisebb vagy egyenlő, mint
==	Egyenlő
!=	Nem egyenlő

A logikai operátorok:

Operátor	*Funkció*
&&	logikai ÉS (konjunkció)
||	logikai VAGY (diszjunkció)
!	logikai NEM (negáció)

Relációs operátor két értéket hasonlít össze. Logikai műveletek logikai értéket kötnek össze vagy a ! esetén negálja (ellenkezőjére váltja) a megadott logikai kifejezés értékét. Ezek precedenciája az alábbi hierarchiába rendezhetők:

Precedencia	*Művelet*
Legmagasabb	!
	> >= < <=
	== !=
	&&
Legalacsonyabb	||

A következő kódrészletben az **if** állítás mögötti rész kiértékelődik és a képernyőn az **x kisebb, mint 10** üzenet jelenik meg.

```
x = 9;
if(x < 10) cout << "x kisebb, mint 10";
```

A második példában nem jelenik meg üzenet, mert a && operátorral összekötött két kifejezés mindegyikének igaznak kellene lennie egy igaz eredményhez.

```
x = 9;
y = 9;
if(x < 10 && y > 10)
    cout << "Nem írok ki semmit.";
```

■ *Bitszintű műveletek*

A C és a C++ olyan műveleteket is biztosít számunkra, amelyekkel változók értékét reprezentáló bitsorozatokat manipulálhatunk. Bitszintű műveletek csak egész típusú változókon használhatók. Alább ezeket soroljuk fel:

Művelet	*Funkció*
&	bitenkénti ÉS
|	bitenkénti VAGY
^	bitenkénti KIZÁRÓ VAGY
~	bitenkénti NEGÁCIÓ (komplementer)
>>	jobbra léptetés
<<	balra léptetés

Az &, a | és a ^ operátorok

Az &, a | és a ^ operátorok igazságtáblázata:

p	q	p&q	p|q	p^q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Ezek a szabályok bitenként alkalmazandók az operandusokra, amikor bitszintű ÉS, VAGY, illetve KIZÁRÓ VAGY műveleteket végzünk.

A bitszintű ÉS működése jól látható az alábbi elrendezésből:

```
  0 1 0 0 1 1 0 1
& 0 0 1 1 1 0 1 1
  ---------------
  0 0 0 0 1 0 0 1
```

A bitszintű VAGY esetén ez így néz ki:

```
  0 1 0 0 1 1 0 1
| 0 0 1 1 1 0 1 1
  ---------------
  0 1 1 1 1 1 1 1
```

Végül a KIZÁRÓ VAGY:

```
  0 1 0 0 1 1 0 1
^ 0 0 1 1 1 0 1 1
  ---------------
  0 1 1 1 0 1 1 0
```

A komplementer operátor

A komplementer operátor (~) az operandusának minden bitjét átbillenti (negálja). Például, ha a **ch** karaktervváltozót a

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

bitsorozat reprezentálja, akkor a

```
ch = ~ ch;
```

A ch bitmintáját így alakítja:

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

A léptető operátorok

A jobbra, illetve a balra léptető operátorok (>> és <<) egy egész változó bitjeit a megadott értékkel tolja jobbra, illetve balra. Amíg az egyik oldalon a bitek az eltolás következtében a bitek kilépnek, addig a másik oldalon 0-k jelennek meg. (Ha a léptetendő érték negatív egész, és jobbra léptetést végzünk, akkor a bal oldalon egyesek jönnek be, így az előjel megőrződik.) A léptető operátor után álló szám adja meg, hogy hány pozíciót kívánunk léptetni. A léptető operátor általános alakja:

érték >> *szám*
érték << *szám*

A *szám* jelöli a léptetendő pozíciók számát. Tekintsük a következő bitmintát (és feltételezzünk egy előjel nélküli értéket).

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

Jobbra léptetés után a

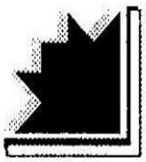
```
0 0 0 1 1 1 1 0
```

bitsorozathoz jutunk.

Balra léptetéssel a

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

mintát kapjuk.



Programozási tipp

A jobbra léptetés a kettővel való osztásnak felel meg, a balra léptetés kettővel szoroz. Sok gép esetén a léptetés gyorsabb, mint a szorzás vagy osztás. Ezért kettővel való szorzáskor, illetve osztáskor nem árt fontolóra venni a léptető műveletek használatát. Az alább közölt kódrészlet először 2-vel szorozza, majd kettővel osztja az **x** *értékét.*

```
int x;

x = 10;
x = x << 1;
x = x >> 1;
```

Természetesen nem szabad, hogy elfeledkezzünk a szélső bitek értékéről, hiszen ezek elvesznek az eltolás során. Vigyázat, ha szorzásnál a felső bit értéke egy, akkor eredményül kisebb számot kapunk.

A bitszintű műveletek precedenciáját az alábbi táblázatból olvashatjuk ki:

Precedencia	*Operátor*
Legmagasabb	~
	>> <<
	&
	^
Legalacsonyabb	|

■ *Mutatók használata*

Két művelet szükséges a mutatók manipulálásához: a * és a &. A *mutató* egy olyan típus, amely egy másik objektummemória címét tartalmazza. A változóra, amely egy másik objektum címét tartalmazza, azt mondjuk, hogy a másik objektumra „mutat".

Az & operátor

Az & operátor az utána álló objektum címét adja vissza. Például az 1000-es memóriacímen elhelyezkedő **x** egészváltozó esetén a

```
p = &x;
```

sor végrehajtásakor a **p** változóba 1000 kerül. A & operátor jelentését legtömörebben a „a ... címe" séma fejezi ki. A fenti állítás úgy olvasható, hogy „helyezd az **x** címét **p**-be".

*A * operátor*

A * az úgy nevezett *indirekció operátor.* Az utána álló változó aktuális értékének megfelelő címet használja, tartalmának kiolvasására, illetve adatok eltárolására. Az alábbi programrészlet végrehajtásával az **x** változóba 100 kerül:

```
p = &x; /* x címe p-be kerül */
*p = 100; /* p-ben tárolt cím tartalmát adjuk meg. */
```

A * operátor hatását röviden „a ... címére" kifejezéssel írhatjuk le. Ebben a példában a második sort így olvashatjuk: „Helyezd a 100-at, mint értéket a **p** címére." Mivel a **p**-ben az **x** címe van, a 100 tulajdonképpen **x**-ben lesz eltárolva. Más szóval, azt használjuk ki, hogy a **p** **x**-re mutat. A * operátor értékadás jobb oldalán is szerepelhet. Például:

```
p = &x;
*p = 100;
z = *p/10;
```

Az értékadások után **z**-be 10 kerül.

■ *Értékadás operátorok*

A C-ben és a C++-ban az értékadás operátornak az egyszerű egyenlőségjel felel meg. Ha több változónak szeretnénk ugyanazt az értéket adni, akkor az értékadások összefűzhetők. Például a

```
a = b = c = 10;
```

utasítássor az **a**, **b**, **c** változókba 10-et másol.

A C/C++ lehetővé teszi egy kényelmes, rövidített forma használatát az alábbi típusú értékadások esetén:

változó = változó op. kifejezés;

Ezek tehát ilyen alakba írhatók át:

változó op. = kifejezés;

Például az a két értékadás, hogy

```
x = x+10;
y = y/z;
```

helyettesíthető úgy, hogy

```
x += 10;
y /= z;
```

■ *A ? operátor*

A ? egy három operandusú operátor. Általános alakban:

kifejezés1 ? kifejezés2 : kifejezés3;

Ha *kifejezés1* igaz, akkor a művelet végeredménye a *kifejezés2*, különben a *kifejezés3*.

Programozási tipp
A ? operátort gyakran az alábbi típusú ***if-else*** *utasítások helyettesítésére használjuk:*

if (*kifejezés1*) *változó* = *kifejezés2;*
else *változó* = *kifejezés3;*

A következő két utasítássor

```
if (y < 10) x = 20;
else x = 40;
```

így írható:

```
x = ( y < 10) ? 20 : 40;
```

Itt, ha y kisebb, mint 10, akkor x értéke 20 lesz, egyébként pedig 40. A ? operátor létjogosultsága mellett – a gépelés rövidítésén kívül – az a tény szól, hogy a fordító rendkívül gyorsan futó kódot tud erre az utasításra generálni – sokkal gyorsabbat, mint az ezt helyettesítő ***if-else*** *utasításra.*

■ *Tagoperátorok*

A . (pont) és a -> (nyíl) operátoroknak a feladata, hogy egy osztály, struktúra vagy union adott tagjára hivatkozzon. A pont operátort a tulajdonképpeni objektumra használjuk, míg a nyíl operátort az objektumra mutató pointerre alkalmazzuk. Tekintsük az alábbi struktúrát.

```
struct datum_ido {
    char datum[16];
    int ido;
} tm;
```

Ha „99/12/3" stringet kívánjuk **tm** objektum **datum** tagjának értékül adni, akkor így járnánk el:

```
strcpy(tm.date,"99/12/3");
```

Tegyük fel, hogy egy **p_tm** mutatónk van, amely a **date_time** osztály egy példányára mutat. Ekkor az előző feladatot így valósítjuk meg:

```
strcpy(p_tm->date,"99/12/3");
```

■ *A vessző operátor*

A vessző operátort egy műveletsorozat végrehajtásakor alkalmazzuk. A vesszőt tartalmazó kifejezés értéke az utolsó kifejezés (jobb oldali) a vesszővel elválasztott listában. Például a

```
y = 15;
x = (y = y-5, 50/y);
```

programrészlet végrehajtása után az **x** változó értéke 5 lesz, ugyanis **y** eredeti értéke (15) először 5-tel csökken, majd az így kapott értékkel (10) osztjuk el az 50-et, ami az 5 végeredményt adja. A vessző operátor hatását szövegesen így fogalmazhatnánk meg: „Csináld ezt, majd azt".

A vessző operátor leggyakrabban a **for** utasításban fordul elő. Például:

```
for(z=10, b=20; z<b; z++, b- -) { // ...
```

Itt a **z** és a **b** változók inicializálása és módosítása a vessző operátor segítségével történik.

■ *A sizeof operátor*

A **sizeof** kulcsszó minősége ellenére egyben fordítási idejű operátor is. Feladata az operandusként megadott változó vagy adattípus (beleértve az osztályokat, struktúrákat és unionokat is) méretének meghatározása. Ha típusra alkalmazzuk, akkor a típusazonosítót zárójelek közé kell tennünk.

A legtöbb 32-bites fordítóval lefordítva, az alábbi kód 4-et ír ki.

```
int x;
cout << sizeof x;
```

■ *Típusmódosító (cast) operátor*

A típusmódosító egy speciális operátor, amelynek segítségével típuskonverziót kényszeríthetünk ki, azaz egyik adattípusról áttérhetünk egy másikra. A konverziót mindkét nyelvben az alábbi általános minta szerint kell végezni:

(*típus*) *kifejezés*

ahol a *típus* a cél adattípust jelenti (amelyikbe konvertálni kívánunk).

A következő konverzió az egész osztás eredményét alakítja **double** típusúvá.

```
double d;
d = (double) 10/3;
```

C++ konverziók

A C++ az előbb említetten kívül más konverziós operátorokat is támogat. Ezek a **const_cast, dynamic_cast, reinterpret_cast** és a **static_cast** operátorok. Általános alakjukban:

const_cast <*típus*> (*objektum*)
dynamic_cast <*típus*> (*objektum*)
reinterpret_cast <*típus*> (*objektum*)
static_cast <*típus*> (*objektum*)

ahol a *típus* a konverzió cél adattípusát, míg az *objektum* a konvertálandó objektumot jelenti.

A **const_cast** operátort akkor használjuk, ha fel akarjuk oldani a **const** és/vagy a **volatile** (l. korábban) módosító hatását egy adott változóra. A céltípusnak ebben az esetben meg kell egyeznie a kiindulási típussal, változás kizárólag annak **const** vagy **volatile** attribútumában következik be. Leggyakrabban a „konstansság" megszüntetésére használják.

A **dynamic_cast**-tal futásidejű konverziókat végeztethetünk, amely egyben ellenőrzi a cast helyességét. Ha egy adott típusmódosítás nem

végrehajtható, akkor a kifejezés null értékűvé válik. Fő felhasználási területét a polimorf típusok konverziója adja. Példaként tekintsünk két polimorf osztályt, nevezzük ezeket B-nek és D-nek, ahol D a B-nek egy leszármazottja. A **dynamic_cast**-tal történő konverzió mindig kivitelezhető, ha D* mutatót alakítunk B*-gá. Visszafelé csak akkor működik a dolog, ha a B* típusú pointer címén ténylegesen egy D típusú objektum található. Általánosan fogalmazva, egy **dynamic_cast** sikeressége azon múlik, hogy a megkísérelt polimorfikus konverzió engedélyezett-e (vagyis hogy a céltípus legálisan alkalmazható-e a konvertálandó objektumra). Ha a cast nem végezhető el, akkor eredményül nullpointert kapunk.

A **static_cast** operátorral nempolimorfikus konverziókat végezhetünk. Például általa megtehetjük, hogy egy alaposztály típusú mutatót (az abból) leszármaztatott osztály típusú mutatóvá alakítsunk. Bármely standard konverzió is végrehajtható vele. Nincsen futásidejű ellenőrzés. A **reinterpret_cast** operátorral megtehető, hogy egy változót annak típusától merőben különböző típusúvá alakítsunk. Például elvégezhető egy mutató egésszé történő konvertálása. A **reinterpret_cast** használandó két inkompatíbilis mutató típus közti konverzió megvalósítására.

Csak a **const_cast** képes a **const** módosító hatását megszüntetni, a **dynamic_cast**, **reinterpret_cast** és a **static_cast** közül semelyik sem rendelkezik ezzel a tulajdonsággal.

■ *I/O operátorok*

C++-ban a << és a >> túlterhelt operátorok, ezek felelősek bizonyos be- és kimeneti (I/O) műveletekért is. Ha olyan formában használjuk, hogy bal operandusként valamilyen csatornát (adatfolyamot) adunk meg, akkor a >> bemeneti (input) operátor szerepét tölti be, míg ugyanezen feltételek mellett a << egy kimeneti operátor. A C++-ban a >> operátort *extraktornak* hívják, mert a bemeneti csatornából adatokat olvas be. A <<-t, mivel adatokat helyez a kimeneti csatornába, *inzerternek* is nevezik. Általános alakban így néznek ki:

bemeneti-csatorna >> *változó*
kimeneti-csatorna << *kifejezés*

Az alábbi példa két egész változó bekérését mutatja be.

```
int i, j;
cin >> i >>j;
```

A következő utasítás azt írja a képernyőre, hogy „Tesztelés 10 20":

```
cout << "Tesztelés" << 10 << ' ' << 4*5;
```

Az I/O operátorokat a C nem támogatja.

■ *Tagra hivatkozó mutatók, a .* és a ->* operátorok*

C++ támogatja olyan speciális mutatók használatát, amelyek egy adott osztály valamelyik tagjára mutatnak. Ezeket a mutatókat *tagra hivatkozó mutatóknak* nevezik. Egy *tagra hivatkozó mutató* különbözik a megszokott mutatóktól annyiban, hogy a tagra mutató pointerek csak egy eltolás (offset) részt tartalmaznak, ami azt mondja meg, hogy a osztály bármely példányának kezdőcíméhez képest mennyivel arrébb kezdődik az adott tag memóriacíme (relatív cím). Mivel a tagra hivatkozó mutatók nem szokványos mutatók, nem alkalmazhatók rájuk a . és -> operátorok. Egy osztálybeli objektum adott tagjának elérésére – ha már elkészítettük a megfelelő tagra hivatkozó mutatót – speciális operátorok szolgálnak: a .* és a ->* .

Az objektum egy tagjának az elérése – ha az objektum adott – a .* operátor segítségével történik. Ha az objektumra hivatkozó mutató adott, akkor a megfelelő tag elérésére a ->* operátor szolgál.

Egy tagra hivatkozó mutató deklarációja általánosan így néz ki:

*típus osztálynév::*ptr;*

ahol a *típus* a tag alaptípusa, *osztálynév* az osztály neve és a *ptr* a deklarálandó tagra mutató pointer változó azonosítója. A deklaráció után a *ptr* az osztály bármely olyan tagjára mutathat, amelyik *típus* típusú.

Most egy rövid példán keresztül bemutatjuk a .* operátor használatát. Fordítsunk különleges figyelmet arra, hogy a tagra hivatkozó mutatókat hogyan deklaráljuk.

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

class cl {
public:
    cl(int i) { val=i;} //konstruktor
    int val;
    int double_val() { return val+val; }
};

int main ()
{
    int cl::*data; //a data a cl egy int tagjára
                   //hivatkozó mutató
    int (cl::*func)();//a func nevű tagra hivatkozó
                      //mutató deklarálása
    cl ob1(1), ob2(2);// objektumok létrehozása

    data = &cl::val; // eltolás betöltése
    func = &cl::double_val; // eltolás betöltése

    cout << "Az értékek: ";
    cout << ob1.*data << " " << ob2.*data << "\n";
    cout << "Megduplázva: ";
    cout << (ob1.*func)() << " ";
    cout << (ob2.*func)() << "\n";

    return 0;
}
```

■ *A :: hatáskör-felbontó operátor*

A :: hatáskör-felbontó operátorral egy adott névtér- vagy osztálybeli tagot érhetünk el. Általános alakban:

név::tagnév

ahol a *név* azon osztály vagy névtér azonosítója, amely a használni kívánt *tagnév* azonosítójú tagot tartalmazza.

Amikor a globális névtér alatti változókra hivatkozunk, nem jelölünk meg névteret. Ha például egy **darab** nevű globális változó nem látszik egy ugyanilyen nevű lokális változó miatt, akkor így hivatkozhatunk rá:

```
::darab
```

A hatáskör-felbontó operátort a C nem támogatja.

■ *A new és a delete operátorok*

A **new** és **delete** a C++ dinamikus memóriafoglaló operátorai, egyben kulcsszavak is. Bővebben ezekről az 5. fejezetben lesz szó.

A C nem támogatja sem a **new**, sem a **delete** operátort.

■ *A typeid operátor*

C++-ban a **typeid** operátor egy **type_info** típusú objektum címét adja vissza, amely az operandusaként szereplő objektum típusáról ad leírást. A **typeid** általános alakban:

typeid (*object*)

A **typeid** operátor a C++-ban lehetővé teszi a futásidejű típusazonosítást (RTTI, az angol Runtime Type Identification kifejezésből).

A **type_info** osztály az alábbi publikus tagokat definiálja:

```
bool operator == (const type_info &ob) const;
bool operator != (const type_info &ob) const;
bool before (const type_info &ob) const;
const char *name( ) const;
```

A túlterhelt == és != operátorokkal a típusok összehasonlítását végezhetjük. A **before( )** függvény logikai igaz értékkel tér vissza, ha a hívó

objektum a paraméterként megjelölt objektum előtt van sorrendben. (Ez a függvény főleg belső használatra készült. Visszatérési értékének semmi köze nincs az osztályok öröklődési hierarchiájához.) A **name( )** függvény egy olyan pointert ad vissza, amely az objektum típusnevét tartalmazó stringre mutat.

A **typeid**-t egy többalakú osztály alaposztályára mutató pointerre alkalmazva automatikusan a mutatott objektum típusát kapjuk vissza. (Többalakú vagy polimorfikus osztály az, amelyik legalább egy virtuális metódussal rendelkezik.) Így a **typeid** egyik hasznos tulajdonsága, hogy lekérdezhetjük az alaptípusú objektum dinamikus típusát.

A C nem támogatja a **typeid** operátort.

■ *Operátorok túlterhelése*

C++ egyik sajátossága, hogy az egyes operátornevek túlterhelhetők. Ehhez eszközként az **operator** kulcsszó szolgál (l. 5. fejezet). Operátorok túlterhelése a C-ben nem megengedett.

■ *Operátor precedencia összefoglaló*

A következő táblázatban az összes C és C++ operátor precedencia-hierarchiáját adjuk meg. Nem árt megjegyezni, hogy minden operátor, kivéve az unáris műveleteket, az értékadásokat és a ? operátort, balról jobbra kapcsolódnak egymáshoz a kiértékelési sorrendben.

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

4. FEJEZET

Az előfeldolgozóról (preprocesszor) és a megjegyzések használatáról

A C-ben és C++-ban számos preprocesszor direktíva áll rendelkezésünkre. Ezekkel közvetlenül a fordítóhoz juttathatunk el instrukciókat. Az előfeldolgozó direktívák teljes listája alább látható:

#define	#error	#include
#elif	#if	#line
#else	#ifdef	#pragma
#endif	#ifndef	#undef

Ebben a fejezetrészben röviden ismertetjük mindegyik funkcióját.

■ *#define*

A **#define** direktívával makróhelyettesítéseket végeztethetünk a fordítóval, az aktuális forrásállomány bizonyos karaktersorozatának minden előfordulását helyettesíti más, általunk meghatározott karaktersorozattal. A direktíva általános alakja:

#define *makrónév karaktersorozat*

Amikor a fordító egy *makrónévhez* ér, azt automatikusan helyettesíti a megadott *karaktersorozattal.* Vegyük észre, hogy az utasítás végén nincs pontosvessző. A karaktersorozat végét a sorvége jel jelzi.

Abban az esetben például, amikor a „TRUE" szót 1-ként a „FALSE" szót 0-ként kívánjuk használni (a forráskód jobb átláthatósága kedvéért gyakran érdemes ilyet csinálni), deklaráljuk az alábbi két makróhelyettesítést:

```
#define TRUE 1
#define FALSE 0
```

Ettől kezdve a fordító 1-et, illetve 0-át fog érteni minden TRUE, illetve FALSE szó alatt.

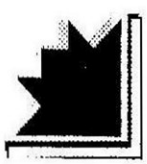
A **#define** direktíva lehetőséget ad argumentumokkal rendelkező makrók definiálására is. Az ilyen makrók sokban hasonlítanak a függvényekre, ezért a szakirodalom ezekre gyakran függvény típusú makróként is hivatkozik. Az argumentumokat az előfeldolgozó a programban található aktuális argumentumokra cseréli. Tekintsük a következő példát:

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

#define ABS(a) ((a)<0 ? -(a) : (a))

int main()
{
cout << "-1 és 1 számok abszolútértéke:" << ABS(-1);
     << ' '    << ABS (1);
return 0
}
```

A program fordítása során a makródefinícióban szereplő **a**-ba a –1, illetve az 1 értékek másolódnak.



Programozási tipp

Lényeges, hogy mindig a megfelelő módon zárójelezzük függvénytípusú makróinkat. Ha nem így tesszük, akkor adódhatnak olyan helyzetek, amelyekben nem működnek helyesen. Az előző példában az **a** *zárójelek közé helyezésével garantáltuk a minden esetben helyes helyettesítést. Amennyiben a zárójeleket elhagyjuk, az*

```
ABS (10-20)
```

kifejezés a makróhelyettesítés után átmenne a

```
10-20<0 ? -10-20 : 10-20
```

sorba, ami nyilvánvalóan rossz eredményre vezet. Érdemes tehát a nem megfelelően működő makróinkban először a zárójelezés helyességét ellenőrizni.

■ *#error*

Az #error direktívával a fordítás felfüggesztésére kényszeríthetjük a fordítót. Ahogy a fordító ezzel a direktívával találkozik, abbahagyja a fordítást. Elsődlegesen nyomkövetési funkciókat tölt be. Általános alakja:

#error *üzenet*

A fordítás megállásakor az üzenet és a **#error**-t tartalmazó sor száma jelenik meg a képernyőn.

■ *#if, #ifdef, #ifndef, #else, #elif, #endif*

Az **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif** és **#endif** preprocesszor direktívák arra szolgálnak, hogy a program különböző részeit szelektíven tudjuk fordítani. Az alapötlet az, hogy az **#if**, **#ifdef** vagy **#ifndef** után álló logikai kifejezés fennállásakor a kód onnantól egészen a következő **#endif**-ig lefordítódik, különben kimarad. Az **#endif** jelöli az **#if** blokk végét. Az **#else** szerepe – a fentiek bármelyikével együtt használva – az alternatíva biztosítása.

Az **#if** általános formája:

#if *konstans_kifejezés*

Ha a *konstans_kifejezés* igaz, a soronkövetkező utasítások lefordítódnak (legalábbis a következő **#endif**-ig vagy **#else**-ig).

Az **#ifdef** általános alakja:

#ifdef *makrónév*

Ha az **#if** kiértékelésekor a *makrónév* nevű makró definiált egy korábbi **#define** által, akkor az utasítást követő kódrész lefordítódik. Az alábbi példában megfigyelhetjük, hogyan működnek együtt ezek a preprocesszor direktívák. A program megjeleníti a „Hello Laci!" és a „Hello Karcsi!" üzeneteket, de nem írja ki a „Hello Lajos!" köszönést:

```
#define Laci 10

// ...

#ifdef Laci
 cout << "Hello Laci\n";
#endif
 cout << "Hello Karcsi\n";
#if 10<9
 cout << "Hello Lajos\n";
#endif
```

Az **#elif** direktíva valójában egy **if-else-if** utasítást hivatott helyettesíteni. Szintaxisa:

#elif *konstanskifejezés*

Több #elif kifejezést is összekapcsolhatunk, lehetővé téve ezáltal több ágú feltételes fordítást is. Az **#if** és **#elif** operátorok használhatók a **defined** előfeldolgozó operátorral, aminek eredményeként eldönthetjük, hogy egy adott makrónév definiált-e a kód aktuális részében. Általános alakban:

#if defined *makrónév*
utasítássorozat
#endif

Ha a *makrónév* nevű makró definiált, az utasítássorozat lefordul, egyébként a fordító figyelmen kívül hagyja. Az alábbi programrészlet például lefordítja a feltételes kódot, mert a **DEBUG** makrót előzőleg definiáltuk.

```
#define DEBUG

// ...
int i=100;
// ...

#if defined DEBUG
cout << "i értéke:" << i << endl;
#endif
```

A **defined** operátor elé helyezett negációval (!) érhetjük el, hogy a fordítás a nem definiált makrónév esetén történjen.

■ *#include*

Az **#include** direktíva utasítja a fordítót, hogy olvasson be és fordítson le egy másik forrásállományt is az aktuális állomány fordítása előtt. Két általános alakja:

```
#include "állománynév"

#include <állománynév>
```

A beolvasandó forrásállomány nevét idézőjelek vagy könyökzárójelek között tüntetjük fel. Például a

```
#include <stdio.h>
```

sor lefordíttatja a C be- és kimeneti (I/O) függvények fejrészét.

Könyökzárójelek között szereplő állománynevek keresésének rendje fordítóspecifikus. Gyakran ez egy külön könyvtárban történő keresést jelent, ami speciálisan a szabványos fejinformációk tárolását szolgálja. Amennyiben az állománynevet idézőjelek közé zárjuk, akkor az állományt más implementációfüggő helyen keresi a fordító. Sok fordító esetén ez az aktuális könyvtárban való keresést jelenti. Ha az állományt a fordító nem találja, akkor a keresés megismétlődik úgy, mintha a zárójé-

les megadási módot használtuk volna. Mindenképpen érdemes átnéznünk a fordítóhoz mellékelt dokumentációt, hogy kiderüljön a két keresési algoritmus közti különbség. Az **#include** utasítások egymásba ágyazhatók, azaz importált állományokon belül is lehetnek további **#include** direktívák.

A C++ összes verziója elfogadja az **#include** direktíva fent leírt használati módjait, a szabvány C++ ezenkívül új stílusú fejeket is definiál, amelyek információkat tartalmaznak a standard C++ könyvtárról. Ezek a fejek nem állománynevek (azonban lehet, hogy megfeleltethetők állományneveknek). Új stílusú fejek importálásának általános alakja:

#include <*fejnév*>

ahol a *fejnév* a 2. fejezetben leírt új stílusú fejek egyike.

Például az I/O rendszerhez kapcsolódó könyvtár fej információinak importálása így történik:

```
#include <iostream>
```

Mivel az új stílusú fejek nem fájlnevek, nem rendelkeznek .H kiterjesztéssel. További felvilágosítást a standard C++ fejekről és azok beágyazásáról a használt fordító dokumentációjából kaphatunk.

■ *#line*

A **#line** direktíva segítségével a __LINE__ és __FILE__ predefinit azonosítók tartalmát változtathatjuk meg. A parancs alakja:

#line *szám "állománynév"*

ahol a *szám* lehet bármilyen pozitív egész, az *állománynév* bármilyen érvényes fájlazonosító. A *szám* értéke a következő fordítandó forrássor számát, míg az állománynév a fordítandó forrásállományt adja meg. Az *állománynév* használata opcionális. A **#line** tulajdonképpen „becsapja" a fordítót, elsősorban diagnosztikai célokra használjuk.

A **__LINE__** azonosító értelemszerűen egésztípusú, a **__FILE__** pedig egy nullvégű string.

Az alábbi részlet a sorszámláló pillanatnyi értékét 10-re állítja be és fordítandó forrásfájlként a „teszt" nevűt jelöli meg:

```
#line 10 "teszt"
```

■ *#pragma*

A **#pragma** egy implementációfüggő direktíva, amely különböző instrukciók közvetítését teszi lehetővé a fordítóhoz. Vannak fordítók például, amelyek lehetővé teszik a végrehajtás nyomkövetését, ekkor a nyomkövetés kérését a **#pragma** direktíván keresztül tolmácsolhatjuk a fordítónak. A lehetőségek teljes körű feltérképezése érdekében forduljunk a fordító dokumentációjához.

■ *#undef*

Az **#undef** direktívával egy korábbi makródefiníció érvényességét szüntethetjük meg.

Általános alakja:

#undef *makrónév*

Például az itt közölt programrészletben mind a **LEN**, mind a **WIDTH** makrók definiáltak egészen az **#undef** utasítás felbukkanásáig:

```
#define LEN 100
#define WIDTH 100

char array [LEN] [WIDTH];

#undef LEN
#undef WIDTH
/* Ezen a ponton sem a LEN, sem a WIDTH nincs definiálva*/
```

■ *A # és a ## preprocesszor operátorok*

C/C++-ban két preprocesszor operátor áll rendelkezésünkre: a # és a ##. Ezeket **#define**-makrókban használjuk.

A # operátorral garantálhatjuk, hogy a mögötte található argumentumba idézőjelek közé zárt string kerüljön. Tekintsük példaként az alábbi programot:

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

#define mkstr(s) # s

int main()
{
 cout << mkstr(Tetszik a C++);
 return 0;
}
```

Az előfeldolgozó a

```
cout <<mkstr(Tetszik a C++);
```

sort

```
cout << "Tetszik a C++";
```

sorral helyettesíti.

A ## operátorral két alapszimbólumot konkatenálhatunk (fűzhetünk össze). Ezt illusztrálja a következő program:

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

#define concat(a, b) a ## b

int main ()
```

```
{
 int xy = 10;
 cout << concat(x, y);
 return 0;
}
```

A preprocesszor a

```
cout << concat(x, y);
```

sort

```
cout << xy;
```

utasítássá alakítja.

Első látásra ezek az operátorok furcsának tűnhetnek az olvasó számára, megnyugtatásul megjegyezzük, hogy a legtöbb programban ezekre nincs szükség. Létjogosultságukat elsősorban az magyarázza, hogy lehetővé teszik, hogy néhány különleges esettel a preprocesszort terheljük.

■ *Predefinit makrónevek*

C/C++ öt beépített makrónevet biztosít számunkra. Ezek a következők:

__LINE__

__FILE__

__DATE__

__TIME__

__cplusplus

A **__LINE__** és a **__FILE__** makrókról már beszéltünk korábban a **#line** címszó alatt. A többit itt részletezzük:

A **__DATE__** makró egy *hónap/nap/év* formátumú string, amely a forrásállomány tárgykóddá (object code) alakításának dátumát foglalja magában.

A forrásállomány tárgykóddá alakításának idejét a **__TIME__** string tárolja *óra:perc:másodperc* formátumban.

A **__cplusplus** makró C++ programok fordításakor nyer értelmet. Ezt a makrót C fordítók nem definiálják. C programok fordításakor az **__STDC__** makrót definiálja a C++ fordító. További részleteket a fordító dokumentációjából tudhatunk meg. A legtöbb C/C++ fordító számos más beépített makrót is biztosít a programozóknak. Ezek általában környezet- és implementációfüggők.

■ *Megjegyzések*

C++ forráskódjainkba kétféleképpen helyezhetünk el megjegyzéseket. Az egyik módszerrel többsoros megjegyzéseket iktathatunk be. Ekkor a megjegyzés szövegét a /* és */ jelek közé kell helyeznünk. Minden, a megjegyzés szimbólumok közé eső karaktert a fordító figyelmen kívül hagy. Az ilyen megjegyzések több sort is elfoglalhatnak.

A másik fajtához az egysoros megjegyzések tartoznak. Ezek egy // jelpárral kezdődnek és egészen a sor végéig tartanak.

A C kizárólag a többsoros megjegyzést támogatja. A legtöbb C fordító azonban elfogadja az egysoros megjegyzést is, még ha az nem is szabvány követelmény.

5. FEJEZET

Kulcsszavak összefoglalása

A C nyelv az alábbi 32 kulcsszót tartalmazza:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

A C++ a fentieken kívül még a következõ kulcsszavakkal rendelkezik:

asm	inline	template
bool	mutable	this
catch	namespace	throw
class	new	true
const_cast	operator	try
delete	private	typeid
dynamic_cast	protected	typename
explicit	public	using
false	reinterpret_cast	virtual
friend	static_cast	wchar_t

A C++ korábbi változatai még tartalmazták az **overload** kulcsszót, ez már elavultnak számít. Minden kulcsszó csupa kisbetűből áll. Az egyes kulcsszavak rövid ismertetése következik.

■ *asm*

Az **asm** kulcsszót assembly nyelvű kódrészlet C++ programunkba történő közvetlen beágyazására használjuk. Általános alakja:

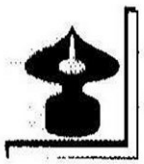
```
asm ("utasítás");
```

ahol, *utasítás* egy assembly nyelvű utasítás, amely az adott helyen beágyazódik a programba.

Sok C++ fordító az **asm** használatának más formáit is támogatja. Például a Borland C++ az alábbi utasításokat is megengedi:

```
asm utasítás;
asm {
      utasítássorozat
}
```

ahol az *utasítássorozat* nem más, mint assembly nyelvű utasítások listája.



Megjegyzés
A Microsoft Visual C++ az **__asm** *kulcsszót használja assembly kód beágyazására. Nevétől eltekintve úgy kezelendő, mint az* **asm**.

■ *auto*

Az **auto** kulcsszóval lokális változókat deklarálhatunk. Teljesen opcionális, ritkán használt.

■ *bool*

A **bool** típusazonosítóval logikai (true/false – igaz/hamis) változókat deklarálhatunk.

■ *break*

A **break** kulcsszó hatására a vezérlés azonnal kilép a **do**, **for**, vagy **while** ciklusból, kikerülve a szokásos feltételvizsgálatot. Másik funkciója a **switch** utasításból való kilépés.

A következő példában a **break** ciklusbeli használatát illusztráljuk:

```
do {
  x = getx();
  if(x < 0) break; // negatív x esetén kilépés
  process(x);
  }while(!done);
```

Amennyiben **x** negatív, a ciklusból kilép a végrehajtás.

A **break** kulcsszónak **switch** utasításon belül az a szerepe, hogy megakadályozza, hogy a vezérlés a követező **case**-re ugorjon. (További részleteket a „**switch**" címszó alatt találhatunk.)

A **break** természetesen csak az őt tartalmazó **for**, **do**, **while** és **switch**-ből lép ki. Nem lép ki azonban egymásba ágyazott ciklusokból vagy **switch** utasításokból.

■ *case*

A **case** utasítást a **switch** utasítással együtt használjuk. További részleteket a „**switch**" címszó alatt találhatunk.

■ *catch*

A **catch** utasítás a **throw** által kiváltott kivételek lekezelését végzi. Lásd még a „**throw**" címszó alatt.

■ *char*

A **char** a karaktertípus típusazonosítója, segítségével deklarálhatunk karakterváltozókat.

■ *class*

A **class** kulcsszó segítségével definiálhatunk osztályokat. C++-ban az osztály a zártság (encapsulation) kritériumának megfelelő alapegységet jelenti. A **class** általános szintaxisa:

```
class osztálynév: öröklődési_lista {
   //privát tagok
protected:
   //olyan privát tagok, amelyekből leszármazottak hozhatók létre
public:
   //nyilvános tagok
} objektumlista
```

ahol az *osztálynév* a **class** deklarációval létrehozott új adattípus nevét jelenti. Az opcionális *öröklődési_lista* az új osztály őseit nevezi meg. Alapértelmezésben az osztály tagjai kifelé nem láthatóak. Védetté vagy publikussá a **protected** ill. a **public** kulcsszavakkal tehetők.

Az *objektumlista* is opcionális. Ha hiányzik, akkor az osztálydeklaráció csupán az osztály specifikációját jelenti, nem hoz létre példányokat.

Megjegyzés
Az első fejezet részletesen tárgyalta az osztályok fogalmát.

■ *const*

A **const** módosító jelzi a fordítónak, hogy az őt követő változó nem módosítható a programfutása során. A konstans változóknak kizárólag inicializáláskor adhatunk értéket.

■ *const_cast*

A **const_cast** operátort akkor használjuk, ha fel akarjuk oldani a **const** és/vagy a **volatile** módosító hatását egy adott változóra. Általános formája:

const_cast<*típus*>(*objektum*)

A céltípusnak ebben az esetben meg kell egyeznie a kiindulási típussal, változás kizárólag annak **const** vagy **volatile** attribútumában következik be. Leggyakrabban a „konstansság" megszüntetésére használják.

■ *continue*

A **continue** kulcsszót használjuk, ha a ciklus hátralevő részét át kívánjuk ugrani. Hatására a következő iterációs lépés veszi kezdetét. Az alábbi példában szereplő **while** a billentyűzetről olvas karaktereket, egészen addig, amíg egy s betűt nem kap.

```
while(ch = getchar()) {
  if(ch != 's') continue; //következő karakter
                          //beolvasása
  process(ch);
}
```

A **process( )** hívására egészen addig kell várni, amíg a **ch** tartalma egy 's' karakter karakter lesz.

■ *default*

A **default** kulcsszót a **switch** utasítás belsejében használhatjuk. Ha nincs a **switch** blokkban igazként kiértékelődő **case**-ág, akkor **default** után álló blokk hajtódik végre. (Lásd még a „**switch**" címszó alatt.)

■ *delete*

A **delete** operátor az argumentuma által mutatott memóriaterület felszabadításáról gondoskodik. Előfeltétel, hogy a felszabadítani kívánt memóriát előzetesen lefoglaltuk a **new** utasítással. Általános alakja:

delete *mutató;*

ahol a *mutató* egy előzetesen lefoglalt memóriaterületre mutató pointer.

Tömböknek lefoglalt memória felszabadítása (az előzetes allokáció után) a következőképpen történik a **delete** használatával:

delete [ ] *mutató;*

■ *do*

A **do** ciklus egyike a C++-ban elérhető három cikluskonstrukciónak. Általános szintaxisa:

```
do {
      utasítássorozat
} while(feltétel);
```

Ha a ciklusmag csak egy utasításból áll, a kapcsos zárójelek elhagyhatók, használatuk azonban ekkor is célszerű a kód átláthatósága érdekében.

A **do** ciklus sajátossága, hogy a ciklusmag legalább egyszer mindenképpen végrehajtásra kerül, mert a feltételellenőrzés a ciklus végén található.

■ *double*

A **double** típusspecifikátor segítségével dupla pontosságú lebegőpontos változókat deklarálhatunk.

■ *dynamic_cast*

A **dynamic_cast** operátorral olyan futási idejű típusmódosítást végezhetünk, amely egyben ellenőrzi a konverzió helyességét is. Általános alakja:

dynamic_cast <*típus*> (*objektum*)

Fő felhasználási területét a polimorf típusok konverziója adja. Példaként tekintsünk két polimorf osztályt, nevezzük ezeket B-nek és D-nek, ahol D a B-nek egy leszármazottja. A **dynamic_cast**-tal történő konverzió mindig kivitelezhető, ha D* mutatót alakítunk B*-gá. Visszafelé csak akkor működik a dolog, ha a B* típusú pointer címén ténylegesen egy D típusú objektum található. Általánosan fogalmazva, egy **dynamic_cast** sikeressége azon múlik, hogy a megkísérelt polimorfikus konverzió engedélyezett-e (vagyis hogy a céltípus legálisan alkalmazható-e a konvertálandó objektumra). Ha a cast nem végezhető el, akkor eredményül nullpointert kapunk.

■ *else*

Lásd „**if**"

■ *enum*

Az **enum** típusspecifikátorral felsorolt típusok előállítását végezhetjük. Egy felsorolt adattípus tulajdonképpen nem más, mint névvel ellátott egészkonstansokból álló lista. Általános alakja:

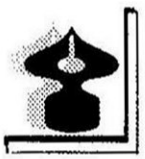
enum *enum_név* {*névlista*} *változólista;*

Az *enum_név* az **enum**-mal létrehozott típus azonosítóját jelenti. A *változólista enum_név* típusú változók deklarációja. Használata opcionális, hiszen változóinkat nem szükséges a típusdefinícióban deklarálnuk, néha azonban – tömörségi megfontolások miatt – célszerű. Az alábbi példa-

program egy **szín** nevű enumerációt deklarál és egy ilyen típusú **c** változót. A lényegi rész egy értékadásból és egy feltételvizsgálatból áll.

```
enum szin {piros, zold, sarga} c;

c = piros;
if(c==piros) cout << "piros\n";
```



Megjegyzés
Az 1. fejezetben további információkat találhatunk az enumerációkra vonatkozóan.

■ *explicit*

Az **explicit** specifikátornak csak konstruktorokban van értelme. Az **explicit** kulcsszóval megjelölt konstruktorok használata garantálja, hogy az inicializálás csak akkor megy végbe, ha az formailag megfelel a konstruktorban megadottaknak. Automatikus konverzió ilyenkor nem történik. („Nem konvertáló konstruktort" hoz létre.)

■ *extern*

Az **extern** az adattípus-módosítók közé tartozik. Arra szolgál, hogy a fordítónak jelezze egy olyan változó jelenlétét, amelyet a program egy más részében definiáltunk. Gyakran ez más forrásállományban definiált globális változót jelent, ekkor az összeszerkesztés helyessége miatt van szükség az **extern** használatára. Összefoglalva az **extern** a fordítót informálja egy változó típusáról, anélkül, hogy azt újradefiniálná. Példaként tegyük fel, hogy a **first** egész típusú változót egy másik forrásállományban már definiáltuk. Ekkor az összes további állományban az alábbi sort kell beiktatnunk, ha ugyanezt a változót kívánjuk használni.

```
extern int first;
```

Ez a deklaráció megjelöli a **first** típusát, de nem foglal számára erőforrást.

C++-ban az **extern**-t még külső függvények beágyazására is használjuk. Szintaxisa:

extern *"programnyelv" függvény-prototípus*

A *programnyelv* azt mondja meg, hogy a beszerkesztendő függvény milyen nyelven készült. A C és C++ linkelés támogatása garantált. A fordítótól függően más szerkesztések is elképzelhetőek. Egy kalap alatt több külső függvényt is deklarálhatunk az alábbi szerkesztési specifikáció mintájára:

```
extern "programnyelv" {
        függvény-prototípusok
}
```

■ *false*

A **false** logikai konstans a logikai nem értéket jelenti.

■ *float*

A **float** egy adattípus-specifikátor, lebegőpontos számok deklarálásához használjuk.

■ *for*

A **for** ciklus automatikus inicializálást és egy számláló inkrementálását teszi lehetővé. Általános alakban:

```
for (inicializálás, feltétel, inkrementálás) {
   utasítássorozat
}
```

Ha az utasítássorozat egyetlen utasítást jelent, akkor a kapcsos zárójelek elhagyhatók.

A **for** ciklus *inicializálás részébe* – habár nem törvényszerű – általában a ciklusváltozó kezdőértékének beállítását végző utasítás kerül. A *feltétel* általában egy olyan reláció, amely a ciklusváltozót hasonlítja össze valamilyen terminálló értékkel, az *inkrementálás* a ciklusváltozó növelését, illetve csökkentését végzi. Ha a *feltétel* már ciklusba lépéskor hamis, akkor a **for** ciklus magja egyszer sem hajtódik végre.

Az alábbi utasítás tízszer kiírja a képernyőre a „hello" üzenetet:

```
for(t=0; t<10; t++) cout << "hello/n";
```

■ *friend*

A **friend** (magyarul: barát) kulcsszó segítségével egy adott osztályhoz nem tartozó függvény számára lehetővé tehetjük, hogy az osztály privát tagjaihoz hozzáférjen. Ha egy függvényt egy adott osztály „barátjaként" kívánunk kezelni, akkor a prototípusát az osztálydeklaráció **public** részében kell elhelyeznünk, úgy, hogy a deklarációt a **friend** kulcsszó vezesse be. Az alábbi példában a **myfunc( )** egy ilyen függvény, nem tagja a **myclass**-nak.

```
class myclass {
  //...
public:
  friend void myfunc (int a, float b);
  //...
};
```

Érdemes megjegyezni, hogy a **friend** függvényeknek nincs **this** mutatójuk, ez ismételten abból adódik, hogy nem osztálytagok.

■ *goto*

A **goto** kulcsszó lehetővé teszi, hogy a vezérlés a megadott címkénél folytatódjon. A **goto** általános alakja:

goto *címke;*

.

.

.

címke:

Minden címkét kettőspont kell, hogy kövessen. Továbbá ügyelnünk kell arra, hogy ne használjunk olyan címkét, amely megegyezik valamelyik kulcsszóval vagy függvénynévvel. A **goto**-val csak az aktuális függvényen belüli címkékre ugorhatunk – más függvény belsejébe nem.

Programozási tipp

*Habár a **goto**, mint a programvezérlés egy eszköze, évtizedekkel ezelőtt elvesztette a programozók kegyét, mégis vannak olyan szituációk, amelyekben hasznos lehet. Ezek közül az egyik egy többszörösen egymásba skatulyázott ciklusrendszer belsejéből történő kilépés. Tekintsük az alábbi kódrészletet:*

```
int i, j, k;
int stop = 0;

for(i=0; i<100 && !stop; i++) {
  for(j=0; j<10 && !stop; j++) {
    for(k=0; k<20; k++) {
      //...
      if(valami()) {
        stop = 1;
        break;
      }
    }
  }
}
```

*Látható, hogy a **stop** változó gondoskodik arról, hogy bizonyos esemény bekövetkezésekor a két külső ciklusból kilépjünk. A **goto** használatával ez a program így egyszerűsíthető:*

```
int i, j, k
for(i=0; i<100; i++) {
  for(j=0; j<10; j++) {
    for(k=0; k<20; k++) {
      //...
      if(valami()){
      goto done;
      }
    }
  }
}

done: //...
```

*A **goto** segítségével megszabadultunk attól az extraköltségtől, amit a **stop** ismételt ellenőrzése jelentett az előző változatban. A **goto** használata a legtöbb esetben kerülendő, vannak azonban kivételek, amikor használata célszerű lehet.*

■ *if*

Egy feltétel eredményétől függő utasítás-végrehajtást az **if** kulcsszó teszi lehetővé. Általános alakja:

```
if (feltétel) {
   utasításblokk1
}
else {
   utasításblokk2
}
```

Ha elemi utasításokat használunk, a kapcsos zárójelek elhagyhatók. Az **else**-ág opcionális.

A *feltétel* bármilyen kifejezés lehet. Ha igazra értékelődik ki (értéke 0-tól különböző), akkor az *utasításblokk1* végrehajtódik, egyébként – ha létezik – az *utasításblokk2* kerül végrehajtásra.

Az alábbi programrészlet megvizsgálja, hogy **x** értéke nagyobb-e, mint 10.

```
if (x > 10)
  cout << "x nagyobb, mint 10";
else
  cout << "x kisebb, mint 10";
```

■ *inline*

Az **inline** specifikátor utasítja a fordítót, hogy a függvény kódját – ha lehet – építse be a hívás helyére, ahelyett, hogy valóban elvégezné a hívást. Az **inline** nem tekinthető parancsnak, csak egy kérést tolmácsol, amelyet a fordító bizonyos esetekben figyelmen kívül hagyhat. Vannak ugyanis olyan helyzetek, amikor ez a kifejtés nem végezhető el: pl. ha a beépítendő függvény rekurzív, illetve ha ciklust, **switch** utasítást vagy statikus adatokat tartalmaz. Az **inline** specifikátort a függvénydeklaráció elé kell beszúrni.

Az alábbi sor a **myfunc( )** függvény kódjának kifejtését kéri a rá hivatkozó helyeken.

```
inline void myfunc(int i)
{
  //...
}
```

Minden olyan függvényt, amelynek definíciója osztálydeklaráción belül található, a fordító automatikusan **inline** függvényként kezel.

■ *int*

Az **int** egy típusspecifikátor, segítségével egész típusú változókat hozhatunk létre.

■ *long*

A **long** a hosszú egészek deklarálásakor használt adattípus-módosító.

■ *mutable*

A mutable specifikátor egy objektum tagját mentesti a **const** hatálya alól. Egy konstans objektum **mutable** specifikátorral megjelölt tagja módosíthatóvá válik. Legtöbbször arra használjuk, hogy **const** függvénytagok számára lehetővé tegyük egyes adattagok módosítását.

■ *namespace*

A **namespace** kulcsszó segítségével megoldhatóvá válik, hogy olyan tartományokat definiáljunk változóinkhoz, amelyekre azok közvetlen hozzáférhetősége korlátozható. Célja a nevek lokalizálása. A *névterek* segítségével tehát deklaratív tartományokat hozhatunk létre. Általános szintaxisuk:

```
namespace név {
  //deklarációk
}
```

Azonosító nélküli névtereket is definiálhatunk, az alábbiak mintájára:

```
namespace {
  //deklarációk
}
```

Azonosító nélküli névterek lehetővé teszik olyan egyedi változók létrehozását, amelyek csak az adott állományban érhetők el.

Az alábbi példa a **namespace** használatát illusztrálja.

```
namespace MyNameSpace {
  int i, k;
  void myfunc(int j) {cout << j; }
}
```

Itt az **i**, **k** változók és a **myfunc( )** függvény alkotják a **MyNameSpace** névteret.

Mivel a névtér egy láthatósági tartományt definiál, a belsejében definiált objektumokra a hatáskörfelbontó operátorral hivatkozhatunk. Ha az előbb deklarált **i**-nek 10-et szeretnénk értékül adni, akkor az alábbi utasítást használjuk:

```
MyNameSpace::i = 10;
```

Amennyiben a névtér tagjaira gyakran hivatkozunk, akkor – elérésüket egyszerűbbé téve – érdemes a **using** direktívát használni. A **using** kulcsszó az alábbi két alakban használható:

using namespace *név;*

using *név::tag;*

Az első formában a *név* a „látni" kívánt névtér azonosítóját jelenti. Ekkor a névtéren belül definiált bármely taghoz közvetlenül hozzáférhetünk. A második alakkal csak a névtér egy tagját tehetjük láthatóvá. Tekintsük az alábbi programrészletet, és tegyük fel, hogy a **MyNameSpace**-t a fentieknek megfelelően már definiáltuk.

```
using MyNameSpace::k; //csak a k-t teszi láthatóvá

k = 10; //jó, mert k elérhető

using namespace MyNameSpace;
  // minden MyNameSpace-beli tag látható
i = 10; // így i-re is hivatkozhatunk
```

■ *new*

A **new** operátor memóriaterület foglalását végzi dinamikus módon (az adott adattípusnak megfelelően) és a foglalt területre mutató megfelelő típusú pointerrel tér vissza. Általános formában:

m_változó = new *típus;*

ahol az *m_változó* az a mutató, amely megkapja az allokált memóriatartomány címét, a *típus* azt az adattípust jelenti, ami ezen a címen tárolásra kerül. A **new** operátor automatikusan megfelelő méretű memóriát foglal ahhoz, hogy a megadott típus egy eleme tárolható legyen. Az alábbi részletben memóriát foglalunk egy **double** típusú adat tárolására:

```
double *p;
p = new double;
```

Ha az allokáció nem végezhető el, két dolog történhet: vagy nullpointert kapunk vissza, vagy egy **bad_alloc** kivétel váltódik ki. (Újabban néhány fordítóval az **xalloc** kivételt kapjuk.)

Megjegyzés
Jelenleg (e könyv írásakor) az, hogy a **new** *hogyan viselkedjen hiba esetén, még nem alakult ki minden részletében. Különböző fordítók másmás módon kezelik az allokálási hibákat. Ezért az olvasó jól teszi, ha megnézi a használt fordító dokumentációját, figyelembe véve az aktuális munkakörnyezetet.*

Lehetőség van a lefoglalt memória inicializálására az alábbi módon:

m_változó = new *típus* (*inicializáló*);

ahol, az *inicializáló* a lefoglalt memóriatartományba másolandó kezdeti érték.

Egy egydimenziós tömb allokálását az alábbi sor mintájára tehetjük meg:

m_változó = new *típus [méret];*

ahol a *méret* a tömb hosszát jelenti. A **new** ebben az esetben is elegendő helyet foglal a megadott méretű és típusú tömb tárolására. Tömböknek foglalt memóriaterületek nem inicializálhatóak.

■ *operator*

Az **operator** kulcsszó lehetővé teszi, hogy predefinit operátorokat túlterhelhessünk. A létrehozni kívánt túlterhelt operátorokat két csoportra oszthatjuk, aszerint, hogy osztályok tagjai-e (metódusok), vagy sima függvények. Egy operátormetódus általános alakja így néz ki:

```
visszatérésitípus osztálynév::operator # (paraméterlista) {
  // . . .
}
```

ahol *visszatérésitípus* a függvény visszatérési típusa, az *osztálynév*, azon osztálynak a neve, amelyre nézve az operátor túlterhelt, végül a # magát a túlterhelt operátort jelöli. Egy unáris operátor túlterhelésénél a *paraméterlista* üres marad. (Az operandus implicit módon a **this**-szel adódik át.). Ha bináris operátort terhelünk túl, akkor a *paraméterlista*-ban az operátor jobb oldalán szereplő operandust adjuk meg. (Az bal oldali operandus implicit módon a **this**-szel adódik át.)

Nem tagfüggvények esetén túlterhelt operátorhoz így rendelhetünk egy másik definíciót:

```
visszatérésiérték operator # (paraméterlista) {
// . . .
}
```

ahol a *paraméterlista* egyetlen paraméterből áll, ha egy uniáris operátort terhelünk túl és két paraméterből, ha egy bináris operátort definiálunk felül egy általunk létrehozott típusra. A bal oldali operandust az első, a jobb oldalit a második paraméter képviseli.

Számos megszorítás vonatkozik az operátorok túlterhelésére. Először is, az operátor precedenciája nem változtatható meg. Az operandusok száma nem módosulhat az eredetihez képest. A C++ beépített típusaira vonatkozólag nem változhat meg az operátorok hatása. Nem hozhatunk létre új operátort. A preprocesszor operátorok (# és ##) nem terhelhetők túl. Továbbá az alábbi operátorok túlterhelése sem megengedett:

```
. : .* ?
```

■ *private*

A **private** láthatósági specifikátor segítségével egy osztályban privát függvénytagokat hozhatunk létre. Használható továbbá egy alaposztály privát módon történő származtatására. Privát tagok deklarálása az osztálydefinícióban így történik:

class *osztálynév* {
  // . . .
private:
  // priváttagok
};

Egy osztály tagjai alapértelmezésben privát minősítésűek. Ezért a **private** specifikátor használatának akkor van valódi értelme, ha egy újabb **private** blokkot akarunk nyitni egy **public** vagy **protected** deklarációsorozat után. Az alábbi érvényes osztálydeklarációban megfigyelhetjük a **private** szerepét:

```
class myclass {
  int a, b; // alapértelmezés miatt privát
public: // nyilvános deklarációk kezdete
  int x, y; // ezek publikusak
private: // visszatérés a privát deklarációkhoz
  int c, d; // ezek privát tagok
};
```

Öröklődési specifikátorként ilyen formában használjuk a **private** kulcsszót:

class *osztálynév*: private *alaposztály* { // . . .

Származtatásnál az alaposztály **private** módosítóval történő megjelölése azt eredményezi, hogy az alaposztály összes nyilvános és védett tagja a leszármazottban már privát tagként viselkedik. A privát tagok természetesen megmaradnak privátnak a gyermekosztályban is.

■ *protected*

A **protected** kulcsszóval deklarált tagok (védett tagok) az adott osztályban privátként viselkednek, de tovább öröklődnek a származtatott osztályokba. Használata:

class *osztálynév* {
  //. . .
protected: // védett tagok deklarációja következik
  // védett tagok
};

Nézzük a következő példát:

```
class alap {
  //...
protected:
  int a;
  //...
};

//Most az alap-ból leszármaztatjuk a gyermek-et
class gyermek : public alap {
  //...
public:
  //...
  void f() { cout << a; }
};
```

Az **a** az **alap** privát tagja, azaz kizárólag tagfüggvények érhetik el. A **gyermek** osztály örökli **a**-t, ha azonban az **a** eredetileg privát lett volna, **gyermek**-nek nem lett volna hozzáférése.

Öröklődési specifikátorként az alábbi kontextusban használhatjuk:

class *osztálynév*: protected *alaposztály* {// . . .

Származtatásnál az alaposztály **protected** kulcsszóval történő megjelölése azt eredményezi, hogy az alaposztály minden védett és nyilvános

tagja a gyermekosztályban védettként szerepel. Az öröklődés során nem változik az a tény, hogy az alaposztály privát tagjait csak az alaposztály függvénytagjai érhetik el.

■ *public*

A **public** láthatósági specifikátorral egy adott osztály nyilvános tagjait jelöljük meg. Használható továbbá egy alaposztály nyilvános módon történő származtatásra. Nyilvános tagok deklarálására az alábbi kontextusban használjuk:

```
class osztálynév {
   // privát tagok (alapértelmezés)
public: // nyilvános tagok deklarációja következik
   // nyilvános tagok
};
```

Alapértelmezésben az osztály tagjai privátként viselkednek. Ha nyilvános tagokat kívánunk deklarálni, akkor a **public** kulcsszót kell használni a fenti módon.

Láthatósági specifikátorként a **public** szót így használjuk:

```
class osztálynév: public alaposztály {//. . .
```

Származtatásnál az alaposztály **public** kulcsszóval történő megjelölése azt eredményezi, hogy az alaposztály minden nyilvános tagja a gyermekosztályban is nyilvános marad. Az alaposztály védett tagjai a leszármazott védett tagjaivá válnak. Az öröklődés során nem változik az a tény, hogy az alaposztály privát tagjait csak az alaposztály függvénytagjai érhetik el.

■ *register*

A **register** kulcsszóval egy kérést tolmácsolunk a fordítónak, tudniillik hogy egy adott változó elérését sebesség szerint optimalizálja (ha tudja). A C nyelv alkonyán a **register** specifikátor csak egész vagy karakter

típusú változókra vonatkozhatott, akkor a **register** használatával arra adhattunk utasítást a fordítónak, hogy kísérelje meg a változó értékét a processzor egy regiszterében tárolni ahelyett, hogy ezt a memóriában tenné. Az adott változó elérése így nagyságrendekkel gyorsabb lett. A **register** definícióját azóta kiterjesztették. Ma már minden változó ellátható a **register** módosítóval, a fordító feladata, hogy optimalizálja a hozzáférés sebességét ezen változókra. Karakterek és egészek esetén most is a CPU regiszterek használata jelenti a leggyorsabb elérés biztosítását, más adattípusok esetén viszont a gyorsítás jelentheti például a cache memória (gyorstár) használatát. Elképzelhetők olyan esetek, amikor a kérést a fordító figyelmen kívül hagyja.

A **register** kulcsszót csak lokális változókra alkalmazhatjuk. C-ben nem kérdezhetjük le a **register-rel** megjelölt változó címét. C++-ban ez ugyan lehetséges, de ekkor az optimalizálás nem garantálható.

■ *reinterpret_cast*

A **reinterpret_cast** operátorral megtehető, hogy egy változót annak típusától merőben különböző típusúvá alakítsunk. Például elvégezhető egy mutató egésszé történő konvertálása. Általános alakban:

reinterpret_cast <*típus*> (*objektum*)

A **reinterpret_cast** használandó két inkompatíbilis mutató típus közti konverzió megvalósítására.

■ *return*

A **return** utasítás visszatérésre készteti az aktuális függvényt és egyúttal továbbítja a feltüntetett értéket – ha van – a hívó rutinnak. Az alábbi két formában használható:

return;
return *érték;*

C++-ban a **return** előbbi alakját (amelyikben nem adunk meg visszatérési értéket) csak abban az esetben használhatjuk, ha a függvény **void**-ként volt deklarálva.

A következő függvény két integer argumentumának szorzatával tér vissza:

```
int szor(int a, int b)
{
  return a*b;
}
```

Tartsuk mindig szem előtt, hogy minden esetben amikor a vezérlés egy **return**-t talál, a kurrens függvény végrehajtása befejeződik (visszatérünk a hívást végző blokkba), és a **return**-t követő függvényrész már nem hajtódik végre.

Egy függvény több **return** utasítást is tartalmazhat.

■ *short*

A **short** egy adattípus-módosító, segítségével rövid egészeket deklarálhatunk.

■ *signed*

A **signed** típusmódosító leglényegesebb szerephez a **signed char** típus nevében jut. Használata integer típus esetén fölösleges, hiszen alapértelmezésben az integer előjeles egészeket tartalmaz.

■ *sizeof*

A **sizeof** kulcsszó minősége ellenére egyben fordítási idejű operátor is. Feladata az operandusként megadott változó vagy adattípus (beleértve az osztályokat, struktúrákat és unionokat is) méretének meghatározása. Ha típusra alkalmazzuk, akkor a típusazonosítót zárójelek közé kell tennünk. Változóval használva a zárójelek opcionálisak.

A legtöbb 32-bites C++ fordítóval lefordítva, az alábbi kód 4-et ír ki.

```
int i;
cout << sizeof(int);
cout << sizeof i;
```

■ *static*

A **static** adattípus-módosító utasítja a fordítót, hogy az utána álló lokális változó élettartamát a program futásának befejeztéig biztosítsa. Így a változót nem kell minden alkalommal létre hozni, amikor a végrehajtás az érvényességi körén belülre kerül és megszüntetni, amikor kilép onnan. Ezért egy változó statikussá tétele azt is jelenti, hogy értéke megőrződik két függvényhívás között.

A **static** módosító globális változókra is alkalmazható. Ez a változó érvényességi körét arra az állományra szűkíti le, amelyben deklaráltuk, azaz kizárjuk a változó exportálásának lehetőségét.

C++-ban, ha a **static**-ot egy osztály adattagjára alkalmazzuk, akkor az közössé válik az osztály minden példánya számára, azaz a tagot nem az egyes objektumokhoz kötjük, hanem magához az osztályhoz.

■ *static_cast*

A **static_cast** operátorral nempolimorfikus konverziókat végezhetünk. Például általa megtehetjük, hogy egy alaposztály típusú mutatót (az abból) leszármaztatott osztály típusú mutatóvá alakítsunk. Bármely standard konverzió is végrehajtható vele. Nincsen futásidejű ellenőrzés. Általános alakja:

static_cast<*típus*> (*objektum*)

■ *struct*

Struktúrákat a **struct** kulcsszó segítségével hozhatunk létre. C++-ban a struktúrák tartalmazhatnak adat- és függvénytagot egyaránt. A C++-ban az egyedüli különbség a **struct** és a **class** között, hogy alapértelmezésben

a struktúra minden tagja nyilvános. Ahhoz, hogy egy tagot priváttá tegyünk, a **private** kulcsszót kell használnunk. A C++-ban egy struktúra deklaráció általános formája így néz ki:

```
struct strukt_név : öröklődési_lista {
  //nyilvános tagok, ez az alapértelmezés, nem kell külön megjelölni
protected:
  //privát tagok, amelyek azonban öröklődhetnek
private:
  //privát tagok
} objektumlista;
```

ahol a *strukt_név* a struktúrához rendelt típusazonosító, ami egyben egy osztályt is definiál a C++-ban. Az egyes tagokat a pont operátorral érhetjük el, ha a struktúrával dolgozunk, illetve a nyíl operátort használjuk, ha a struktúrára mutató pointerrel rendelkezünk. Az *objektumlista* és az *öröklődési_lista* opcionális.

A C-ben számos megszorítás vonatkozik a struktúrákra. Először is csak adattagokat tartalmazhatnak, azaz függvénytagok nem megengedettek. C-beli struktúrák nem támogatják az öröklődést. Továbbá minden tag nyilvános (**public**), és a **protected**, illetve a **private** kulcsszavak nem használhatók.

Az alábbi C-stílusú struktúra egy stringet (**name**) és két integert fog össze (**high** és **low**), valamint deklarál egy **my_var** nevű változót.

```
struct my_struct {
  char name[80];
  int high;
  int low;
} my_var;
```

Megjegyzés
A struktúrákat átfogóan az első fejezet tárgyalja.

■ *switch*

A **switch** utasítás a C/C++ többirányú programelágaztatás egyik eszköze. A vezérlést irányítja a megfelelő helyre a program állapotától függően. Általános alakja:

```
switch (kifejezés) {
   case konstans1: utasítássorozat1;
      break;
   case konstans2: utasítássorozat2;
      break;
   .
   .
   .
   case konstansN: utasítássorozatN;
     break;
   default: egyéb utasítássorozat;
}
```

Minden egyes utasításblokk egy vagy több utasítást tartalmazhat. A **default**-ág megadása nem kötelező. Mind a *kifejezés*, mind a **case** konstansok egésztípusúak kell hogy legyenek.

A **switch** utasításban a *kifejezés* összehasonlításra kerül a konstansokkal (*konstans1*, stb.). Ha egyezés van, akkor az abban az ágban található utasítássorozat végrehajtódik. Amennyiben a utasításblokkot nem követi **break** utasítás, akkor a következő **case**-ágon folytatódik a végrehajtás. Másképpen fogalmazva, az egyezés helyétől a végrehajtás egészen addig folytatódik, amíg vagy egy **break** utasításhoz, vagy a **switch** utasítás végéhez nem érünk. Abban az esetben, ha nincs egyezés és van **default** sor, akkor az ehhez tartozó utasítások (*egyéb utasítássorozat*) kerülnek végrehajtásra. Egyébként nem történik semmi. Az alábbi példa egy menüpontválasztást modellez:

```
switch(ch) {
  case 'b': bevitel();
    break;
  case 'k': kilepes();
```

```
    break;
  case 'l': lista();
    break;
  case 'r': rendezes();
    break;
  default:
    cout << "Ilyen parancs nem létezik!\n";
    cout << "Próbálja újra.\n";
}
```

■ *template*

A **template** kulcsszó genericfüggvények és osztályok (makrószerű sablon függvények és osztályok) készítésére szolgál. Annak az adattípusnak a nevét, amellyel egy generic függvény dolgozik, paraméterként adjuk át. Ez teszi lehetővé, hogy adott függvény- vagy osztálydefiníció különböző típusú adatokkal is használható legyen. Most részletesen foglalkozunk ezekkel a függvény- és osztálysablonokkal.

A genericfüggvények általános jellegű funkciókat valósítanak meg, amelyek több adattípusra alkalmazhatók. A generic függvény paraméterként kapja meg annak az adattípusnak a nevét, amelyen majd dolgozni fog. Ennek a mechanizmusnak a lényege, hogy ugyanaz az általános eljárás merőben eltérő adatokra is alkalmazhatóvá válik. Tény, hogy sok olyan eljárás van, amelyek logikailag ugyanazt az algoritmust takarják, csak éppen más adattípusokat használnak. Vegyük például a jól ismert Quicksort rendező algoritmust. Működésének elvén mit sem változtat, ha nem egy integer tömbre, hanem egy lebegőpontos számokból álló tömbre alkalmazzuk. Csupán a rendezendő adatok típusa különbözik. Genericfüggvények használatával adattípusoktól függetlenül valósíthatjuk meg az algoritmusok lényegét. Amikor egy generic-re való hivatkozásban paraméterként átadjuk a manipulálandó adatok típusát, a fordító automatikusan a típusnak megfelelő kódot generál, a tényleges végrehajtásnál ez fut le. Egy genericfüggvény készítése olyan függvény létrehozását jelenti, amely automatikusan képes túlterhelni önmagát.

A **template** kulcsszó használatának általános alakja a következő:

template<class *adattípus*> *visszatérésitípus* (*paraméterlista*)
{
  // függvénytörzs
}

ahol az *adattípus* az a típus, amellyel a függvény valójában dolgozni fog.

Az az általános eljárás, amely két adatot kicserél egymással, független a változók típusától, ezért jó eséllyel pályázhat egy genericfüggvénnyel történő megvalósításra. Az alábbi példaprogramban ezt tettük meg: egy olyan genericfüggvényt készítettünk, amely a két argumentumként átadott változó értékét cseréli fel.

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

// generic függvény példa
template <class X> void csere (X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}

int main ()
{
  int i=10, j=20;
  float x=10.1, y=23.3;

cout <<"Eredeti i, j: " << i << ' '<< j
  <<endl;
cout <<"Eredeti x, y: " << x << ' ' << y
  <<endl;

csere(i, j);// egészek cseréje
csere(x, y);// lebegőpontosak cseréje
```

```
cout <<"Megcserélt i, j: " << i << ' '<< j
  <<endl;
cout <<"Megcserélt x, y: " << x << ' ' << y
  <<endl;

return 0;
}
```

Ebben a programban a

```
template <class X> void csere(X &a, X &b)
```

sor két dolgot mond a fordítónak: először, hogy egy genericfüggvényről van szó, másodszor, hogy az **X** egy generic típus, amely majd egy tényleges típussal helyettesítendő. A **csere** törzse a feloldandó **X** adattípusú változók segítségével van elkészítve (ezeket cseréljük meg). A **main( )**-ben a **csere( )** függvényt két különböző típusú adatpárral hívjuk meg: egésszel és lebegőpontossal. Mivel a **csere( )** egy genericfüggvény, a fordító automatikusan két változatot készít belőle – egy olyat, amelyik integer értékeket cserél ki, és egy másikat, amely lebegőpontosakat.

Több generictípust is megadhatunk, amennyiben a **template** utasítást egy vesszővel elválasztott listával használjuk.

A genericfüggvények hasonlítanak a túlterhelt függvényekhez, de több megszorítást vonnak maguk után. Túlterhelt függvények esetén a különböző típusokhoz különböző függvénytörzsek tartozhatnak. Egy genericfüggvény – mivel a függvénytörzs lényegében nem változik – minden változatában ugyanazt a műveletsorozatot végzi.

A genericfüggvények mellett lehetőségünk van *genericosztályok* definiálására is. Egy genericosztály definíció nem más, mint egy olyan osztály létrehozása, amelyben az összes szükséges (osztályra jellemző) algoritmust definiálunk, de a tulajdonképpeni típusa a manipulálandó adatoknak majd paraméterként adódik át, az osztály példányainak létrehozásakor.

A genericosztályoknak érezhető hasznuk van olyan esetekben, amikor általánosítható logika áll mögöttük. Például ha készítünk egy olyan algoritmusgyűjteményt, amellyel sorokat tudunk karbantartani, nem árt, ha az például nem csak integer típusú elemek esetén működik, hanem karakterelemekre is. Hasonlóképpen elvárható, hogy egy olyan mechaniz-

mus, amely egy postai címeket tartalmazó csatolt listát kezel, kezelni tudjon autóalkatrészek neveiből álló csatolt listát is és bármilyen egyéb típus használata se okozzon nehézségeket. A fordító automatikusan a helyes típusú objektumot generálja, amikor a konkrét objektumokat létrehozzuk, paraméterként átadva a használandó típust. A generic osztálydeklarációk általános formája a következő:

```
template <class adattípus> class osztálynév {
  //...
};
```

ahol az *adattípus* az a típus, amellyel az osztály valójában dolgozni fog. Amikor egy genericosztály egy példányát (objektum) deklaráljuk, a típust sarkos zárójelek között kell feltüntetnünk. Egy ilyen deklaráció általános alakja a következő:

osztálynév <típus> objektum;

Az alábbi példa a genericosztályok használatát illusztrálja. A program egy egyszeresen láncolt listát megvalósító genericosztályt hoz létre, amit azután karaktertípussal példányosítunk.

```
//Generic csatolt-lista
#include <iostream>
using namespace std;

template <class data_t> class list {
  data_t data;
  list *next;
public:
  list (data_t d);// konstruktor deklaráció
  void add (list *node) { node->next = this;
                          next=0; }
  list *getnext() { return next; }
  data_t getdata() { return data; }
}
```

```
template <class data_t>
list <data_t>::list(data_t d) // a konstruktor
                              //definíciója
{
  data = d;
  next = 0;
}

int main ()
{
  list <char> start('a');
  list <char> *p, *last;
  int i;

  //lista készítés
  last = &start;
  for(i=0; i<26; i++) {
  p = new list <char> ('a'+i);
  p -> add(last);
  last = p;
  }

  // a lista elemeinek kiíratása
  p = &start;
  while (p) {
  cout << p->getdata();
  p = p-> getnext();
  }

return(0);
}
```

Látható, hogy egy genericosztály deklarációja hasonlít egy genericfüggvényéhez. A listában tárolt adat típusától függetlenítettük az osztályt. A **main( )**-ben elkészített objektumokkal és pointerekkel megadjuk a hiányzó típust, ami jelen esetben a **char**. Hasonló módon más típusú listák is készíthetők.

Érdemes külön figyelmet fordítanunk a

```
list<char> start('a');
```

deklarációra. Figyeljük meg, hogy a kívánt típust sarkos zárójelek között tüntetjük fel.

■ *this*

A **this** egy olyan pointer, amely arra az objektumra mutat, amelyik az aktuális függvénytagot hívta. Minden függvénytagnak automatikusan átadódik a **this** mutató.

■ *throw*

A **throw** egy fontos eleme a C++ kivételkezelő rendszerének, amelyet az alábbi néhány bekezdésben részletesen ismertetünk.

A kivételkezelés három kulcsszóra épül: a **try**-ra, **catch**-re és a **throw**-ra. Általánosságban elmondható, hogy egy **try** blokkba azt a programrészt ágyazzuk be, amelyben felmerülő hibákat (kivételeket) strukturált módon kívánjuk lekezelni. Ha egy kivétel (azaz egy hiba) történik a **try** blokkon belül, akkor azt mondjuk, hogy a függvény, amelyben a hiba történt „eldobja" (az angol throw szóból) a kivételt, a tevékenységet a *kivétel dobásának* nevezik. A kivételt a **catch** segítségével kaphatjuk el és dolgozhatjuk fel. Az alábbiakban ezt a folyamatot közelebbről vizsgáljuk meg.

Miként azt már említettük, minden olyan utasításnak, amely egy kivételt eldob, a **try** blokkon belül kell szerepelnie. (Azok a függvények, amelyeket **try** blokkból hívtak meg, szintén dobhatnak kivételeket.) A kivételeket a kivételt dobó **try** blokkot közvetlenül követő **catch** utasítással kapjuk el. A **catch** és a **try** az alábbi módon működik együtt:

```
try{
   // try blokk
}
```

```
catch(típus1 arg){
  // catch blokk
}
catch(típus2 arg){
  // catch blokk
}
// . . .
catch(típusN arg){
  // catch blokk
}
```

A **try** blokkoknak kell tartalmaznia a program azon részeit, amelyeken hibakezelést kívánunk végezni. Ez jelentheti a függvény egy-két sorát vagy akár az egész **main( )** függvénykódot is **try** blokkba zárhatjuk, ezáltal az egész programot bevonva a kivételkezelésbe.

Egy eldobott kivétel elkapása a hozzátartozó **catch** utasítás feladata, amely egyben fel is dolgozza azt. Egy **try**-hoz több **catch** utasítás is tartozhat. A kivétel típusa határozza meg, hogy melyik **catch** utasítás kerül végrehajtásra. Az a **catch**-blokk hajtódik végre, amelyben a megjelölt típus megegyezik a dobott kivétel típusával (és a többit a vezérlés átugorja). A kivétel elkapásakor az *arg* (a catch argumentuma, l. feljebb) megkapja az értékét. Mindenféle típusú adat elkapható, beleértve a programozó által definiált osztályokat is. Ha nincsen dobott kivétel, akkor egyik **catch**-ág sem hajtódik végre.

A **throw** utasítás általános alakja így néz ki:

```
throw kivétel;
```

A **throw**-t vagy magában a **try** blokkban hajtjuk végre, vagy – közvetett módon – bármilyen a **try** blokkból meghívott függvény belsejében. A *kivétel* a dobott kivétel neve.

Ha egy olyan kivételt dobunk, amelyhez nincsen megfelelő **catch**-ág, akkor a program abnormális leállásával kell számolnunk. Egy lekezeletlen kivétel automatikusan meghívja a **terminate( )** függvényt. Alapértelmezésben a **terminate( )** az **abort( )** függvényt meghívja, ami megállítja a program futását. Lehetőség van az **abort** helyett általunk megadott függvényt meghívatni a **terminate( )**-tel, ehhez a **set_terminate** függvényt kell használnunk.

Az alábbi egyszerű példával illusztráljuk a kivételkezelés működését a C++-ban:

```
//Egyszerű kivételkezelő példa
#include <iostream>
using namespace std;

int main()
{
  cout << "Kezdődik.\n";

  try { // a try blokk kezdete
    << "Bekerültünk a try blokkba.\n";
    throw 100; // hiba dobása
    cout << "Ez az üzenet nem jelenik meg.\n";
  }
  catch (int i) {// hiba elfogása
    cout << "Elfogtam egy hibát - - azonosítója:";
    cout << i << "\n";
  }

  cout << "Vége.";

  return 0;
}
```

A program az alábbi üzeneteket írja a kimenetre:

```
Kezdődik.
Bekerültünk a try blokkba.
Elfogtam egy hibát - - azonosítója: 100
Vége.
```

A példában szereplő **try** blokk három utasításból áll, a blokkot egy **catch(int i)** utasítás követ, amely integer kivételeket dolgoz fel. A **try** blokkban a három utasításból csak kettő hajtódik végre: az első **cout** és **throw** utasítás. A kivétel fellépése után a vezérlés átadódik a **catch**-ág-

nak, miközben a **try** blokkból végérvényesen kilépünk. A **catch** nem meghívódik, hanem a program végrehajtás ugrik oda. (És a program verem is ennek megfelelő állapotba kerül.) Így, érthető módon, a **throw**-t követő **cout** utasítás már nem hajtódik végre.

■ *true*

A **true** logikai konstans a logikai igen értéket jelöli.

■ *try*

A **try** kulcsszó a C++ kivételkezelő mechanizmusának egy alapszava. Lásd a „**throw**" címszó alatt.

■ *typedef*

A **typedef** kulcsszó lehetővé teszi, hogy egy már létező adattípushoz új nevet rendeljünk. Az adattípus lehet a predefinit típusok egyike, vagy valamely osztály, struktúra, union, illetve enumeráció. A **typedef** általános alakja a következő:

typedef *típusspecifikátor új_név*;

Például, ha valamely programban a **hossz** szót kívánjuk használni a **float** helyett, akkor a

```
typedef float hossz;
```

sort kellene beépíteni a programba.

■ *typeid*

C++-ban a **typeid** operátor egy **type_info** típusú objektum címét adja vissza, amely az operandusaként szereplő objektum típusáról ad leírást. A **typeid** általános alakban:

typeid (*object*)

A **typeid** operátor a C++-ban lehetővé teszi a futásidejű típusazonosítást (RTTI, az angol Runtime Type Identification kifejezésből).

Megjegyzés
Lásd még a 3. fejezet idevonatkozó részét.

■ *typename*

A **typename** a C++ által támogatott kulcsszó. A **class** kulcsszót helyettesítheti **template** deklarációkban vagy jelezhet egy nem definiált típust.

■ *union*

Az *union* egy olyan osztálytípus, amelyben minden adattag egyazon memóriaszeleten osztozik. Definiálása és a rá való hivatkozás módja – a . operátor és a –> operátor használata – lényegében megegyezik az osztályoknál látottakkal. Alapértelmezésben a tagjai privát láthatóságúak. Általános alakban:

```
union osztálynév {
   //alapértelmezés szerint nyilvános tagok
private:
   //privát tagok
} objektumlista;
```

ahol az *osztálynév* az uniontípus neve.

Megjegyzés
C-ben az unionok kizárólag adattagokat foghatnak össze és a ***private*** *kulcsszó használata nem megengedett.*

Az alábbi példában egy karakterlánc és egy **double** típus unionját definiáltuk és egyúttal létrehoztunk egy ilyen típusú változót is (**my_var**).

```
union my_union {
  char time[30];
  double offset;
} my_var;
```

Az unionról bővebb információt az első fejezet tartalmaz.

■ *unsigned*

Az **unsigned** egy olyan adattípus-módosító, amellyel előjel nélküli egészeket deklarálhatunk, amelyek csak pozitív értéket vehetnek fel.

■ *using*

Lásd **namespace**.

■ *virtual*

A **virtual** függvényspecifikátorral virtuális függvényeket hozhatunk létre. Azok a függvények, amelyeket alaposztályban virtuálisként definiálunk, a származtatott osztályban felüldefiniálhatók. Ha származtatott osztályban a virtuális függvényt nem definiáljuk fölül, akkor továbbra is az alaposztálybeli definíció marad érvényben.

Tisztán virtuális függvénynek nevezzük azokat a függvényeket, amelyeknek nincsen definíciós részük. Ez azt jelenti, hogy a *tisztán virtuális függvényeket* mindenképpen felül kell definiálnunk a származtatott osztályban. Egy *tisztán virtuális függvény* prototípusát így adjuk meg:

virtual *vtípus fnév* (*paraméterlista*) = 0;

ahol a *vtípus* a függvény visszatérési típusát jelöli, *fnév* a függvény neve és a *paraméterlista* a paramétereket tartalmazza. Az = **0** rész jelenti az igazi különbséget a sima prototípustól. Ez jelzi a fordítónak, hogy a virtuális függvénynek nincsen az alaposztályban definíciója.

Futásidejű polimorfizmusról beszélünk, amikor virtuális függvényekre alaposztály típusú mutatókon keresztül történik hivatkozás. Egy ilyen szituációban a mutatott objektum (dinamikus) típusa határozza meg, hogy a virtuális függvény melyik változata kerül meghívásra.

Programozási tipp

Azokat az osztályokat, amelyek legalább egy tisztán virtuális függvénytaggal rendelkeznek, absztrakt osztályoknak hívjuk. Absztrakt osztályokból nem hozhatunk létre példányokat. Továbbá nem használhatók függvények paramétertípusaiként vagy visszatérési típusként. Absztrakt osztályra mutató pointerek létrehozása azonban megengedett.
Azaz absztrakt osztályból származtatott osztály, amely nem definiálja felül az ősében található összes tisztán virtuális függvényt, maga is absztrakt lesz. Más szóval, ha konkrét, példányosítható osztályt kívánunk származtatni egy absztrakt osztályból, felül kell definiálnunk minden egyes tisztán virtuális függvényt.

■ *void*

A **void** típusspecifikátor elsődlegesen arra szolgál, hogy explicit módon tudjunk olyan függvényeket deklarálni, amelyeknek nincsen visszatérési értékük. Továbbá segítségével típus nélküli mutatókat hozhatunk létre. A típus nélküli mutatóknak az a tulajdonsága, hogy bármilyen típusú objektumra mutathatnak.

C-ben a **void** kulcsszóval jelezzük, ha az adott függvénynek nincsenek paraméterei.

■ *volatile*

A **volatile** minősítő azt jelzi a fordítónak, hogy a mögötte szereplő változó értéke úgy is megváltozhat futás közben, hogy explicit módon arra

nem ad utasítást egyetlen programsor sem. Ilyen lehet például, amikor egy változó értéke az operációs rendszer órarutinja vagy valamely más megszakítás által változik meg.

■ *wchar_t*

A **wchar_t** típusspecifikátorral széles karaktereket deklarálhatunk. A széles karakterek 16-bit hosszúak.

■ *while*

A **while** ciklus általános alakja így néz ki:

```
while (feltétel) {
   utasításblokk
}
```

Ha a ciklusmagot egyetlen utasítás alkotja, akkor a kapcsos zárójelek elhagyhatók.

A **while** a ciklusfeltétel ellenőrzését a ciklus elején végzi. Ezért ha a *feltétel* a ciklusba való belépéskor hamis, a ciklusmag egyszer sem hajtódik végre. A *feltétel* bármilyen logikai kifejezés lehet.

Az alábbi példa a **while** ciklus működését illusztrálja. A program beolvas 100 karaktert és ezeket eltárolja egy karaktertömbben.

```
char s[256];

t = 0;
while(t<100) {
  s[t] = stream.get();
  t++;
}
```

6. FEJEZET

A standard C be- és kiviteli függvényei

Ez a fejezet a szabványos C be- és kiviteli (I/O) függvényeit tárgyalja. Ezeket a függvényeket az ANSI C szabvány definiálja, így minden fordító kötelezően tartalmazza ezeket standard C könyvtáraiban. A kompatibilitás biztosítása miatt a C++ is támogatja használatukat, és nincs okunk rá, hogy ezen függvények létjogosultságát a C++-ban megkérdőjelezzük. Mivel a fejezetben szereplő függvények az ANSI C szabványt követik, közös néven az ANSI C I/O *rendszerként* hivatkozunk rájuk.

A standard I/O függvények fejét tartalmazó állományt STDIO.H néven mellékelik a fordítóhoz. Ez számos olyan makrót és típust definiál, amelyek a fájlrendszer kezeléséhez szükségesek. Ezek közül is kulcsfontosságú a **FILE** típus, amellyel logikai állománymutatót deklarálhatunk. A **size_t** és **fpos_t** két másik lényeges típus, amelyek leginkább az előjel nélküli integerhez hasonlíthatók. A **size_t** olyan típus, amely képes az operációs rendszer által megengedett legnagyobb állomány méretének tárolására. Az állományokban szereplő egyes adatelemek pozíciójának egyértelmű megadására szolgálnak az **ftpos** típusú objektumok.

Az ANSI C I/O rendszer ún. csatornákra vagy adatfolyamokra (használjuk a stream elnevezést is) épül. Egy csatorna egy olyan logikai eszköz, amely a tulajdonképpeni fizikai eszközzel áll összeköttetésben. Az utóbbiakat hívjuk *fájloknak*.

Az ANSI C I/O rendszerben minden csatorna egyforma jellemzőkkel bír, nem úgy, mint a fájlok, amelyek sokfélék lehetnek. Például egy diszken lévő fájl a közvetlen hozzáférést is lehetővé teszi, míg a modem nem. Az ANSI C I/O rendszer által támogatott adatfolyam-kezelés egy absztrakciós szintet vezet be a programozó és a valódi fizikai eszköz közé. Az absztrakció az adatfolyam, az elrejtett eszköz a fájl. Ilyen módon kifelé egy egységes logikai felület látszik, míg a tulajdonképpeni fizikai eszközök különbözhetnek.

A csatornát a fájlokkal az **fopen( )** függvény hívásán keresztül kapcsoljuk össze. Az adatfolyamokkal állománymutatók (***FILE** típusú mutatók) segítségével dolgozhatunk.

Amikor egy program végrehajtása elkezdődik, három predefinit csatorna automatikusan megnyílik: a **stdin**, **stdout** és az **stderr**, ezek sorra a standard inputra (szabványos bemenet), a standard outputra (szabványos kimenet) és a standard error-ra (szabványos hibakimenet) utalnak. Alapértelmezésben ezek a konzolhoz kapcsolódnak, de átirányíthatók bármely más eszközre.

A standard könyvtári függvények nagy része hiba esetén átállítja az **errno** globális változó értékét. A programmal futás közben megvizsgálhatjuk ennek a változónak a tartalmát, ami által információhoz juthatunk a hibáról. Az **errno** értékei és a hibák közti megfeleltetés implementációfüggő.

■ *clearerr*

```
#include <stdio.h>
void clearerr(FILE *stream);
```

A **clearerr( )** függvény kinullázza a *stream* által mutatott csatornához tartozó hiba jelzőbitet. A fájlvége indikátor is visszakapja alapértékét.

Minden csatornához rendelt hiba jelzőbit értéke – egy sikeres **fopen( )** hívás után – kezdetben nulla. Hiba esetén a jelzőbit értéke megváltozik, az eredeti állapotot egy **clearerr( )** vagy pedig egy **rewind( )** függvényhívással állíthatjuk vissza.

Fájlhibák előfordulásának számos különböző oka lehet, amelyek közül sok rendszer függő. A hiba természetének pontos megállapításában a **perror( )** függvény segít, amely kiírja, hogy milyen hiba történt (l. „**perror( )**”).

Vonatkozó függvények: **feof( )**, **ferror( )**, **perror( )**.

■ *fclose*

```
#include <stdio.h>
int fclose(FILE *stream);
```

Az **fclose( )** függvény bezárja a *stream*-nek megfelelő fájlt és kiüríti a pufferjét. Az **fclose( )** kiadásával a *stream* adatfolyam és a fájl kapcsolatát megszüntetjük és valamennyi automatikusan lefoglalt puffer felszabadul.

Ha az **fclose( )** sikeres volt, 0-t kapunk vissza, egyébként **EOF**-fal tér vissza a függvény. Egy már bezárt fájl bezárására irányuló kísérlet természetesen hibához vezet. Ha a tároló eszköz nem elérhető a fájl bezárásának pillanatában, vagy nincs elég szabad tárolási kapacitás, úgyszintén hibát kapunk.

Vonatkozó függvények: **fopen( )**, **freopen( )** és **fflush( )**.

■ *feof*

```
#include <stdio.h>
int feof(FILE *stream);
```

A **feof( )** függvény ellenőrzi a fájlpozíció-indikátort és megállapítja, hogy a *stream*-hez tartozó fájl végét elértük-e. Nullától különböző visszatérési értéket kapunk, ha a fájlpozíció indikátor „fájlvége" jelnél van, egyébként a **feof** nullával tér vissza.

A fájlvége elérését követően minden további olvasási kísérlet **EOF**-t (**end-of-file** =fájlvége jel) eredményez, amíg meg nem hívjuk a **rewind( )** függvényt vagy a fájlpozíció indikátort el nem mozgatja az **fseek( )**. Az **EOF** makró az STDIO.H-ban van definiálva.

Az **feof( )** függvény hasznossága hangsúlyozottan érezhető, amikor bináris fájlokkal dolgozunk, hiszen ekkor a fájlvége jel egyben egy érvényes bináris jel is. Ilyen esetekben nyilvánvalóan az **feof( )** függvény hívása a célszerű és nem a **getc( )** visszatérési értékének az ellenőrzése.

Vonatkozó függvények: **clearerr( )**, **ferror( )**, **perror( )**, **putc( )**, **getc( )**.

■ *ferror*

```
#include <stdio.h>
int ferror(FILE *stream);
```

A **ferror( )** függvény a megadott csatornához tartozó fájlhibákat jelzi. A nulla visszatérési érték jelzi, hogy nem történt hiba, míg a nullától különböző értékek hibát jelentenek.

A *stream*-hez tartozó hibabitek addig nem állnak vissza normális állapotukba, amíg a fájlt be nem zárjuk, vagy egy **rewind( )**, illetve **clearerr( )** függvényhívást nem végzünk.

A hiba természetének kiderítésére használjuk a **perror ( )** függvényt.

Vonatkozó függvények: **clearerr( )**, **feof( )** és **perror( )**.

■ *fflush*

```
#include <stdio.h>
int fflush(FILE *stream);
```

Ha a fájlhoz asszociált *stream*-et írásra nyitottuk meg, akkor egy **fflush( )** hívás azt eredményezi, hogy kimeneti puffer tartalma fizikailag is bekerül fájlba. Ha a *stream* állománymutató egy input fájlra mutat, akkor az **fflush( )**-sal az inputpuffert üríthetjük ki. Mindkét esetben a fájl nyitva marad.

A visszatérésként kapott 0 azt jelenti, hogy minden zökkenőmentesen ment, **EOF** egy írás közben esett hibára utal.

A program normális befejezésekor minden puffer kiürül. Egy betelt puffer is automatikusan kiürül. Egy adott fájl bezárásakor a hozzátartozó puffer felszabadul.

Vonatkozó függvények: **fclose( )**, **fopen( )**, **fread( )**, **fwrite( )**, **getc( )** és **putc( )**.

■ *fgetc*

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Az **fgetc( )** függvény a *stream* bemeneti csatorna következő karakterét adja vissza, és növeli a fájlpozícióindikátort. A karaktert **unsigned char** típusúként olvassa a függvény, amelyet aztán egésszé konvertál.

Ha a fájl végéhez értünk, akkor az **fgetc( )** **EOF**-fal tér vissza. Mivel az **EOF** egy érvényes integer érték, bináris fájlok esetében az **feof( )**-t használjuk a fájlvége észlelésére. Ha az **fgetc( )** működése során hiba történik, akkor is **EOF** kerül visszaadásra. Bináris fájlok esetén az **ferror( )**-ral érhetők tetten a fájlhibák.

Vonatkozó függvények: **fputc( )**, **getc( )**, **putc( )**, **fopen( )**.

■ *fgetpos*

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *position)
```

Az **fgetpos( )** függvény a fájlpozíció-indikátor aktuális értékét abba az objektumba másolja, amelyre a *position* nevű pointer mutat. A *position* által mutatott objektum típusa az STDIO.H-ban definiált **ftpos_t** kell, hogy legyen. Az így eltárolt értékre leggyakrabban az **fsetpos( )** többszöri használata után lehet szükségünk.

Hiba esetén **fgetpos( )** nullától különböző értékkel tér vissza, normális működéskor nem nulla értéket kapunk.

Vonatkozó függvények: **fsetpos( )**, **fseek( )**, és **ftell( )**.

■ *fgets*

```
#include <stdio.h>
char *fgets(char *str, int num, FILE *stream);
```

Az **fgets( )** függvény *num*-1 karakternyit beolvas a *stream* adatfolyamból és azokat a *str* pointer által mutatott karaktertömbbe helyezi. A karakterek olvasásának, a megadott határ elérésén kívül a sorvége, vagy az **EOF** is véget vethet. Az utolsó karakter beolvasása és a tömbbe helyezése után egy 0 is kerül a tömb végére. Az újsor karakter – ha ilyen volt a beolvasottak végén (ez megszakítja a tovább olvasást) még bemásolódik a **str*-be.

Sikeres végrehajtás esetén **fgets( )** a *str*-rel tér vissza, máskülönben nullpointert kapunk. Amennyiben olvasási hiba történik, a *str* címén lévő

tömb tartalma határozatlan lesz. Mivel hiba esetén is és fájlvége jel olvasásakor is nullpointert kapunk vissza, annak kiderítésére, hogy pontosan melyik történt a **feof( )** vagy a **ferror( )** használata szükséges.

Vonatkozó függvények: **fputs( )**, **fgets( )**, **gets( )**, **puts ( )**.

■ *fopen*

```
#include <stdio.h>
FILE *fopen(const char *fname, const char *mode);
```

Az **fopen( )** függvény az **fname* által meghatározott fájlt nyitja meg és rendeli hozzá egy csatornához. Visszatérési értékként az adatfolyamot kapjuk meg. A megengedett műveleteket a *mode* értéke határozza meg. Ennek elképzelhető értékeit az alábbi táblázatban soroltuk fel. A megadott fájlnév meg kell hogy feleljen az operációs rendszer által előírt fájlnév-képzési szabályoknak, ha a környezet támogatja, teljes elérési útvonalat is megadhatunk.

Mód	*Jelentés*
„r"	Szövegfájl megnyitása olvasásra
„w"	Szövegfájlt hoz létre írásra
„a"	Hozzáfűzés szövegfájlhoz
„rb"	Bináris fájl megnyitása olvasásra
„wb"	Bináris fájl létrehozása írásra
„ab"	Hozzáfűzés bináris fájlhoz
„r+"	Szövegfájl megnyitása írásra és olvasásra
„w+"	Szövegfájl létrehozása írásra és olvasásra
„a+"	Szövegfájl megnyitása írásra és olvasásra
„rb+"	Bináris fájl megnyitása írásra és olvasásra
„wb+"	Bináris fájl létrehozása írásra és olvasásra
„ab+"	Bináris fájl megnyitása írásra és olvasásra

Ha az **fopen( )**-nel sikeresen megnyitottuk a megadott fájlt, akkor egy **FILE** mutatót kapunk vissza. Ha a fájl valamilyen oknál fogva nem nyitható meg, a függvény nullpointerrel tér vissza.

Látható, hogy a fájlokat kétféleképpen nyithatjuk meg: bináris és szöveges módban. Szöveges módban számíthatunk néhány karaktertransz-

formációra is: pl. új sorok kocsivissza/soremelés sorozatokra konvertálódhatnak. Bináris fájloknál ilyen nem fordulhat elő.

A helyes fájlmegnyitást az alábbi kódrészlet illusztrálja:

```
FILE *fp;

if ((fp = fopen("teszt", "w"))==NULL) {
  printf("Nem tudom megnyitni a fájlt.\n");
  exit(1);
}
```

Ez a módszer arra is jó, hogy ellenőrizzük a megnyitás sikerességét, még mielőtt írni próbálnánk a fájlba: pl. egy írásvédett vagy megtelt lemez esetén üzenetet kapunk. A **NULL** segítségével mutathatjuk ki a hibát, mert a fájlpointerek ezt az értéket nem vehetik fel. A **NULL**-t a STDIO.H definiálja.

Amennyiben az **fopen( )** függvénnyel írásra nyitunk meg egy fájlt és volt már ilyen nevű fájlunk, akkor az elveszik és helyét egy új veszi át. Ha nem létezett előzőleg az adott nevű fájl, akkor most létre jön. Ha egy létező fájl végéhez szeretnénk hozzáírni, akkor az „a" módot kell használnunk. Amennyiben a megjelölt fájl nem létezik, hibát kapunk. Végül ha egy olyan fájlt nyitunk meg írásra és olvasásra, amelyik már létezik, akkor annak tartalma nem veszik el, ha még nem létezik az adott nevű fájl, akkor a megnyitáskor létrejön.

Írásra és olvasásra megnyitott fájlok esetén nem tehető meg, hogy egy output műveletet input művelettel követünk, anélkül, hogy közben ne hívjuk meg az **fflush( )**, **fseek( )**, **fsetpos( )** vagy az **frewind( )** függvények valamelyikét. Ez fordítva is igaz: egy input műveletet nem követhet output művelet, anélkül, hogy közben a fenti függvények egyikét ne alkalmaznánk.

Vonatkozó függvények: **fclose( )**, **fread( )**, **fwrite( )**, **putc( )**, **getc( )**.

Programozási tipp

Minden fájl megnyitható bináris és szövegfájlként is, függetlenül attól, hogy mit tartalmaz a fájl. Például egy ASCII szöveget tartalmazó fájl megnyitható és manipulálható bináris fájlként. Ami az ANSI C fájlkezelést illeti, az egyedüli különbség a két módozat között, hogy bináris fájlok esetén nem történhetnek automatikus karakterhelyettesítések.

Érdemes megnyitni szövegfájlokat binárisként, ha nem szövegalapú manipulációkat kívánunk végrehajtani rajtuk. Például azok az eszközök, amelyek tömörítéseket, összehasonlítást végeznek, vagy rendeznek, tipikusan olyanok, amelyek a fájlokat bináris hozzáférésre nyitják meg. Állományok titkosítására használt programok is majdnem mindig binárisként használják a fájlokat.
Egy szöveg és egy bináris fájl közti lényeges különbséget nem a tartalmuk jelenti, hanem a megnyitásukkor használt mód jelenti.

■ *fprintf*

```
#include <stdio.h>
int fprintf(FILE * stream, const char *format, ...);
```

Az **fprintf( )** függvény kiírja a *stream*-hez asszociált fájlba a paraméterlistában megadott értékeket a *format* string által meghatározott formátumban. A függvény a valójában kiírt karakterek számát adja vissza. Hiba esetén negatív számot kapunk vissza.

Az argumentumok száma 0-tól egy a rendszer által meghatározott értékig terjedhet. A formátumkezelő string ugyanúgy kezelendő, mint a **printf( )** függvényben. Részletes leírásához lapozzuk fel a „**printf( )**" fejezetrészt.

■ *fputc*

```
#include <stdio.h>
int fputc(int ch, FILE *stream);
```

Az **fputc( )** függvény az aktuális fájlpozícióra írja a *ch* karaktert, majd előrébb állítja a fájlpozícióindikátort. Annak ellenére, hogy a *ch*-t – történeti okok miatt – integerként deklaráltuk, az **fputc( )** ezt **unsigned char** típusúvá konvertálja. A karakterváltozó híváskor integerként adódik át, ezért argumentumként szabadon használhatunk karaktereket. Ha egész számokat használunk a felső bájtot a függvény egyszerűen nem veszi számításba.

Az **fputc( )** visszatérési értéke a kiírt karakter értéke, kivéve hiba felléptekor, amikor is **EOF**-fal tér vissza. Bináris műveletekre megnyitott fájlok esetén az **EOF** egy érvényes karakter, így annak eldöntésére, hogy tényleg hiba történt-e, a **ferror( )** függvényt használjuk.

Vonatkozó függvények: **fgetc( )**, **fopen( )**, **fprintf( )**, **fread( )**, **fwrite( )**.

■ *fputs*

```
#include <stdio.h>
int fputs(const char *str, FILE *stream);
```

Az **fputs( )** függvény a *str* pointer által mutatott string tartalmát írja ki a megadott adatfolyamba. Fontos: a string végét jelző nullkaraktert a függvény nem írja ki.

Az **fputs( )** sikeres működés után nem negatív értékkel tér vissza, hibát a visszaadott **EOF** jelzi.

Ha a csatornát szöveges módban nyitjuk meg, akkor bizonyos karakterhelyettesítések történhetnek. Ez azt jelenti, hogy nem biztosított a string egy az egyben történő bemásolása a fájlba. Binárisként megnyitott fájlok esetén ilyen nem fordulhat elő, a string és a fájl adott szakasza között garantált az egyezés.

Vonatkozó függvények: **fgets( )**, **gets( )**, **puts( )**, **fprintf( )**, **fscanf( )**.

■ *fread*

```
#include <stdio.h>
int fread(void *buf, size_t méret, size_t számláló, FILE
*stream);
```

A **fread( )** függvény *számláló* darab *méret* méretű objektumot olvas be a *stream* által mutatott adatfolyamból a *buf* pointer által meghatározott tömbbe. A fájlpozíció-indikátor a beolvasott karakterek számának megfelelően állítódik előrébb.

Az **fread( )** függvény a ténylegesen beolvasott egységek számát adja vissza. Ha ez kevesebb, mint a híváskor igényelt, akkor vagy hiba történt, vagy ideje-

korán elértük a fájlvégét. Ahhoz, hogy pontosan megtudjuk mondani melyik esettel állunk szemben, az **feof( )** vagy az **ferror( )** függvényt kell használnunk.

Ha a csatornát szöveges módban nyitjuk meg, előfordulhatnak bizonyos karakterhelyettesítések, mint pl. a kocsivissza/soremelés új sorokká transzformálása.

Vonatkozó függvények: **fwrite( )**, **fopen( )**, **fscanf( )**, **fgetc( )**, **getc( )**.

■ *freopen*

```
#include <stdio.h>
FILE *freopen(const char *fname, const char *mode, FILE
*stream);
```

A **freopen( )** függvény egy létező csatornát egy másik fájlhoz rendel. Az új fájl nevére a *fname* pointer mutat a hozzáférési módot megadó stringre a *mode* mutat, és az adatfolyamot, amelyhez a fájlt kívánjuk rendelni a *stream* mutatón keresztül adjuk meg. A *mode* string lehetséges értékei és azok hatásai ugyanazok, mint ahogy azt az **fopen( )**-nél láttuk.

Egy **freopen( )** híváskor az alábbiak történnek: először a *stream*-hez jelenleg rendelt fájlt próbálja bezárni. Másodszor megpróbálja megnyitni a másik fájlt, függetlenül az előző művelet esetleges sikertelenségétől.

A **freopen( )** függvény normális működéskor a *stream*-re mutató pointert, egyébként nullpointert ad vissza.

A **freopen( )** leggyakoribb alkalmazása a rendszer által definiált fájlok (**stdin**, **stdout** és **stderr**) átirányítása valamilyen más fájlba.

Vonatkozó függvények: **fopen( )**, **fclose( )**.

■ *fscanf*

```
#include <stdio.h>
int fsanf(FILE *stream, const char *format,...);
```

Az **fscanf( )** függvény lényegében úgy működik, mint a **scanf( )** függvény, az egyedüli különbség, hogy nem a **stdin**-t használja az adatok olvasására, hanem a *stream*-et. Részletesen l. a „**scanf( )**” fejezetrészt.

Az **fscanf( )** függvény azon argumentumok számát adja vissza, amelyek ténylegesen értéket kaptak a rutin végrehajtása során. Ez az érték nem tartalmazza a kihagyott mezőket. Az **EOF**-fal történő visszatérés azt jelzi, hogy az első értékadás előtt hiba történt.

Vonatkozó függvények: **scanf( )**, **fprintf( )**.

■ *fseek*

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int origin);
```

Az **fseek( )** függvény a fájlpozíció-indikátort állítja az *offset* és az *origin* paraméterekben megadott értékeknek megfelelően. Célja a közvetlen elérésű I/O műveletek biztosítása. Az *offset* azt mondja meg, hogy mennyit menjen a fájlpozíció-indikátor előre az *origin*-hoz képest. Az *origin* értékei az alábbi makrókból (az STDIO.H definiálja őket) kell hogy kikerüljenek.

Név	*Jelentés*
SEEK_SET	A fájl elejétől
SEEK_CUR	Jelenlegi helytől
SEEK_END	Fájlvégétől

A visszatéréskor kapott 0 érték a függvény hibátlan végrehajtására utal. A nullától különböző érték hibát jelez.

Az **fseek( )** segítségével a fájlpozíció-indikátort a fájl bármely részére állíthatjuk. Hibához vezet – természetesen –, ha a fájlpozíció-indikátort a fájl eleje elé akarnánk állítani.

Az **fseek( )** függvény törli az adott csatornához tartozó fájlvége jelzőbitet. Továbbá hatástalanítja az esetlegesen ugyanerre az adatfolyamra előzőleg kiadott **ungetc( )**-t (l. „**ungetc( )**").

Vonatkozó függvények: **ftell( )**, **rewind( )**, **fopen( )**, **fgetpos( )**, **fsetpos( )**.

■ *fsetpos*

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *position);
```

Az **fsetpos( )** függvény a fájlpozíció-indikátort a *stream*hez rendelt fájl azon pontjára állítja, amelyet a *position* által mutatott objektum értéke jelöl ki. Ez az érték egy korábbi **fgetpos( )** hívásból kell hogy származzon. Az **fpos_t** típust az STDIO.H definiálja. Minden **fsetpos( )** hívás után a fájlvége indikátor automatikusan beállítódik. Továbbá hatástalanítja az esetlegesen ugyanerre az adatfolyamra előzőleg kiadott **ungetc( )**-t.

Hiba fellépésekor az **fsetpos( )** nullától különböző értéket ad vissza. Sikeres működést a nulla visszatérési érték jelzi.

Vonatkozó függvények: **fgetpos( )**, **fseek( )**, **ftell( )**.

■ *ftell*

```
#include <stdio.h>
long ftell(FILE *stream);
```

Az **ftell( )** függvény a megadott csatornához tartozó fájlpozíció-indikátor aktuális értékét adja vissza. Bináris adatfolyamok esetén ez az érték azt mondja meg, hogy az aktuális pozíció hány bájtnyira található a fájl elejétől. Szöveges adatfolyamok esetén a visszakapott érték lehet, hogy nem elég informatív az esetleges karakterhelyettesítések miatt, leginkább az **fseek( )** argumentumaként szokás használni. Elég, ha arra gondolunk, hogy a kocsivissza/soremelés pár újsor karakterként szerepel a csatornában, ami megváltoztatja a fájl látszólagos méretét.

Hiba esetén visszatérési értékként –1-et kapunk. Ha a fájl természeténél fogva képtelen közvetlen hozzáférést biztosítani – pl. modem használatakor –, akkor értéke nem definiált.

Vonatkozó függvények: **fseek( )**, **fgetpos( )**.

■ *fwrite*

```
#include <stdio.h>
int fwrite(const void *buf, size_t size, size_t count, FILE
*stream);
```

Az **fwrite( )** függvény *count* számú *size* méretű objektumot ír ki a *stream* által mutatott csatornába a *buf* címen található karaktertömbből. A művelet során a fájlpozíció-indikátor a kiírt karakterek számával kerül tovább.

Az **fwrite( )** függvény a ténylegesen kiírt adatcsomagok számát írja ki, amely sikeres esetben megegyezik a *count*-ban megadott értékkel. Ha ennél kevesebb kerül a csatornába, akkor az azt jelzi, hogy valahol hiba történt. Szöveges fájlok esetén a különböző karakter-helyettesítéseknek nincs hatásuk a visszatérési értékre.

Vonatkozó függvények: **fread( )**, **fscanf( )**, **getc( )**, **fgetc( )**.

■ *getc*

```
#include <stdio.h>
int getc(FILE *stream);
```

A **getc( )** függvény a bemeneti adatfolyam következő karakterét olvassa be az aktuális fájlpozícióról, miközben egy hellyel előrébb állítja a fájlpozíció-indikátort. A karakterek **unsigned char**-ként kerültek beolvasásra, amelyeket majd a függvény egész típusúvá konvertál.

A fájlvége elérésekor a függvény **EOF**-t ad vissza. Mivel azonban az **EOF** egy érvényes integer érték, binárisként megnyitott fájlok használatakor az **feof( )** függvény szükséges annak megállapítására, hogy valóban elértük-e a fájlvégét. Ha a **getc( )** végrehatása során hiba lép fel, visszatérési értéke akkor is **EOF** lesz. Bináris fájlok esetében a hibát az **ferror( )** függvénnyel mutathatjuk ki.

A **getc( )** és **fgetc( )** függvények között nincsen különbség. A legtöbb implementációban a **getc( )** egyszerűen az alábbi makróként van definiálva:

```
#define get C(fp) fgetc(fp)
```

Ez azt jelenti, hogy a **getc( )** minden előfordulása **fgetc( )**-re cserélődik.
Vonatkozó függvények: **fputc( )**, **fgetc( )**, **putc( )**, **fopen( )**.

■ *getchar*

```
#include <stdio.h>
int getchar(void);
```

A **getchar( )** függvény visszaad egy a **stdin**-ról beolvasott karaktert. A karaktert **unsigned char** típusúként olvassa be, amelyet integer-ré alakít.

A fájlvége elérésekor a függvény **EOF**-t ad vissza. Mivel azonban az **EOF** egy érvényes integer érték, binárisként megnyitott fájlok használatakor az **feof( )** függvény szükséges annak megállapítására, hogy valóban elértük-e a fájlvégét. Ha a **getchar( )** végrehajtása során hiba lép fel, visszatérési értéke akkor is **EOF** lesz. Bináris fájlok esetében a hibát az **ferror( )** függvénnyel mutathatjuk ki.

A **getchar( )** függvényt gyakran makróként valósítják meg.

Vonatkozó függvények: **fputc( )**, **fgetc( )**, **putc( )**, **fopen( )**.

■ *gets*

```
#include <stdio.h>
char *gets(char *str);
```

Az **gets( )** függvény karaktereket olvas be a **stdin**-ról és azokat a *str* pointer által mutatott karaktertömbbe helyezi. A karakterek olvasásának a sorvége vagy az **EOF** jel vet véget. Az utolsó karakter beolvasása és a tömbbe helyezése után egy 0 kerül a tömb végére. Az újsor karakter – ha volt ilyen a beolvasottak végén (ez megszakítja a tovább olvasást) nem másolódik a **str*-be.

Sikeres végrehajtás esetén **gets( )** *str*-el tér vissza, máskülönben nullpointert kapunk. Amennyiben olvasási hiba történik, a *str* címén lévő tömb tartalma határozatlan lesz. Mivel hiba esetén is és fájlvége jel olvasásakor is nullpointert kapunk vissza, annak kiderítésére, hogy pontosan melyik történt a **feof( )** vagy a **ferror( )** használata szükséges.

Nincs módunkban korlátozni a **gets( )** által beolvasott karakterek számát, ezért nekünk kell gondoskodnunk arról, hogy a *str* által mutatott tömb túl ne csorduljon. (Lásd az alábbi programozási tippet.)

Programozási tipp

*A **gets( )** használatakor fennáll az a veszély, hogy a használt tömb tartományából kilépünk, amikor a felhasználó a vártnál több karakterrel áraszt el minket. Ennek az az oka, hogy a **gets( )** nem végez határ ellenőrzést. A problémát úgy is megkerülhetjük, hogy az **fgets( )-t** használjuk és input adatfolyamként a **stdin**-t jelöljük meg, ugyanis az **fgets( )** hívásakor argumentumként meg kell adni a maximálisan beolvasható karakterek számát. Ezáltal megakadályozhatjuk a tömb tartományának átlépését, viszont az **fgets( )** nem szűri ki az input végét jelentő újsor karaktert, a **gets** viszont igen. Így a programozó feladata az esetleges újsor karakter manuális eltávolítása. Ezt mutatja be az alábbi program:*

```
#include <stdio.h>
#include <string.h>

int main(void)
{
 char str[10];
 int i;

 printf("Kérek egy stringet: ");
 fgets(str, 10, stdin);

 /* az újsor karakter kihagyása, ha van */
 i= strlen(str)-1;
 if(str[i]=='\n') str[i] = '\0';

 printf("Ez a megadott string: %s", str);

 return 0;
}
```

*Habár az **fgets( )** használata egy kicsivel több munkát igényel, megéri: megakadályozhatjuk a kétséges utóhatásokkal járó tömbtartomány túlcsordulását.*

■ *perror*

```
#include <stdio.h>
void perror(const char *str);
```

A **perror( )** függvény az **errno** globális változó értékét másolja be egy stringbe, majd kiírja azt a **stderr**-ra. Ha az *str* értéke nem nulla, akkor először az kerül kiírásra, majd az implementáció által definiált hibaüzenet. A kettőt pontosvessző választja el egymástól.

■ *printf*

```
#include <stdio.h>
int printf(const char *format, ...);
```

A **printf( )** függvény a **stdout**-ra írja ki argumentumlistáját alkotó értékeket a *format* által mutatott stringben meghatározott formátumban.

A **format* string két különböző dologból áll. Az elsőt azon karakterek jelentik, amelyeket kiíratunk a képernyőre, a másik azon karaktersorozatok, amelyek formázó parancsokként működnek, ezek felelősek az argumentumok megjelenítésének módjáért. A formázóparancsok százalékjellel kezdődnek, ezt követi a formázókód. Az argumentumok számának meg kell egyeznie a formázóparancsok számával, az argumentumoknak és a rájuk vonatkozó formázódirektíváknak egyforma sorrendben kell szerepelniük. Az alábbi **printf( )** hívás például azt írja ki, hogy „Holnap találkozunk mr. X-szel 10-kor."

```
printf("Holnap %s mr. %c-szel %d-kor", "találkozunk",
'X', 10);
```

Amennyiben kevesebb az argumentum, mint a formázóparancs, akkor a kimenet definiálatlan. Fordított esetben – amikor több argumentum van, mint formázóparancs –, akkor az argumentumtöbblet nem kerül kiírásra. A formázóparancsokat az alábbi táblázat foglalja össze:

Kód	*Formátum*
%c	Karakter
%d	Előjeles decimális integer
%i	Előjeles decimális integer
%e	Tudományos jelölés (kisbetűs „e"-vel)
%E	Tudományos jelölés (nagybetűs „E"-vel)
%f	Decimális lebegőpontos
%g	%e vagy %f közül a rövidebbik (ha %e a rövidebb, akkor az „e" kisbetűs)
%G	%E vagy %f közül a rövidebbik (ha %E a rövidebb, akkor az „E" nagybetűs)
%o	Előjel nélküli oktális
%s	Karakterlánc
%u	Előjel nélküli decimális egész
%x	Előjel nélküli hexadecimális (kisbetűs megjelenítés)
%X	Előjel nélküli hexadecimális (nagybetűs megjelenítés)
%p	Mutató megjelenítése
%n	Az ennek megfelelő argumentum egy integer mutató, amelybe az eddig kiírt karakterek száma kerül, ez az érték meg is jelenik.
%%	% kiíratása

A **printf( )** függvény a ténylegesen kiírt karakterek számát adja vissza. Negatív visszatérési érték hibára utal.

A formázóparancsok kiegészíthetők különböző módosítókkal, amelyekkel meghatározhatjuk a kijelzendő adatra szánt hely szélességét (mezőszélesség), pontosságot, és beállíthatjuk a „balra igazít" flaget. A % és a formázó kód közé beszúrt egész számmal a minimális mezőszélességet adhatjuk meg. Hatására – amennyiben a kiírandó adat nem éri el a meghatározott minimális szélességet – extra 0-k és szóközök biztosítják a minimális méretet. Ha a string vagy szám hossza meghaladja a megadott minimumot, akkor az teljes hosszában megjelenítésre kerül. Alapértelmezésben a kiegészítés szóközökkel történik, ha ezt 0-val kívánjuk megvalósítani, akkor egy 0-t kell írni a szélességspecifikátor elé. Például a **%05d** egy 5 jegyűnél kisebb számot 0-kal egészíti ki, hogy a teljes hossza 5 legyen.

A pontosság módosító hatása függ attól, hogy milyen formázókódot módosítunk. Megadása úgy történik, hogy a mezőszélesség specifikátor mögé egy ponttal elválasztva írjuk a kívánt pontosságot. Az **e**, **E** és **f** formátumok esetén a pontosságmódosító a kiírandó tizedes jegyek számát jelenti. Például a **%10.4.f** hatására egy olyan szám jelenik meg a képernyőn, amelynek hossza legalább 10 karakternyi és 4 tizedes jegye van. A pontosságmódosító, amikor a **g** vagy **G** formátumokkal alkalmazzuk, a maximális értékes kiírandó számjegyet határozza meg. Egészekre használva a kiírandó számjegyek minimális számát rögzíti. Ekkor a szám, ha szükséges, vezető nullákkal egészül ki.

A pontosság módosító stringekre való alkalmazásakor, a pont utáni szám a maximális mezőhosszt határozza meg. Például a **%5.7s** egy olyan stringet ír ki, amely legalább 5 karakterből áll és hossza nem haladja meg a 7 karaktert. A hosszabb stringek ilyenkor (a végükön) megcsonkítva jelennek meg.

Alapértelmezésben minden kimenet *jobbra igazított* módban jelenik meg: ha az adat rövidebb, mint a neki szánt mezőben rendelkezésre álló hely, akkor az a mező jobb oldalára kerül. Ezt megváltoztathatjuk, ha a % jel után közvetlenül egy mínusz jelet szúrunk be. Például a **%–10.2f** az adott lebegőpontos számból két tizedesjegyet hagy meg, majd azt balra igazítva megjeleníti a számára kijelölt 10 karakter hosszú mezőben.

A hosszú és rövid integerek megjelenítésére meg kell említeni két további formátummódosítót. Ezek a **d**, **i**, **o**, **u** és **x** kódokkal használhatók. Az **l** módosító jelzi a **printf( )**-nek, hogy hosszú adat következik. Például a **%ld** hosszú integert jelenít meg. A **h** módosító egy rövid integer kiírására utasítja a **printf( )**-t. Így a **%hu**-val rövid előjel nélküli integer típusú adatot jelenítünk meg.

Az **l** módosító lebegőpontos formázó parancsokat (**e**, **f** és **g**) is megelőzhet. Ekkor az a típus **double** voltára utal.. Egy **long double** típusú adatot a %**L**-lel írathatunk ki.

Az **%n** parancs az addig kiírt karakterek számát helyezi be a hozzá tartozó argumentum által mutatott egész változóba. Az alábbi kódrészletben az „Ez egy próbasor" szöveg mögött a 15-ös szám is megjelenik a képernyőn.

```
int i;

printf("Ez egy próbasor%n", &i);
printf("%d", i);
```

A **printf( )**-ben – formázókódokkal kombinálva – a # karakter is speciális jelentéshez jut. A **g**, **f**, valamint az **e** kódok előtt szerepeltetve a tizedes pont kiíratását kényszeríthetjük ki, olyan esetekben is, amikor a kiírandó számban nem szerepelnek tizedes jegyek. Ha az **x** direktíva elé #-ot helyezünk, akkor az illető hexadecimális szám a **0x** prefixszel íródik ki. Az **o** formázó paranccsal kombinálva az adott oktális szám **0**-val fog kezdődni. A többi formátumspecifikátorral párban a # nem használható.

A **printf( )**-ben minimális mezőszélesség és a pontossági formátummódosítók paraméterezhetők is, rugalmasabbá téve a megjelenítést. Ennek megvalósítására szolgál a konkrét számok helyett szereplő *, amit a formátumstring kiértékelésekor **printf( )**-nek kell feloldania a megfelelő argumentumhelyettesítéssel.

Vonatkozó függvények: **scanf( )**, **fprintf( )**.

■ *putc*

```
#include <stdio.h>
int putc(int ch, FILE *stream);
```

A **putc( )** függvény a *ch* alsó bájtjában tárolt karaktert írja a *stream* által meghatározott kimeneti csatornára. A hívásnál használt karaktereket egészként jutnak el a függvényhez, ezért a **putc( )** argumentumaként bátran használhatunk karakterváltozót is.

Visszatérési értékként a **putc( )** a sikeresen kiírt karaktert adja vissza. Hiba esetén **EOF**-t kapunk. Amennyiben a csatornát binárisként nyitottuk meg, a *ch*-nak értékül **EOF** is adható. Ez egyben azt jelenti, hogy az **ferror( )** függvényt kell meghívnunk, hogy az esetleges hibákra fény derüljön.

Vonatkozó függvények: **fgetc( )**, **fputc( )**, **getchar( )**, **putchar( )**.

■ *putchar*

```
#include <stdio.h>
int putchar(int ch);
```

A **putchar( )** függvény a *ch* alsó bájtjában elhelyezett karaktert írja ki a **stdout**-ra (standard output). Funkcionálisan megegyezik a **putc(ch, stdout)** függvényhívással. A hívásnál használt karaktereket egészként jutnak el a függvényhez, ezért a **putchar( )** argumentumaként bátran használhatunk karakterváltozót is.

A **putchar( )** függvény sikeres végrehajtás esetén, a kiírt karaktert adja vissza. Hiba esetén **EOF** értéket kapunk. Amennyiben a csatornát binárisként nyitottuk meg, a *ch*-nak teljesen legális módon **EOF** értéket is adhatunk. Ez egyben azt jelenti, hogy az **ferror( )** függvényt kell meghívnunk, hogy az esetleges hibákra fény derüljön.

Vonatkozó függvények: **putc( )**.

■ *puts*

```
#include<stdio.h>
int puts(char *str);
```

A **puts( )** függvény a *str* által mutatott karakterláncot írja ki a szabványos kimenetre (stdout). A string végét jelző nullkaraktert a függvény újsor karakterré alakítja. A **puts( )** függvény normális működés esetén nem negatív értékkel tér vissza, míg az **EOF** visszatérési érték hibára utal.

Vonatkozó függvények: **putc( )**, **gets( )**, **printf( )**.

■ *remove*

```
#include<stdio.h>
int remove(const char *fname);
```

A **remove( )** függvény törli az **fname* nevű fájlt. Sikeres törlés után 0-t, hiba esetén nullától különböző értéket kapunk vissza.

Vonatkozó függvények: **rename( )**.

■ *rename*

```
#include <stdio.h>
int rename(const char *oldfname, const char *newfname);
```

A **rename( )** függvény az *oldfname* stringmutató által megadott fájlt nevezi át a *newfname* által mutatott nevűre. A *newfname* nem lehet létező könyvtárnév.

Sikeres átnevezés után 0-t, hiba esetén nullától különböző értéket kapunk vissza.

Vonatkozó függvények: **remove( )**.

■ *rewind*

```
#include <stdio.h>
void rewind(FILE *stream);
```

A **rewind( )** függvény a *stream*-hez asszociált fájl fájlpozícióindikátorát állítja az adatfolyam elejére. A függvény egyúttal törli a fájlhoz rendelt hiba- és a fájlvége flageket. Nincs visszatérési értéke.

Vonatkozó függvények: **fseek( )**.

■ *scanf*

```
#include <stdio.h>
int scanf (const char *format, ...);
```

A **scanf( )** függvény egy általános célú beolvasó rutin, amely a **stdin** csatornáról olvas és a beolvasott információt az argumentum listában megadott mutatók által meghatározott változókban tárolja el. A **scanf()** képes minden predefinit adattípus beolvasására és azok megfelelő formátumúvá alakítására.

A *format* mutató által megjelölt vezérlőstringet alkotó karakterek három csoportba sorolhatók:

Formátumspecifikátorok
Üreskarakterek
Közönséges karakterek

A formátumspecifikátorokat a % jel vezeti be, az ezután szereplő karakter mondja meg a **scanf( )**-nek, hogy a következő beolvasandó adat milyen típusú. Például **%s** stringet, míg a **%d** integert olvas. A megadott pointerek által mutatott változókba a **scanf( )** kódok segítségével beolvasott adatok az argumentumlistának megfelelő sorrendben kerülnek be. A formátum specifikátorokat az alábbi táblázat foglalja össze:

Kód	*Jelentés*
%c	Egy karaktert olvas
%d	Decimális integert olvas
%i	Integert olvas
%e	Lebegőpontos számot olvas
%f	Lebegőpontos számot olvas
%g	Lebegőpontos számot olvas
%o	Oktális számot olvas
%s	Stringet olvas
%x	Hexadecimális számot olvas
%p	Mutatót olvas
%n	Megjelenéséig beolvasott karakterek számát kapja meg
%u	Előjel nélküli egészt olvas
%[]	Elfogadás egy megadott karakterhalmazból
%%	Százalékjel olvasása

A formázó stringet a függvény balról jobbra olvassa el és a formázó kódokat sorban illeszti az argumentumlistát alkotó argumentumokhoz.

A formátumstringben található üreskarakter megadásával azt közöljük a **scanf( )** függvénnyel, hogy amennyiben a bementeti csatornán egy vagy több ilyen karaktert olvas, azt ugorja át. Az üreskarakter lehet szóköz, tabulátor, vagy újsor karakter. Átfogalmazva, a megadott kitöltő karakter egymás utáni előfordulásait (lehet 0 db. is) a **scanf( )** beolvassa (az első közönséges karakterig), de nem tárolja őket.

A formátumstringben található közönséges karakter az adatfolyam soron következő karakterét kell, hogy jelentse (szükséges az egyezés), amelyet a

függvény átolvas. Például: **%d**, **%d** karaktersorozat hatására először egy integer kerül beolvasásra, majd egy vessző, amely nem tárolódik el, végül egy másik integer. Ha a megadott karakter nem egyezik az adatfolyam következő karakterével, a rutin végrehajtása befejeződik.

A **scanf( )**-nek minden változót cím szerint kell átadnunk. Ez azt jelenti, hogy minden argumentum az input fogadására kijelölt változókra mutató pointerek kell hogy legyenek.

A beolvasott adatokat egymástól szóközök, tabulátorok vagy újsor karakterek kell hogy elválasszák. Egyéb központozáshoz használt írásjelek, úgymint vesszők, pontosvesszők stb. nem számítanak elválasztó jeleknek. Ez azt jelenti, hogy pl. a

```
scanf("%d%d", &r, &c);
```

hívás elfogadja a **10 20** inputot, de leáll a **10,20** bemenet olvasása esetén.

A % elé helyezett * karakterrel elérhető, hogy a megfelelő formátumú adat eltárolás nélkül kerüljön beolvasásra. Ennek megfelelően a következő parancs a **10/20** bemeneti folyam hatására 10-et másol **x**-be, majd átolvassa az osztásjelet és az **y**-nak 20-at ad értékül:

```
scanf("%d%*c%d", &x, &y);
```

A formázóparancsokkal megadhatjuk a beolvasott adat maximális mezőszélességét is. A mezőszélesség specifikátor a % jel és a formázókód közé elhelyezett szám, amellyel az egy mezőhöz tartozó beolvasott karakterek számát korlátozhatjuk. Például: ha nem kívánunk 20 karakternél többet olvasni az **address** stringbe, akkor a következő sort használnánk:

```
scanf("%20s", address);
```

Amennyiben a bemeneti csatorna több, mint 20 karakterből áll, akkor több hívással lehet végig olvasni az inputcsatornát. Egy hívás mindig onnan kezdi a beolvasást, ahol az előző abbahagyta. Egy adott mező feltöltése üreskarakter olvasáskor befejeződik, akkor is, ha a megadott maximális mezőszélességet még nem értük el. Ebben az esetben **scanf( )** a következő mezővel folytatja.

Habár a szóközt, a tabulátort és az újsor karaktert mezőszeparátorként használjuk, egy egyszeri karakterolvasás ezeket is úgy kezeli, mint bármely más karaktert. **x y** adatfolyamot feltételezve a

```
scanf("%c%c%c", &a, &b, &c);
```

sor eredményeként az **a**, **b** és **c** változókba rendre az x, a szóköz és az y karakter kerül.

Figyelem: a vezérlőstringben előforduló egyéb karakterek – beleértve a szóközöket, a tabulátorokat és az újsorokat – a korábban említettek szerint azt a célt szolgálják, hogy átolvassák a bemeneti adatfolyam soron következő karakterét (egyezés szükséges). Például a **10t20** adatfolyamot tekintve a

```
scanf("%st%s", &x, &y);
```

sor végrehajtása után **x**-be 10, **y** 20 kerül. A **t** nem kerül tárolásra, éppen a vezérlőstringben található **t** miatt.

Egy másik hasznos szolgáltatása a **scanf( )**-nek az úgynevezett elfogadó halmaz megadásán keresztül érhető el. Egy elfogadó halmazt azon karakterek alkotják, amelyeket következő beolvasáskor a függvény elfogad és bemásolja a megfelelő karaktertömbbe. Egy elfogadó halmaz megadása úgy történik, hogy százalékjel után a halmazt alkotó karaktereket szögletes zárójelek közé zárjuk. Például a következő elfogadó halmazzal azt jelezzük a **scanf( )** függvénynek, hogy a csak az A, B és C karakterek olvashatók be:

```
%[ABC]
```

A függvény egészen addig olvas a bemeneti csatornáról és tárolja el a karaktereket a megadott stringben, amíg egy olyan karakterbe nem „botlik", amely nincs benne az elfogadó halmazban. Ehhez a formázó parancshoz tartozó argumentum szükségképpen egy karaktertömb-mutató kell hogy legyen. Visszatéréskor a tömbben a beolvasott karakterekből álló nullavégű stringet kapunk.

Lehetőség van az elfogadó halmazt komplementerével definiálni, ilyenkor a ^ karakter áll a nyitózárójel utáni első helyen. Ilyenkor a **scanf( )** csak olyan karaktereket fogad el, melyek nincsenek a zárójelek között.

Az elfogadó halmaz megadás bizonyos esetekben egyszerűsíthető, ugyanis megadhatunk tartományokat a kötőjel segítségével. Az alábbi formázó parancs hatására a **scanf( )** minden karaktert elfogad A-tól Z-ig.

```
%[A-Z]
```

Fontos észben tartani, hogy az elfogadó halmazban a nagybetűk és a kisbetűk különbözőnek számítanak. Ezért ha bizonyos karakterek nagy- és kisbetűs változatát is el akarjuk fogadtatni, akkor kénytelenek vagyunk mindkét formában szerepeltetni őket az elfogadó halmazban.

A **scanf( )** függvény visszatérési értéke a sikeresen beolvasott és eltárolt mezők számát tartalmazza. Így ez az érték nem foglalja magába azon mezőket, amelyeket a * módosító hatása miatt a függvény csupán átolvas. **EOF**-t kapunk vissza, ha hiba történt az első mező beolvasása előtt.

Vonatkozó függvények: **printf( )**, **fscan( )**.

■ *setbuf*

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

A **setbuf( )** függvénnyel a meghatározhatjuk a puffert, amelyet a megadott *stream* használhat, vagy, ha *buf* nullpointer, akkor kikapcsolhatjuk a pufferelést. A programozó által definiált puffernak mindenképpen **BUFSIZ** karakterhosszúságúnak kell lennie. A **BUFSIZ**-t az STDIO.H definiálja.

A **setbuf( )** függvénynek nincs visszatérési értéke.

Vonatkozó függvények: **fopen( )**, **fclose( )**, és **setvbuff( )**.

■ *setvbuf*

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, size_t
  size);
```

A **setvbuf( )** függvény lehetővé teszi a programozó számára, hogy megadja az argumentumként szereplő csatornához rendelt puffert, annak méretét és a pufferolás módját. A *buf* által mutatott karaktertömb lesz az I/O műveleteknél használt puffer. A puffer méretét a *size* paraméteren keresztül állíthatjuk be, a *mode*-dal a puffer használatának módját határozhatjuk meg. Ha *buf*-ban nullpointert adunk meg, akkor a **setvbuf( )** saját puffert allokál.

A *mode* paraméter lehetséges értékei a következők: **_IOFBF**, **_IONBF**, **_IOLBF**. Ezeket a STDIO.H könyvtár definiálja. Ha a *mode* **_IOFBF**-re van állítva, akkor teljes pufferolás történik. Az **_IOLBF** mód ún. sorpufferolást eredményez, ami azt jelenti, hogy kimeneti adatfolyam esetén a puffer kiürül minden újsor karakter kiírásakor, illetve bemeneti csatorna esetén beolvasáskor a pufferolás a következő újsor karakterig történik. Mindkét esetben a megtelt puffer automatikusan kiürítésre kerül. Ha a *mode*-nak **_IONBF**-et jelölünk meg, akkor nincs pufferelés.

A *size* értéke pozitív kell hogy legyen.

A **setvbuf( )** függvény nullával tér vissza hibamentes működés után, egyébként nullától különböző értéket kapunk.

Vonatkozó függvények: **setbuf( )**.

■ *sprintf*

```
#include <stdio.h>
int sprintf(char *buf, const char *format,...);
```

Az **sprintf( )** csak abban tér el a **printf( )**-től, hogy a kimenet ahelyett, hogy a konzolra kerülne, a *buf* által mutatott tömbbe kerül. Részletesen, l. **printf( )**.

A függvény visszatérési értéke megegyezik a tömbbe írt karakterek tényleges számával.

Vonatkozó függvények: **printf( )**, **fsprintf( )**.

■ *sscanf*

```
#include <stdio.h>
int sscanf(const char *buf, const char * format,...);
```

A **sscanf( )** függvény csak abban tér el a **scanf( )**-től, hogy az adatokat nem **stdin**-ről olvassa, hanem a *buf* által mutatott tömbből. További részletek a **scanf( )** leírásánál találhatók.

A **scanf( )** függvény visszatérési értéke a sikeresen beolvasott és eltárolt mezők számát tartalmazza. Így ez az érték nem foglalja magába azon mezőket, amelyeket a * módosító hatása miatt a függvény csupán átolvas. A nulla visszatérési érték arra utal, hogy nem volt eltárolt mező a művelet során. **EOF**-t kapunk vissza, ha hiba történt az első mező beolvasása előtt.

Vonatkozó függvények: **scanf( )**, **fscanf( )**.

■ *tmpfile*

```
#include <stdio.h>
FILE *tmpfile(void);
```

A **tmpfile( )** függvény egy ideiglenes állományt nyit meg és visszaadja az ehhez tartozó csatornára mutató pointert. A függvény gondoskodik egyedi fájlnév használatáról, nehogy ütközés legyen más létező fájlokkal.

A **tmpfile( )** függvény hiba esetén nullpointert ad vissza, egyébként az állománymutatóval tér vissza.

A függvény által létrehozott ideiglenes fájl automatikusan törlődik a fájl bezárásakor vagy – ennek elmulasztása esetén – a program befejezésekor.

Vonatkozó függvények: **tmpnam( )**.

■ *tmpnam*

```
#include <stdio.h>
char *tmpnam(char *name);
```

A **tmpnam( )** függvény egy egyedi fájlnevet generál, amit eltárol a *name* által mutatott stringben. A **tmpnam( )** főfunkciója, hogy olyan ideiglenes fájlt tudjunk használni, amelynek a neve különbözik az összes többi fájlnévtől az aktuális könyvtárban.

A függvény maximum **TMP_MAX**-szor hívható meg. **TMP_MAX**-t a STDIO.H definiálja, értéke legalább 25. A **tmpnam( )** minden hívásakor új fájlnevet állít elő.

Sikeres működés esetén a *name*-re mutató pointert kapunk vissza, egyébként a függvény nullpointerrel tér vissza.

Vonatkozó függvények: **tmpfile( )**.

■ *ungetc*

```
#include<stdio.h>
int ungetc(int ch, FILE *stream);
```

Az **ungetc( )** függvény a *ch* alsó bájtján tárolt karaktert helyezi a *stream* paraméterben megadott bemeneti adatfolyam végére. Ezt a karaktert kapjuk meg a *stream*-en végzett következő olvasáskor. Az **fflush( )** és az **fseek( )** törli a karaktert és az **ungetc( )** hatását semmissé teszi.

Egy karakter ilyen módon történő „hozzácsapása" a bemeneti adatfolyam végéhez mindig megtehető. Egyes implementációk lehetővé teszik több karakter hozzávételét a függvény többszöri alkalmazásával.

EOF karakterrel a függvény nem használható.

Az **ungetc( )** törli az adatfolyamhoz tartozó fájlvége jelzőbitet. Szöveges csatorna esetén a fájlpozíció-indikátor értéke az összes beszúrt karakter végigolvasásáig határozatlan lesz, azután pedig visszakapja az első **ungetc( )** előtti értéket. Bináris csatornák esetén minden egyes **ungetc( )** hívás eggyel csökkenti a fájlpozíció-indikátort.

Normális működés esetén a visszatérési érték megegyezik *ch*-val. **EOF**-t kapunk vissza, ha végrehajtása során hiba történt.

Vonatkozó függvények: **getc( )**.

■ *vprintf, vfprintf, vsprintf*

```
#include <stdarg.h>
#include <stdio.h>
```

```
int vprint(char *format, va_list arg_ptr);
int vfprintf(FILE *stream, const char *format, va_list
  arg_ptr);
int vsprintf(char *buf, char *format, va_list  arg_ptr);
```

A **vprintf( )**, **vfprintf( )** és a **vsprinf( )** függvények funkcionálisan megegyeznek rendre a **printf( )**, az **fprintf( )** és az **sprintf( )** függvényekkel, azzal a kivétellel, hogy az argumentumlista helyett most egy argumentumokból álló listára mutató pointer szerepel. Ez a mutató **va_list** típusú, melyet a STDARG.H definiál.

Vonatkozó függvények: **va_arg( )**, **va_start( )** és **va_end( )**.

7. FEJEZET

A C string- és karakterkezelő függvényei

A C standard könyvtár gazdag és hasznos string- és karakterkezelő függvénykészlettel rendelkezik. C-ben és C++-ban a stringek nullavégű karaktertömbök. A stringkezelő függvények prototípusai a STRING.H fejben találhatók, míg karakterkezelő függvények prototípusait a CHAR.H fej tartalmazza.

Mivel a C-ben és a C++-ban nincsen értékhatár-ellenőrzés a tömbök manipulálásakor, a programozóra hárul a felelősség és a feladat, hogy megelőzze a tömb-index túlcsordulást. Ennek elhalasztása a program összedőléséhez vezethet.

A C/C++ terminológia szerint egy *megjeleníthető karakter* olyan karaktert jelent, amely a terminálon megjeleníthető. Ezek általában a szóköz (0x20) és a tilde (0xFE) közti karaktertartományt jelentik. A *vezérlő* karaktereket 0 és 0x1F közé eső karakterek és az DEL (0x7F) alkotják.

Történeti okok miatt a karakterkezelő függvények argumentumai egészek, azonban kizárólag az alsó bájtnak van szerepe; a karakterkezelő függvények automatikusan átváltják az argumentumaikat **unsigned char** típusúvá. Ezeket a függvényeket nyugodtan hívhatjuk karakterargumentumokkal, mert ezek automatikusan egésszé konvertálódnak.

A STRING.H fejállományban van definiálva a **size_t** típus, amely lényegében ugyanaz, mint az **unsigned**.

■ *isalnum*

```
#include<ctype.h>
int isalnum(int ch);
```

Az **isalnum( )** függvény nullától különböző értéket ad vissza, ha az argumentumként megadott karakter az ábécé egy betűje vagy egy számjegy. Ha a karakter nem alfanumerikus, a visszaadott érték 0.

Vonatkozó függvények: **isalpha( )**, **iscntrl( )**, **isdigit( )**, **isgraph( )**, **isprint( )**, **ispunct( )**, **isspace( )**.

■ *isalpha*

```
#include <ctype.h>
int isalpha(int ch);
```

Az **isalpha( )** függvény nullától különböző értéket ad vissza, ha *ch* az ábécé egy betűje, egyébként visszatérési értéke 0. Az, hogy milyen betűk alkotják az ábécét, nyelvfüggő. Az angol nyelvben ezek a kis- és nagybetűk A-tól Z-ig.

Vonatkozó függvények: **isalnum( )**, **iscntrl( )**, **isdigit( )**, **isgraph( )**, **isprint( )**, **ispunct( )**, **isspace( )**.

■ *iscntrl*

```
#include <ctype.h>
int iscntrl(int ch);
```

Az **iscntrl( )** függvény nullától különböző értéket ad vissza, ha a *ch* 0 és 0x1F közé esik vagy 0x7F (DEL), minden más esetben a visszaadott érték 0.

Vonatkozó függvények: **isalnum( )**, **isalpha( )**, **isdigit( )**, **isgraph( )**, **isprint( )**, **ispunct( )**, **isspace( )**.

■ *isdigit*

```
#include <ctype.h>
int isdigit(int ch);
```

Az **isdigit( )** függvény nullától különböző értéket ad vissza, ha *ch* egy számjegy, azaz a 0-tól 9-ig terjedő karakterek valamelyike. Nullát kapunk minden más esetben.

Vonatkozó függvények: **isalnum( )**, **isalpha( )**, **iscntrl( )**, **isgraph( )**, **isprint( )**, **ispunct( )**, **isspace( )**.

■ *isgraph*

```
#include <ctype.h>
int isgraph(int ch);
```

Az **isgraph( )** függvény nullától különböző értékkel tér vissza, ha *ch* egy megjeleníthető karakter és nem a szóköz. Minden más esetben nullát kapunk vissza. A megjeleníthető karakterek általában a 0x21 és a 0x7E közötti tartományba esnek.

Vonatkozó függvények: **isalphanum( )**, **isalpha( )**, **iscntrl( )**, **isdigit( )**, **isprint( )**, **ispunct( )**, **isspace( )**.

■ *islower*

```
#include <ctype.h>
int islower(int ch);
```

Az **islower( )** függvény nullától különböző értékkel tér vissza, ha *ch* kisbetű, minden más esetben 0-t kapunk vissza.

Vonatkozó függvények: **isupper( )**.

■ *isprint*

```
#include <ctype.h>
int isprint(int ch);
```

Az **isprint( )** függvény nullától különböző értékkel tér vissza ha *ch* megjeleníthető karakter, beleértve a szóközt is. Minden más esetben 0-t kapunk vissza. Megjeleníthető karakterek általában a 0x21 és a 0x7E közötti tartományba esnek.

Vonatkozó függvények: **isalnum( )**, **isalpha( )**, **iscntrl( )**, **isdigit( )**, **isgraph( )**, **ispunct( )**, **isspace( )**.

■ *ispunct*

```
#include <ctype.h>
int ispunct(int ch);
```

Az **ispunct( )** függvény nullától különböző értékkel tér vissza, ha a *ch* valamely *írásjel*. Minden más esetben 0-t kapunk vissza. Az *írásjel* minden olyan megjelenítendő karakter, amely nem alfanumerikus és nem a szóköz.

Vonatkozó függvények: **isalnum( )**, **isalpha( )**, **iscntrl( )**, **isdigit( )**, **isgraph( )**, **ispunct( )**, **isspace( )**.

■ *isspace*

```
#include <ctype.h>
int isspace(int ch);
```

Az **isspace( )** függvény nullától különböző értékkel tér vissza, ha *ch* a szóköz, a vízszintes tabulátor a függőleges tabulátor, a lapdobás, a kocsivissza, vagy az újsor karakterek valamelyike. Minden más esetben 0-t kapunk vissza.

Vonatkozó függvények: **isalnum( )**, **isalpha( )**, **iscntrl( )**, **isdigit( )**, **isgraph( )**, **ispunct( )**.

■ *isupper*

```
#include <ctype.h>
int isupper(int ch);
```

Az **isupper( )** függvény nullától különböző értékkel tér vissza, ha *ch* nagybetű. Minden más esetben 0-t kapunk vissza.

Vonatkozó függvények: **islower( )**.

■ *isxdigit*

```
#include <ctype.h>
int isxdigit(int ch);
```

Az **isxdigit( )** függvény nullától különböző értékkel tér vissza, ha *ch* egy hexadecimális számjegy. Minden más esetben 0-t kapunk vissza. Egy hexadecimális számjegy az alábbi karakterek közül kerülhet ki: A-F, a-f, vagy 0-9.

Vonatkozó függvények: **isalnum( )**, **isalpha( )**, **iscntrl( )**, **isdigit( )**, **isgraph( )**, **ispunct( )**, **isspace( )**.

■ *memchr*

```
#include <string.h>
void *memchr(const void *buffer, int ch, size_t
             count);
```

A **memchr( )** függvény a *buffer* által mutatott tömbben keresi a *ch* első előfordulását az első *count* karakterben.

A **memchr( )** függvény a *ch* első *buffer*-beli előfordulására mutató pointert ad vissza, ha nem talál ilyet, akkor nullpointerrel tér vissza.

Vonatkozó függvények: **memcpy( )**, **isspace( )**.

■ *memcmp*

```
#include <string.h>
int memcmp(const void *buf1, const void *buf2, size_t
           count);
```

A **memcmp( )** függvény összehasonlítja a *buf1* és *buf2* pointerek által mutatott tömbök első *count* karakterét.

A **memcmp( )** függvény visszatérési értéke egy egész szám, amelynek jelentését az alábbi táblázatból olvashatjuk ki.

Érték	*Jelentés*
negatív szám	*buf1* kisebb, mint *buf2*
0	*buf1* egyenlő *buf2*-vel
pozitív szám	*buf2* nagyobb, mint *buf1*

Vonatkozó függvények: **memchr( )**, **memcpy( )**, **strcmp( )**.

■ *memcpy*

```
#include <string.h>
void *memcpy(void *to, const void *from, size_t count);
```

A **memcpy( )** függvény a *from* által mutatott tömbből *count* karakternyit átmásol a *to* által mutatott tömbbe. Ha a tömbök átfedik egymást, a függvény viselkedése definiálatlan.

A **memcpy( )** a *to* mutatóval tér vissza.

Vonatkozó függvények: **memmove( )**.

■ *memmove*

```
#include <string.h>
void *memmove(void *to, const void *from, size_t count);
```

A **memmove( )** függvény a *from* által mutatott tömbből *count* karakternyit átmásol a *to* által mutatott tömbbe. Ha a tömbök átfedik egymást, a másolás akkor is helyesen megy végbe: *from* tartalma bekerül *to*-ba, de a *from* módosulhat.

A **memmove( )** függvény a *to* mutatóval tér vissza.

Vonatkozó függvények: **memcpy( )**.

■ *memset*

```
#include <string.h>
void *memset(void *buf, int ch, size_t count);
```

A **memset( )** függvény a *ch* alsó bájtját másolja *buf* által mutatott tömb első *count* karakter helyére. Visszatéréskor a *buf*-t adja vissza.

A **memset( )** függvényt leggyakrabban egy adott memóriaterület inicializálására használjuk: feltöltjük valamilyen ismert értékkel.

Vonatkozó függvények: **memcmp( )**, **memcpy( )**, **memmove( )**.

■ *strcat*

```
#include <string.h>
char *strcat(char *str1, const char *str2);
```

A **strcat( )** függvény a *str2* által mutatott stringhez konkatenálja (végéhez csatolja) a *str1* által mutatott string másolatát, a kapott string is nulla végű lesz. Az eredeti *str1* végét jelző nullkarakter felülíródik: oda kerül a *str2* első karaktere. A *str2* a művelet során nem változik. Ha a tömbök átfedik egymást a művelet eredménye nem definiált.

A **strcat( )** függvény a *str1*-et adja vissza.

Vigyázat, nem történik indexhatár-ellenőrzés! A programozónak kell biztosítani, hogy *str1* elég nagy legyen ahhoz, hogy eredeti tartalma mellett *str2* is beleférjen.

Vonatkozó függvények: **strchr( )**, **strcmp( )**, **strcpy( )**.

■ *strchr*

```
#include <string.h>
char *strchr(const char *str, int ch);
```

A **strchr( )** függvény megkeresi a *str* által mutatott stringben a *ch* alsó bájtjának első előfordulását. Találat hiányában nullpointert kapunk vissza.

Vonatkozó függvények: **strpbrk( )**, **strspn( )**, **strtok( )**.

■ *strcmp*

```
#include <string.h>
int strcmp(const char *str1, const char *str2);
```

A **strcmp( )** függvény két stringet hasonlít össze lexikografikusan és egy egészt ad vissza, melynek értelmezéséhez az alábbi táblázat ad segítséget:

Érték	*Jelentés*
negatív szám	*str1* kisebb, mint *str2*
0	*str1* egyenlő *str2*-vel
pozitív szám	*str1* nagyobb, mint *str2*

Vonatkozó függvények: **strchr( )**, **strcpy( )**, **strcmp( )**.

■ *strcoll*

```
#include <string.h>
int strcoll(const char *str1, const char *str2);
```

A **strcoll( )** függvény két stringet hasonlít össze, nevezetesen a *str1* és a *str2* által mutatottakat. Az összehasonlítás a helyi beállításoknak megfelelően történik, amelyet előzetesen a **setlocale( )** függvénnyel állíthatunk be (l. „**setlocale**").

A **strcoll( )** által visszaadott egész számot a következőképpen kell értelmezni:

Érték	*Jelentés*
negatív szám	*str1* kisebb, mint *str2*
0	*str1* egyenlő *str2*-vel
pozitív szám	*str1* nagyobb, mint *str2*

Vonatkozó függvények: **memcmp( )**, **strcmp( )**.

■ *strcpy*

```
#include <string.h>
char *strcpy(char *str1, const char *str2);
```

A **strcpy ( )** függvény a *str2* által mutatott stringet másolja a *str1* által mutatott tömbbe. Fontos, hogy a *str2* egy nullavégű stringre mutasson. A függvény a *str1* mutatót adja vissza.

Ha *str1* és *str2* átfedik egymást, a **strcpy( )** viselkedése nem definiált.

Vonatkozó függvények: **memcpy( ), strchr( ), strcmp( ), strncmp( ).**

■ *strcspn*

```
#include <string.h>
size_t strcspn(const char *str1, const char *str2);
```

A **strcspn( )** függvény megadja a *str1* által mutatott string leghosszabb olyan kezdőszeletének a hosszát, amelyben nem szerepel a *str2* által mutatott karakterek egyike sem. Átfogalmazva, a **strcspn( )** a *str1* által mutatott tömb első olyan karakterének az indexét adja vissza, amely a *str2* által mutatott string valamely karakterével megegyezik.

Vonatkozó függvények: **strchr( ), strpbrk( ), strstr( ), strtok( ).**

■ *strerror*

```
#include <string.h>
char *strerror(int errnum);
```

A **strerror( )** függvény egy pointert ad vissza az *errnum* értékéhez tartozó implementáció által definiált stringre. Semmilyen körülmények között se módosítsuk ezt a stringet!

■ *strlen*

```
#include <string.h>
size_t strlen(char *str);
```

A **strlen( )** függvény a *str* által mutatott nullavégű string hosszát adja vissza, a hossz a nullterminátor nélkül értendő.

Vonatkozó függvények: **memcpy( ), strchr( ), strcmp( ), strncmp( ).**

■ *strncat*

```
#include <string.h>
char *strncat (char *str1, const char *str2, size_t
               count);
```

Az **strncat( )** függvény a *str2* által mutatott string *count* hosszúságú kezdőszeletét konkatenálja a *str1* által mutatott stringhez és gondoskodik az eredménystring (amelyre *str1* mutat) végét jelző nullkarakterről is. A **strncat( )** a bemenő *str1*-et lezáró nullkaraktert felülírja a *str2* első karakterével. A *str2*-t a művelet érintetlenül hagyja. Ha a stringek átfedik egymást, az eredmény definiálatlan.

Az **strncat( )** függvény a *str1* mutatót adja vissza.

Vigyázat, nem történik indexhatár ellenőrzés. A programozónak kell biztosítani, hogy a *str1* elég nagy legyen ahhoz, hogy eredeti tartalma mellett a *str2* kívánt szakasza is beleférjen.

Vonatkozó függvények: **strcat( )**, **strnchr( )**, **strncmp( )**, **strncpy( )**.

■ *strncmp*

```
#include <string.h>
int strncmp(const char *str1, const char *str2, size_t
            count);
```

Az **strncmp( )** függvény lexikografikusan összehasonlítja a *str1* és a *str2* pointerek által mutatott két nullavégű string első *count* karakterét és eredményül egy egész számot ad vissza, amely a következőképpen értelmezhető:

Érték	*Jelentés*
negatív szám	*str1* kisebb, mint *str2*
0	*str1* egyenlő *str2*-vel
pozitív szám	*str1* nagyobb, mint *str2*

Elképzelhető, hogy a két string közül valamelyik rövidebb, mint *count*. Ekkor az összehasonlítás az első nullkarakter felbukkanásakor véget ér.

Vonatkozó függvények: **strcmp( )**, **strnchr( )**, **strncpy( )**.

■ *strncpy*

```
#include <string.h>
char *strncpy (char *str1, const char *str2, size_t
               count);
```

A **strncpy( )** függvénnyel a *str2* által mutatott string első *count* karakterét másolhatjuk a *str1* által mutatott stringbe. Követelmény, hogy a *str2* egy nullavégű stringre mutasson.

Ha *str1* és *str2* átfedi egymást, akkor a függvény viselkedése nem definiált.

Amennyiben a *str2* által mutatott string rövidebb, mint *count*, akkor a *str1* végére annyi nulla kerül, hogy összesen meglegyen a *count* átmásolt karakter.

Ha viszont a *str2* hosszabb, mint a *count*, akkor a *str1* által mutatott string nem lesz nullavégű.

A **strncpy( )** függvény a *str1* mutatót adja vissza.

Vonatkozó függvények: **memcpy( )**, **strchr( )**, **strncat( )**, **strncmp( )**.

■ *strpbrk*

```
#include <string.h>
char *strpbrk (const char *str1, const char *str2);
```

Az **strpbrk( )** függvény egy pointert ad vissza, amely a *str1* első olyan karakterére mutat, amely megegyezik a *str2* által mutatott string valamelyik karakterével. A nullkarakterek egyezésére a függvény nem érzékeny. Találat híján nullpointert kapunk vissza.

Vonatkozó függvények: **strspn( )**, **strrchr( )**, **strstr( )**, **strtok( )**.

■ *strrchr*

```
#include <string.h>
char *strrchr(const char *str, int ch);
```

Az **strrchr( )** függvény egy olyan pointert ad vissza, amely a *str* által mutatott stringben *ch* alsó bájtja által meghatározott karakter utolsó előfordulására mutat. Ha ilyet nem talál, akkor nullpointert kapunk vissza.

Vonatkozó függvények: **strpbrk( )**, **strspn( )**, **strstr( )**, **strtok( )**.

■ *strspn*

```
#include <string.h>
size_t strspn(const char *str1, const char *str2);
```

A **strspn( )** függvény a *str1* által mutatott string leghosszabb olyan kezdőszeletének a hosszát, amelyben az összes karakter a *str2* által mutatott stringben is megtalálható. Más szavakkal, a **strspn( )** a *str1* által mutatott string első olyan karakterének indexét adja vissza, amely nem a *str2* által mutatott stringből kerül ki.

Vonatkozó függvények: **strpbrk( )**, **strrchr( )**, **strstr( )**, **strtok( )**.

■ *strstr*

```
#include <string.h>
char *strstr(const char *str1, const char *str2);
```

A **strstr( )** függvény egy olyan pointerrel tér vissza, amely a *str2* által mutatott string első előfordulásának kezdetére mutat a *str1* által mutatott stringben. Ha ilyen nincs, akkor nullpointert kapunk vissza.

Vonatkozó függvények: **strchr( )**, **strcspn( )**, **strspn( )**, **strtok( )**, **strrchr( )**.

■ *strtok*

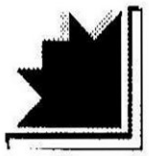
```
#include <string.h>
char *strtok(char *str1, const char *str2);
```

A **strtok( )** függvény a *str1* által mutatott string következő alapszimbólumára (token) mutató pointert ad vissza. A *str2* által mutatott stringben a tokeneket elválasztó karaktereket adjuk meg. Ha a függvény nem talál tokent, akkor nullpointert ad vissza.

Egy string alapszimbólumokra bontásához nincs másra szükség, mint a **strtok( )** többszöri meghívására. Az első hívást a magával a felbontandó stringgel végezzük. Az ezt követő hívások első argumentumaként nullpointert kell megadnunk (l. a **programozási tipp**-et). Ily módon az egész string tokenizálható.

Lehetőség van minden híváskor más-más elhatárolójel-halmazt használni.

Vonatkozó függvények: **strchr( )**, **strcspn( )**, **strpbrk( )**, **strrchr( )**, **strspn( )**.



Programozási tipp

*A **strtok( )** függvény egy olyan eszközt ad a kezünkbe, amellyel viszonylag kényelmesen bonthatunk fel egy stringet alapszimbólumokra.*
Az alábbi példaprogram az „Egy, kettő, három és négy." stringet tokenizálja:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char *p;

  p = strtok("Egy, kettő, három és négy.", ",");
  printf(p);
  do {
    p = strtok(NULL, ",. ");
    if(p) printf("|%s", p);
```

```
    } while(p);
    return 0;
}
```

A program az alábbiakat jeleníti meg a kimeneten:

Egy|kettő|három|és|négy

Vegyük észre, hogy a ***strtok( )****-ot először a tokenizálandó stringgel hívtuk meg, de a következő hívások alkalmával az első argumentum nullpointer (***NULL***) volt. A magyarázathoz tudnunk kell, hogy a függvény egy belső pointert használ, amely a felbontandó stringre mutat. Ha a függvény első argumentuma egy stringre mutat, akkor a belső pointer is annak a stringnek az elejére állítódik. Ha azonban az első argumentum* ***NULL****, akkor a* ***strtok( )*** *folytatja az előző string tokenizálást attól a ponttól, ahol az előző hívás után állt. Minden visszatérés előtt a belső pointer a leválasztott token végére kerül. Ily módon bonthatjuk részekre a* ***strtok( )*** *függvénnyel az egész stringet. Az is látható a programból, hogy az elhatárolójelek halmaza, a hívások között, megváltoztatható.*

■ *strxfrm*

```
#include <string.h>
size_t strxfrm(char *str1, const char *str2, size_t
                count);
```

A **strxfrm( )** függvény a *str2* pointer által mutatott string első *count* karakterét alakítja át előkészítve a stringet a **strcmp( )** függvényben való használatra. Az átalakított string a *str1* címére kerül. Az átalakítás utáni *str1*-gyel hívva meg a **strcmp( )**-t ugyanazt az eredményt kapjuk, mint az eredeti stringre mutató *str2*-vel meghívva a **strcoll( )** függvényt.

A **strxfrm( )** függvény visszatérési értékként az átalakított string hosszát adja.

Vonatkozó függvények: **strcoll( )**.

■ *tolower*

```
#include <ctype.h>
int tolower(int ch);
```

A **tolower( )** függvény a *ch* kisbetűs megfelelőjét adja vissza, ha *ch*-nak betűt adunk meg. Minden más esetben a függvény *ch*-t változatlanul adja vissza.

Vonatkozó függvények: **toupper( )**.

■ *toupper*

```
#include <ctype.h>
int toupper(int ch);
```

A **toupper( )** függvény a *ch* nagybetűs megfelelőjét adja vissza, ha *ch*-nak betűt adunk meg. Minden más esetben a függvény *ch*-t változatlanul adja vissza.

Vonatkozó függvények: **tolower( )**.

8. FEJEZET

A C matematikai függvényei

A C standard könyvtár számos matematikai függvényt tartalmaz. Ezek a következőképpen osztályozhatók:

Trigonometriai függvények
Hiperbolikus függvények
Exponenciális függvények
Vegyes célú függvények

A legtöbb matematikai függvényt a MATH.H fejállományon keresztül importálhatjuk. A matematikai függvények deklarálása mellett a fej három makrót is definiál: az **EDOM**-ot, az **ERANGE**-t és a **HUGE_VAL**-t. Ha egy matematikai függvény argumentuma nincsen az értelmezési tartományában, akkor egy implementációtól függő érték adódik vissza, továbbá az **errno** beépített globális integer változó értékül **EDOM**-ot kap. Amikor egy függvényvégrehajtás eredményeként egy túl nagy számot kapnánk, amely kívül esik a **double** által reprezentálható értékeken, akkor túlcsordulás történik, ekkor a függvény **HUGE_VAL** értékkel tér vissza. Továbbá az **errno**-nak adott **ERANGE** érték is az értékhatár megsértését jelzi. Alulcsordulás esetén a függvény 0-val tér vissza és az **errno** értékét **ERANGE**-re állítja.

A szögeket mindig radiánban kell megadni.

■ *acos*

```
#include <math.h>
double acos(double arg);
```

Az **acos( )** függvény az argumentum arkuszkoszínuszát adja vissza. A függvény argumentuma a –1 és 1 értékhatárok közé kell hogy essen. Ha nem így történik „értelmezési tartomány megsértése" hibát kapunk.

Vonatkozó függvények: **asin( )**, **atan( )**, **atan2( )**, **sin( )**, **cos( )**, **tan( )**, **sinh( )**, **cosh( )**, **tanh( )**.

■ *asin*

```
#include <math.h>
double asin(double arg);
```

Az **asin( )** függvény az argumentum arkuszszínuszát adja vissza. A függvény argumentuma a –1 és 1 értékhatárok közé kell hogy essen. Ha nem így történik „értelmezési tartomány megsértése" hibát kapunk.

Vonatkozó függvények: **acos( )**, **atan( )**, **atan2( )**, **sin( )**, **cos( )**, **tan( )**, **sinh( )**, **cosh( )**, **tanh( )**.

■ *atan*

```
#include <math.h>
double atan(double arg);
```

Az **atan( )** függvény az argumentum arkusztangensét adja vissza.

Vonatkozó függvények: **asin( )**, **acos( )**, **atan2( )**, **sin( )**, **cos( )**, **tan( )**, **sinh( )**, **cosh( )**, **tanh( )**.

■ *atan2*

```
#include <math.h>
double atan2(double y, double x);
```

Az **atan( )** függvény az *y/x* arkusztangensét adja vissza. Az argumentumok előjele határozza meg, hogy melyik negyedbeli eredményt kapunk.

Vonatkozó függvények: **asin( )**, **acos( )**, **atan( )**, **tan( )**, **cos( )**, **sin( )**, **sinh( )**, **cosh( )**, **tanh( )**.

■ *ceil*

```
#include <math.h>
double ceil(double num);
```

A **ceil( )** függvény a legkisebb olyan egészt (**double**-ként reprezentálva) adja vissza, amely nem kisebb, mint *num*. Például 1.02 argumentummal meghívva a **ceil( )** 2.0-t ad vissza, míg ugyanezt –1.02-vel téve –1.0 értékkel térne vissza.

Vonatkozó függvények: **floor( )**, **fmod( )**.

■ *cos*

```
#include <math.h>
double cos(double arg);
```

A **cos( )** függvény *arg* koszinuszát adja vissza. Az argumentumot radiánban kell megadni.

Vonatkozó függvények: **asin( )**, **acos( )**, **atan( )**, **atan2( )**, **tan( )**, **sin( )**, **sinh( )**, **cosh( )**, **tanh( )**.

■ *cosh*

```
#include <math.h>
double cosh(double arg);
```

A **cosh( )** függvény az *arg* koszinusz hiperbolikuszát adja vissza.

Vonatkozó függvények: **asin( )**, **acos( )**, **atan( )**, **atan2( )**, **tan( )**, **sin( )**, **sinh( )**, **cos( )**, **tanh( )**.

■ *exp*

```
#include <math.h>
double exp(double arg);
```

Az **exp( )** függvény az e (természetes alapú logaritmus alapja) számot emeli *arg* hatványra.

Vonatkozó függvények: **log( )**.

■ *fabs*

```
#include <math.h>
double fabs(double num);
```

A **fabs( )** függvény *num* abszolút értékét adja vissza.

Vonatkozó függvények: **abs( )**.

■ *floor*

```
#include <math.h>
double floor(double num);
```

A **floor( )** függvény a legnagyobb olyan egész számot (**double**-ként reprezentálva) adja vissza, amely nem nagyobb *num*-nál. Például 1.02 argumentummal meghívva a **floor( )** 1.0-t ad vissza, míg ugyanezt –1.02-vel téve –2.0 értékkel térne vissza.

Vonatkozó függvények: **ceil( )**, **fmod( )**.

■ *fmod*

```
#include <math.h>
double fmod(double x, double y);
```

Az **fmod( )** függvény *x/y* osztás maradékát adja vissza.

Vonatkozó függvények: **ceil( )**, **floor( )**, **fabs( )**.

■ *frexp*

```
#include <math.h>
double frexp(double num, int *exp);
```

Az **frexp( )** függvény a *num* számot felbontja egy 0.5 és 1 közé eső mantisszára, és egy egész exponensre úgy, hogy fennálljon a *num* = *mantissza*$*2^{exp}$. A függvény a mantisszát adja vissza, míg az exponens az *exp* mutatón keresztül lesz elérhető.

Vonatkozó függvények: **ldexp( )**.

■ *ldexp*

```
#include <math.h>
double ldexp(double num, int exp);
```

A **ldexp( )** függvény a $num*2^{exp}$ értéket adja vissza. Túlcsordulás esetén **HUGE_VAL** értékkel tér vissza.

Vonatkozó függvények: **fexp( )**, **modf( )**.

■ *log*

```
#include <math.h>
double log(double num);
```

A **log( )** függvény a *num* természetes alapú logaritmusát adja vissza. Ha *num* negatív „értelmezési tartomány megsértése" hibát kapunk, a 0 argumentum „értékhatár megsértése" hibát vált ki.

Vonatkozó függvények: **log10( )**.

■ *log10*

```
#include <math.h>
double log10(double num);
```

A **log10( )** függvény a *num* 10-es alapú logaritmusát adja vissza. Ha *num* negatív „értelmezési tartomány megsértése" hibát kapunk, a 0 argumentum „értékhatár megsértése" hibát vált ki.

Vonatkozó függvények: **log( )**.

■ *modf*

```
#include <math.h>
double modf(double num, double *i);
```

A **modf( )** függvény a *num* argumentumának megadott számot kettébontja az egészrészére és a törtrészére. Visszatérési értéke a törtrész, míg az egészrészt az *i* által mutatott változóba helyezi.

Vonatkozó függvények: **fexp( )**, **ldexp( )**.

■ *pow*

```
#include <math.h>
double pow(double base, double exp);
```

A **pow( )** függvény a *base*-t emeli az *exp* hatványra ($base^{exp}$). „Értelmezési tartomány megsértése" hibát kapunk, ha a *base* egyenlő nulla, és az *exp* negatív vagy ha a *base* negatív és *exp* nem egész szám. Túlcsorduláskor értékhatár megsértési hibát kapunk.

Vonatkozó függvények: **exp( )**, **log( )**, **sqrt( )**.

■ *sin*

```
#include <math.h>
double sin(double arg);
```

A **sin( )** függvény az *arg* szinuszát adja vissza. Az *arg* értékét radiánban kell megadni.

Vonatkozó függvények: **asin( )**, **acos( )**, **atan2( )**, **atan( )**, **tan( )**, **cos( )**, **sinh( )**, **cosh( )**, **tanh( )**.

■ *sinh*

```
#include <math.h>
double sinh(double arg);
```

A **sinh( )** függvény az *arg* szinusz hiperbolikuszát adja vissza.

Vonatkozó függvények: **asin( )**, **acos( )**, **atan2( )**, **atan( )**, **tan( )**, **cos( )**, **sin( )**, **cosh( )**, **tanh( )**.

■ *sqrt*

```
#include <math.h>
double sqrt(double num);
```

Az **sqrt( )** függvény a *num* négyzetgyökét adja vissza. Negatív argumentummal hívva meg „értelmezési tartomány megsértése" hibát kapunk.

Vonatkozó függvények: **exp( )**, **log( )**, **pow( )**.

■ *tan*

```
#include <math.h>
double tan(double arg);
```

A **tan( )** függvény az *arg* tangensét adja vissza. Az *arg* értékét radiánban kell megadni.

Vonatkozó függvények: **asin( )**, **acos( )**, **atan2( )**, **atan( )**, **cos( )**, **sin( )**, **sinh( )**, **cosh( )**, **tanh( )**.

■ *tanh*

```
#include <math.h>
double tanh(double arg);
```

A **tanh( )** függvény az *arg* tangens hiperbolikuszát adja vissza.

Vonatkozó függvények: **asin( )**, **acos( )**, **atan2( )**, **atan( )**, **tan( )**, **cos( )**, **sinh( )**, **cosh( )**, **tan( )**.

9. FEJEZET

A C idő-, dátum- és helykezelő függvényei

Ez a fejezet a C idő- és dátumkezelő függvényeket tárgyalja, valamint itt foglalkozunk azokkal a függvényekkel is, amelyekkel az egyes gépek földrajzi helyével kapcsolatos tulajdonságait állíthatjuk be, illetve kérdezhetjük le.

A C számos függvényt definiál, amelyekkel a rendszeridőt, illetve valamilyen relatív időt tudunk lekérdezni, illetve használni. Ehhez a TIME.H fej láthatóvá tétele szükséges. A TIME.H-ban három idővel kapcsolatos típus van definiálva: a **clock_t**, a **time_t**, és a **tm**. A **clock_t** és a **time_t** a rendszeridőt és -dátumot valamilyen módon egész számként reprezentálják. Ezt naptári időnek nevezik. A **tm** egy olyan struktúratípus, amely elemeire bontja a naptári időt. A **tm** struktúra a következőképpen van definiálva:

```
struct tm {
  int tm_sec;   /*másodperc, 0-61*/
  int tm_min;   /*perc, 0-59*/
  int tm_hour;  /*óra, 0-23*/
  int tm_mday;  /*a hónap hányadik napja, 1-31*/
  int tm_mon;   /*hónap januártól kezdődően, 0-11*/
  int tm_year;  /*évszám 1990-től számolva*/
  int tm_wday;  /*vasárnaptól eltelt napok száma, 0-6*/
  int tm_yday;  /*január 1-jétől eltelt napok száma,
                  0-365*/
  int tm_isdst; /*nyári időszámítás jelző*/
}
```

A **tm_isdst** pozitív, ha a nyári időszámítás van érvényben, 0 ha nem, és negatív érték esetén nincs elérhető információ. A naptári idő fenti alakja az ún. *lebontott idő*.

A TIME.H definiálja a **CLOCKS_PER_SEC** makrót is, amely a rendszer másodpercenkénti órajelét adja meg.

A földrajzi hely specifikus információk kezelését végző függvények a LOCALE.H fejben vannak definiálva.

A legtöbb C/C++ fordító további idő- és dátumkezelő függvényeket is támogat, ezért tanácsos áttanulmányozni a fordítóhoz mellékelt felhasználói kézikönyv idevonatkozó részét.

■ *asctime*

```
#include <time.h>
char *asctime(const struct tm *ptr);
```

Az **asctime( )** függvény a *ptr* által mutatott tm struktúrából merített adatokat, egy stringbe másolja az alábbiakban ismertetett módon, majd visszaad egy a stringre mutató pointert. Az adatok ilyen formában kerülnek a stringbe:

nap hónap dátum óra:perc:másodperc év\n\0

Példa:

```
Wed Jun 19 12:05:34 1999
```

Az **asctime( )**-nak átadott struktúra pointer általában egy előzetes **localtime( )** vagy **gmtime( )** hívásból származik.

Az **asctime( )** által használt puffer, amelyben a formázott eredménystring tárolódik, nem más, mint egy statikusan allokált karaktertömb, amely minden hívás alkalmával felülíródik. Ha a string tartalmát meg kívánjuk őrizni későbbi felhasználásra, akkor gondoskodnunk kell az átmásolásáról.

Vonatkozó függvények: **localtime( )**, **gmtime( )**, **time( )**, **ctime( )**.

■ *clock*

```
#include <time.h>
clock_t clock(void);
```

A **clock( )** függvény a hívó program becsült futási idejéről ad felvilágosítást. A visszaadott értéket a **CLOCK_PER_SEC** értékével elosztva, megkapjuk a futási időt másodpercben. A –1 visszatérési érték azt jelenti, hogy az idő valamilyen oknál fogva nem áll rendelkezésre.

Vonatkozó függvények: **time( )**, **asctime( )**, **ctime( )**.

■ *ctime*

```
#include <time.h>
char *ctime(const time_t *time);
```

A **ctime( )** függvény a naptári időre mutató *time* alapján az alábbi formában előállított stringre mutató pointert ad vissza:

nap hónap dátum óra:perc:másodperc év\n\0

A naptári időt a **time( )** függvény hívásával kaphatjuk meg.

A **ctime( )** által használt puffer, amelyben a formázott eredménystring tárolódik, nem más, mint egy statikusan allokált karaktertömb, amely minden hívás alkalmával felülíródik. Ha a string tartalmát meg kívánjuk őrizni későbbi felhasználásra, akkor gondoskodnunk kell az átmásolásáról.

Vonatkozó függvények: **localtime( )**, **gmtime( )**, **time( )**, **asctime( )**.

■ *difftime*

```
#include <time.h>
double difftime(time_t time2, time_t time1);
```

A **difftime( )** függvény a *time1* és *time2* időpontok közti különbséget (azaz a *time2-time1*-et) adja meg másodpercben.

Vonatkozó függvények: **localtime( )**, **time( )**, **asctime( )**.

■ *localeconv*

```
#include <locale.h>
struct lconv *localeconv(void);
```

A **localeconv( )** függvény egy **lconv** típusú struktúrát ad vissza, amely különböző országspecifikus adatokat tartalmaz bizonyos számformátumokkal kapcsolatban. A **lconv** struktúra definíciója a következő:

```
struct lconv {
  char *decimal_point;      /* tizedes vessző karakter
                               nem pénzösszegek esetén*/
  char *thousands_sep;      /* ezreseket elválasztó jel
                               nem pénzösszegek esetén*/
  char *grouping;           /* számjegycsoportosítás
                               nem pénzösszegek esetén*/
  char int_curr_symbol;     /* nemzetközi valutajel*/
  char *currency_symbol;    /* helyi valutajel*/
  char *mon_decimal_point; /* tizedes vessző karakter
                               pénzösszegekben*/
  char *mon_thousands_sep; /* ezreseket elválasztó jel
                               pénzösszegek esetén*/
  char *mon_grouping;       /* számjegycsoportosítás
                               pénzösszegek esetén*/
  char *positive_sign;      /* pozitív érték jelző
                               pénzösszegek esetén*/
  char *negative_sign;      /* negatív érték jelző
                               pénzösszegek esetén*/
  char int_frac_digits;     /* a tizedes vessző után álló
                               megjelenítendő számjegyek
                               száma pénzösszegek esetén
                               nemzetközi formátumot
                               használva*/
  char frac_digits;         /*a tizedes vessző után álló
                               megjelenítendő számjegyek
                               száma pénzösszegek esetén
                               helyi formátumot használva*/
```

```
    char p_cs_precedes;      /* 1, ha a valutajel a pozi-
                             tív pénzösszeg előtt áll,
                             0, ha a pénzösszeg
                             megelőzi a valutajelet*/
    char p_sep_by_space;     /*1, ha a valutajelet a
                             pozitív összegtől szóköz
                             választja el,
                             0 egyébként*/

    char n_cs_precedes;      /* 1, ha a valutajel a nega-
                             tív pénzösszeg előtt áll,
                             0, ha a pénzösszeg megelőzi
                             a valutajelet*/

    char n_sep_by_space;     /* 1, ha a valutajelet a pozi-
                             tív összegtől szóköz
                             választja el, 0 egyébként*/

    char p_sign_posn;        /*a pozitív érték jel
                             pozícióját határozza meg*/
    char n_sign_posn;        /*a negatív érték jel
                             pozícióját határozza meg*/
}
```

A **localeconv( )** függvény egy pointert ad vissza az **lconv** struktúrára, semmiképpen se változtassunk ezen struktúra tartalmán. A fordító dokumentációja további, implementáció specifikus információt tartalmazhat az **lconv** struktúráról.

Vonatkozó függvények: **setlocale( )**.

■ *localtime*

```
#include <time.h>
struct tm *localtime(const time_t *time);
```

A **localtime( )** függvény a *time* lebontott formátumú időreprezentációjára (**tm** struktúra) mutató pointerrel tér vissza. Az idő alatt ekkor a helyi

idő értendő. A *time* értéke legtöbbször egy korábbi **time( )** hívásból származik.

Az **localtime( )** által használt puffer, amelyben a *lebontott időt* tartalmazó struktúra tárolódik, statikusan allokált és minden hívás alkalmával felülíródik. Ha a string tartalmát meg kívánjuk őrizni későbbi felhasználásra, akkor gondoskodnunk kell az átmásolásáról.

Vonatkozó függvények: **gmtime( )**, **time( )**, **asctime( )**.

■ *mktime*

```
#include <time.h>
time_t mktime(struct tm *time);
```

A **mktime( )** függvény a *time* által mutatott lebontott idő struktúrát naptári idő formátumúvá konvertálja. A **tm_wday** és a **tm_yday** a függvény automatikusan beállítja, ezért ezeket fölösleges megadnunk a híváskor.

Ha **mktime( )** nem tudja a megadott információt érvényes naptári időként értelmezni, akkor –1-gyel tér vissza.

Vonatkozó függvények: **time( )**, **gmtime( )**, **asctime( )**, **ctime( )**.

Programozási tipp

*Az **mktime( )** függvény különösen hasznos lehet, amikor azt akarjuk kideríteni, hogy egy adott dátum, milyen napra esett vagy fog esni. Milyen nap lesz például 2012. január 12-ike? Ahhoz, hogy ezt megtudjuk, elég meghívni a **mktime( )** függvény, a pontos dátummal. A hívás után a **tm** struktúra **tm_wday** tagja tartalmazza majd a választ. Az alábbi programmal illusztráljuk ennek működését.*

```
/*Milyen napra esik 2012. január 12-ike*/
#include <stdio.h>
#include <time.h>

char day[][20] = {
    "vasárnap",
    "hétfő",
    "kedd",
```

```
        "szerda",
        "csütörtök",
        "péntek",
        "szombat"
};

int main(void)
{
  struct tm t;

  t.tm_mday = 12;
  t.tm_mon = 0;
  t.tm_year = 112;
  t.tm_hour = 0;
  t.tm_min = 0;
  t.tm_sec = 0;
  t.tm_isdst = 0;

  mktime(&t); /* a nap kiszámítása */

  printf("A keresett nap %s.\n", day[t.tm_wday]);

  return 0;
}
```

A program futtatásakor a ***mktime( )*** *automatikusan kiszámolja a keresett napot, ami jelen esetben csütörtök. Mivel a visszatérési érték érdektelen volt számunkra, ezért azt figyelmen kívül hagytuk.*

■ *setlocale*

```
#include <locale.h>
char *setlocale(int type, const char *locale);
```

A **setlocale( )** függvénnyel a futtatást végző gép földrajzi helyére jellemző paramétereket állíthatunk be és kérdezhetünk le. Ez fontos lehet, hiszen például Európában nem tizedes pontot, hanem tizedes vesszőt használunk.

Ha *locale* nullpointer, akkor a függvény az aktuális helyi információkat tartalmazó stringre mutató pointerrel tér vissza. Egyébként a **setlocale( )** megpróbálja a megadott *locale* string és a *type* értéke alapján a helyfüggő paramétereket beállítani.

A *type* az alábbi könyvtári makrók valamelyikével kell hogy megegyezzen:

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC-NUMERIC
LC-TIME

Az **LC_ALL** az összes helyi adatra utal. **LC_COLLATE** az **strcoll( )** függvény működésének beállításakor használatos. Az **LC_CTYPE**-pal a karakterkezelő függvények viselkedését állíthatjuk be. Az **LC_MONETARY** a valuta formátum meghatározásakor jut szerephez. Az **LC_NUMERIC**-kal a formázott be- és kiviteli függvények által használt tizedes vessző karaktert módosíthatjuk. Legvégül az **LC_TIME** makróval a **strftime( )** függvény viselkedését állíthatjuk be.

Az ANSI C szabvány definiál két lehetséges stringet, ami a *locale* paraméterben használható. Az első a „C", amely a C fordítás minimális környezeti követelményeit állítja be. A második a „ " üres string, amely az implementációfüggő környezeti beállításokat jelenti, ez egyben az alapértelmezés is. Minden más érték implementációfüggő és a hordozhatóságot befolyásolják. A **setlocale( )** függvény a *type* paraméternek megfelelő stringmutatót ad vissza.

Vonatkozó függvények: **localeconv( )**, **time( )**, **strcoll( )**, **strftime( )**.

■ *strftime*

```
#include <time.h>
size_t strftime(char *str, size_t maxsize, const char
                fmt, const struct tm *time);
```

A **strftime( )** függvény dátum, idő és egyéb információkat helyez a *str* által mutatott stringbe az *fmt* által mutatott stringben található formázó

parancsok által meghatározott formátumban. A **strftime( )** a lebontott időt tartalmazó *time* paramétert használja a kívánt adatok kinyerésére. A kimeneti stringbe maximálisan *maxsize* karakter kerülhet.

A **strftime( )** működése valamennyire hasonlít a **sprintf( )**-éhez abból a szempontból, hogy viselkedését ugyancsak formázó parancsokon keresztül befolyásolhatjuk, amelyek itt is százalék (%) jellel kezdődnek, a **strftime( )** is egy stringben tárolja a formattált kimenetet. A formázó parancsok feladata, hogy pontosan leírja, milyen módon jelenjenek meg a különböző dátum és idő információk a *str*-ben. A formázó string nem formázó parancs karakterei változtatás nélkül bekerülnek a *str*-be. A használt idő és dátum helyi időt és dátumot jelent. A formázó parancsokat az alábbi táblázat foglalja össze. Figyeljük meg, hogy a legtöbb parancs érzékeny a kis-, illetve nagybetűkre.

Parancs	*Helyettesítő adat*
%a	Hét napja szövegesen rövidítve
%A	Hét napja szövegesen
%b	Rövidített hónapnév
%B	Teljes hónapnév
%c	Szabvány dátum-idő string
%d	A hónap napja (1-31)
%H	Óra (0-23)
%I	Óra (1-12)
%j	Év napja (1-366)
%m	Hónap számként reprezentálva (1-12)
%M	Perc (0-59)
%p	Délután, illetve Délelőtt jelzése a helynek megfelelő formában
%S	Másodperc (0-61)
%U	Hét sorszáma, vasárnapot tekintve a hét első napjának (0-53)
%w	Hét napja, számként (0-6, vasárnap felel meg a 0-nak)
%W	Hét sorszáma, hétfőt tekintve a hét első napjának (0-53)
%x	Szabványos dátumstring
%X	Szabványos időstring
%y	Évszám század nélkül (0-99)
%Y	Teljes évszám
%Z	Időzóna név
%%	Százalék jel

A **strftime( )** függvény a *str* által mutatott stringbe helyezett karakterek számát adja vissza, ha nem történt hiba. Hiba esetén 0-t kapunk vissza.

Vonatkozó függvények: **time( )**, **localtime( )**, **gmtime( )**.

■ *time*

```
#include <time.h>
time_t time(time_t *time);
```

A **time( )** függvény a rendszer aktuális naptári idejét adja vissza. Ha ez nem elérhető, akkor a visszaadott érték -1.

A **time( )** függvény hívható a nullpointerrel, illetve egy **time_t** típusú változóra mutató pointerrel. Az utóbbi esetben a változó is megkapja a naptári időt.

Vonatkozó függvények: **localtime( )**, **gmtime( )**, **strftime( )**, **ctime( )**.

10. FEJEZET

A C dinamikus memóriafoglaló és -felszabadító függvényei

Ez a fejezet a C dinamikus memóriakezelő függvényeit tárgyalja. A fókuszban a **malloc( )** és a **free( )** függvények állnak. Minden **malloc( )** hívással a szabad memória egy részét foglaljuk le. A **free( )** hatása pontosan ellenkező: a rendszernek visszaad memóriát. Azt a szabad memóriatartományt, amelyből lefoglalhatunk, *heap*-nek nevezik. A dinamikus memória allokáló függvények prototípusai a STDLIB.H fejben találhatók.

Minden C/C++ fordító támogatja a következő négy dinamikus allokálásra, illetve memóriafelszabadításra használt függvényt: **calloc( )**, **malloc( )**, **free( )**, **realloc( )**. Valószínű azonban, hogy az olvasó fordítója ezen függvények más változatait is támogatja, amelyek az adott programozói környezetből fakadó sajátosságokat is figyelembe veszik. Célszerű a fordítóhoz mellékelt dokumentáció ide vonatkozó részeit áttanulmányozni.

Annak ellenére, hogy a C++ támogatja a fenti memóriakezelő műveleteket, ezek használata ritka egy tipikus C++ programban. Ennek az az oka, hogy a C++ speciális allokáló műveletekkel rendelkezik: a **new**-val és a **delete**-tel. Használatuknak számos előnye van. Először is a **new** automatikusan az adattípusnak megfelelő méretű memóriaterületet foglal le. Másodszor a helyes típusú pointerrel tér vissza. Harmadszor mind a **new**, mind pedig a **delete** túlterhelhetők. Ezen vitathatatlan előnyök miatt, használatuk ajánlott a C++ programokban. (A **new** és **delete** műveletek tárgyalására az 5. fejezetben került sor.)

■ *calloc*

```
#include <stdlib.h>
void *calloc(size_t num, size_t size);
```

A **calloc( )** *num*size* méretű memóriát foglal le. Azaz a függvény elegendő memóriát foglal le egy olyan *num* hosszúságú tömbnek, amely *size* méretű elemeket tartalmaz.

A **calloc( )** függvény az allokált terület első bájtjára mutató pointert ad vissza. Ha az allokálási kérelem memória hiánya miatt nem teljesíthető, a **calloc( )** nullpointert ad vissza. Ezért fontos, hogy mindig ellenőrizzük, nem kaptunk-e nullpointert, mielőtt a mutatóra hivatkoznánk.

Vonatkozó függvények: **free( )**, **malloc( )**, **realloc( )**.

■ *free*

```
#include <stdlib.h>
void free(void *ptr);
```

A **free( )** függvény visszaadja a *ptr* által mutatott memóriaterületet a heapnek, amely aztán – egy későbbi allokáció által – ismét használható lesz.

Nagyon fontos, hogy a **free( )** argumentumaként kizárólag olyan pointert adjunk meg, amelyet korábban valamely dinamikus allokáló függvénnyel foglaltunk le (**malloc( )** vagy **alloc( )**). Érvénytelen mutató argumentumként történő használata a memóriakezelő mechanizmust felboríthatja, ami a rendszer összeomlását eredményezheti.

Vonatkozó függvények: **calloc( )**, **malloc( )**, **realloc( )**.

■ *malloc*

```
#include <stdlib.h>
void *malloc(size_t size);
```

A **malloc( )** függvény a heapből *size* méretű memóriaterületet allokál, és annak első bájtjára mutató pointerrel tér vissza. Ha memória hiányában az allokáció nem hajtható végre, a függvény nullpointert ad vissza. Ezért fontos, hogy mindig ellenőrizzük, nem kaptunk-e nullpointert, mielőtt a mutatóra hivatkoznánk. Nullpointer használata a rendszer összeomlásához vezethet.

Vonatkozó függvények: **free( )**, **realloc( )**, **calloc( )**.

Programozási tipp

Ha 8086-os processzorcsaládra (pl. 80486 vagy Pentium) fejlesztünk 16 bites programokat, akkor a fordító további allokáló függvényeket bocsát a rendelkezésünkre, amely ezen processzorok 16 bites módja által használt szegmentált memóriakezelést támogatják. Például lesznek olyan függvények, amelyek segítségével a FAR heapből (az a heap, amely kívül esik az alapértelmezett adatszegmensen) foglalhatunk memóriát, és olyanok is, amelyekkel egy szegmensnél nagyobb méretű memóriát is igényelhetünk, illetve felszabadíthatunk.

■ *realloc*

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

A **realloc( )** függvénnyel a korábban lefoglalt a *ptr* által mutatott memóriatartomány méretét változtathatjuk *size* méretűre. A *size* egyaránt lehet nagyobb és kisebb, mint az eredeti méret. Visszatérési értékként egy mutatót kapunk, amely a kívánt méretű memóriaterületre mutat. Elképzelhető, hogy ez nem egyezik meg a paraméterként átadott mutatóval, mert a függvénynek el kellett mozgatnia az eredeti tartományt ahhoz, hogy méretét megnövelhesse. Ebben az esetben a régi blokk tartalma automatikusan átmásolódik az új területre, azaz nem vesztünk el adatokat. Ha *ptr* nullpointer, akkor a **realloc( )** egyszerűen lefoglal *size* bájtnyi memóriát és visszaad egy arra mutató pointert. Amennyiben a *size* nulla, akkor a *ptr* által mutatott memória felszabadul.

Ha nincs elegendő szabad memória az allokálás végrehajtására, akkor nullpointert kapunk vissza, de az eredeti blokk érintetlen marad.

Vonatkozó függvények: **free( )**, **malloc( )**, **calloc( )**.

11. FEJEZET

Vegyes C függvények

Az ebben a fejezetben tárgyalásra kerülő függvények nem könnyen illeszthetők bele más kategóriákba. Ide tartoznak különböző konverziós, argumentumfeldolgozó, rendező, kereső és véletlenszám-generáló függvények.

A fejezetben szereplő legtöbb függvény használathoz az include blokkban az STDLIB.H fej nevét kell feltüntetnünk. Ez a fejben két új típusdefiníciót is tartalmaz: a **div_t**-t és az **ldiv_t**-t, ilyen típusúak rendre a **div( )** és az **ldiv( )** függvények visszatérési értékei. A **size_t** definíciója is megtalálható itt, amelyet a **sizeof( )** függvény használ visszatérési típusaként. Az alábbi makrók is a STDLIB.H fejben vannak definiálva:

Makró	*Jelentése*
NULL	Nullpointer
RAND_MAX	A **rand( )** függvény által visszaadható legnagyobb érték
EXIT_FAILURE	A hívó folyamatnak átadandó érték, ha a program terminálása nem sikeres
EXIT_SUCCESS	A hívó folyamatnak visszaadott érték a program sikeres terminálása esetén.

Külön jelezni fogjuk, ha egy függvény használatához az STDLIB.H-tól különböző fejet kell az include utasításban megadnunk.

■ *abort*

```
#include <stdlib.h>
void abort(void);
```

Az **abort( )** függvény a program azonnali abnormális terminálását idézi elő. Általában a fájlpufferek nem ürítődnek ki (így az állományok nem frissítődnek). Egyes környezetekben az **abort( )** egy implementáció-definiált értéket ad vissza a hívó folyamatnak (általában az operációs rendszernek), amely jelzi a hibát.

Vonatkozó függvények: **exit( )** és **atexit( )**.

■ *abs*

```
#include <stdlib.h>
int abs(int num);
```

Az **abs( )** függvény a *num* egész abszolút értékét adja vissza.

Vonatkozó függvények: **labs( )**.

■ *assert*

```
#include <assert.h>
void assert(int exp);
```

Az **assert( )** makrót, amelyet az ASSERT.H fej definiál, hibával kapcsolatos információkat ír a **stderr**-ra, majd megszakítja a program futását, ha az *exp* kifejezés értéke 0. Minden más esetben az **assert( )** nem csinál semmit. Habár a függvény pontos kimenete implementációfüggő, sok fordító az alábbi üzenetet használja:

Assertion failed: <kifejezés>, file<fájl>, sor<sorszám>

Az **assert( )** makrót általában annak megállapítására használjuk, hogy a programunk jól fut-e, *exp* argumentumként pedig olyan kifejezést adunk meg, amely csak abban az esetben lesz nullától különböző, ha nem történt hiba.

A hibakeresés után nem szükséges törölnünk az **assert( )** utasításokat a forráskódból, mert a **NDEBUG** makró definiálásával (bárminek definiálhatjuk) az **assert( )** függvényeket a fordító átugorja.

Vonatkozó függvények: **abort( )**.

■ *atexit*

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Az **atexit( )** függvény hatására a program normális termináláskor a *func* pointer által mutatott függvény végrehajtódik. Az **atexit( )** 0-val tér vissza, ha a függvényt sikeresen regisztrálta, mint termináláskor végrehajtandó függvényt, egyébként nullától különböző értéket kapunk vissza.

Legalább 32 terminálási függvényt adhatunk meg, meghívásuk a regisztrálással ellenkező sorrendben történik.

Vonatkozó függvények: **exit( )**, **abort( )**.

■ *atof*

```
#include <stdlib.h>
double atof(const char *str);
```

Az **atof( )** függvény a *str* által mutatott stringet konvertálja **double** típusúvá. Ehhez természetesen szükséges, hogy a string egy érvényes lebegőpontos számot tartalmazzon. Ha ez nem áll fenn, akkor a visszatérési érték definiálatlan. A stringben megadott szám végét jelenti bármilyen olyan karakter, amely nem lehet része egy érvényes lebegőpontos számnak. Ilyenek az üreskarakterek, írásjelek (kivéve a pont), és az összes betű, kivéve az „E"-t és az „e"-t. Például ha az **atof( )** függvényt a „100.00HELLÓ" stringre mutató pointerrel hívjuk meg, 100.00-t kapunk vissza.

Vonatkozó függvények: **atoi( )**, **atol( )**.

■ *atoi*

```
#include <stdio.h>
int atoi(const char *str);
```

Az **atoi( )** függvény a *str* által mutatott stringet konvertálja **int** típusúvá. Ehhez természetesen szükséges, hogy a string egy érvényes egész

számot tartalmazzon. Ha ez nem áll fenn, akkor a visszatérési érték definiálatlan; a legtöbb implementációban azonban 0-t kapunk vissza. A stringben megadott szám végét jelenti bármilyen olyan karakter, amely nem lehet része egy érvényes egész számnak. Ilyenek az üres karakterek, írásjelek és az összes betű. Ha az **atoi( )**-t az „123.23" stringre mutató pointerrel hívjuk meg, akkor a 123 értéket kapjuk vissza, a „.23" rész a terminálo pont miatt nem játszik szerepet a végeredményben.

Vonatkozó függvények: **atof( )**, **atol( )**.

■ *atol*

```
#include <stdlib.h>
long atol(const char *str);
```

Az **atol( )** függvény az *str* által mutatott stringet konvertálja **long** típusúvá. Ehhez természetesen szükséges, hogy a string egy érvényes hosszú integert tartalmazzon. Ha ez nem áll fenn, akkor a visszatérési érték definiálatlan a legtöbb implementációban azonban 0-t kapunk vissza.

A stringben megadott szám végét jelenti bármilyen olyan karakter, amely nem lehet része egy érvényes egész számnak. Ilyenek az üreskarakterek, írásjelek, és az összes betű. Ha az **atol( )**-t az „123.23" stringre mutató pointerrel hívjuk meg, akkor a 123L értéket kapjuk vissza, a „.23" rész a terminálo pont miatt nem játszik szerepet a végeredményben.

Vonatkozó függvények: **atof( )**, **atoi( )**.

■ *bsearch*

```
#include <stdlib.h>
void *bsearch(const void *key, const void *buf, size_t
              num,   size_t size, int (*compare) (const
              void *, const void *));
```

A **bsearch( )** függvény logaritmikus keresést végez a *buf* által mutatott rendezett tömbben, és a *key* kulccsal megegyező első elemre mutató poin-

tert ad vissza. A tömbben található elemek számát a *num* paraméterben adjuk meg, az egyes elemek méretét a *size* tartalmazza.

A *compare* pointer egy olyan függvényre mutat, amely egy elem kulccsal való egyezését definiálja. A *compare* függvénynek formálisan így kell kinéznie:

int *f_név* (const void **arg1*, const void **arg2*);

A visszatérési értékeket a **bsearch( )** így értelmezi:

Visszatérési érték	*Jelentés*
Negatív	*arg1* kisebb, mint *arg2*
0	*arg1* egyenlő *arg2*-vel
Pozitív	*arg1* nagyobb, mint *arg2*

A helyes működés előfeltétele, hogy az elemek növekvő sorrendben kövessék egymást a tömbben (a legkisebb címen kell szerepelnie a legkisebb elemnek).

Ha semelyik elem sem egyezik meg a kulccsal, akkor nullpointert kapunk vissza.

Vonatkozó függvények: **qsort( )**.

■ *div*

```
#include <stdlib.h>
div_t div(int numerator, int denominator);
```

A **div( )** függvény a *numerator/denominator* művelet hányadosát és maradékát adja vissza **div_t** struktúra formájában.

A **div_t** típust a STDLIB.H fej definiálja és legalább az alábbi két mezővel rendelkezik:

```
int quot; /*hányados*/
int rem;  /*maradék*/
```

Vonatkozó függvények: **ldiv( )**.

■ *exit*

```
#include <stdlib.h>
void exit(int exit_code);
```

Az **exit( )** függvény a program azonnali normál terminálását okozza.

Az *exit_code* – ha ezt a környezet támogatja – átadódik a hívó folyamatnak, amely gyakran az operációs rendszert jelenti. Ha az *exit_code* 0, vagy **EXIT_SUCCES**, akkor normális programterminálás történt. Nullától különböző érték vagy **EXIT_FAILURE** implementációtól függően különböző hibákat jeleznek.

Vonatkozó függvények: **atexit( )**, **abort( )**.

■ *getenv*

```
#include <stdlib.h>
char *getenv(const char *name);
```

A **getenv( )** függvénnyel az aktuális programkörnyezetről kaphatunk információkat. A függvény a *name* által mutatott implementációfüggő információs tábla alapján nyeri az adatokat.

A program környezetén olyan dolgokat értünk, mint elérési utak, elérhető eszközök stb. Ezen adatok pontos természete implementációfüggő, így tanácsos a fordító kézikönyvének ide vonatkozó részét áttanulmányozni.

Ha a hívást olyan argumentummal végezzük, amelyet a **getenv( )** függvény nem tud környezeti információként értelmezni, a visszakapott mutató nullpointer lesz.

Vonatkozó függvények: **system( )**.

■ *labs*

```
#include <stdlib.h>
long labs(long num);
```

A **labs( )** függvény a *num* abszolút értékét adja vissza.

Vonatkozó függvények: **abs( )**.

■ *ldiv*

```
#include <stdlib.h>
ldiv_t ldiv(long numerator, long denominator);
```

Az **ldiv( )** függvény a *numerator/denominator* művelet hányadosát és maradékát adja vissza **div_t** struktúra formájában.

A **div_t** típust a STDLIB.H fej definiálja és legalább az alábbi két mezővel rendelkezik:

```
long quot; /*hányados*/
long rem; /*maradék*/
```

Vonatkozó függvények: **div( )**.

■ *longjmp*

```
#include < setjmp.h>
void longjmp(jmp_buf envbuf, int status);
```

A **longjmp( )** függvény hatására vezérlés a legutóbbi **setjmp( )** utasítás utáni sorra kerül. Ez a függvénypár eszközt kínál két függvény közti ugrás megvalósításához. Vegyük észre, hogy a SETJMP.H fej használata szükséges.

A **longjmp( )** függvény a vermet helyezi az *enbuf* paraméterben megadott állapotba, amelyet egy előzetes **setjmp( )** hívással nyerhetünk. A verem átállításának eredményeként azt „hitetjük" el a programmal, hogy soha nem is hagytuk el a **setjmp( )** hívást tartalmazó függvényt.

Az *envbuf* puffer típusa **jmp_buf**, amelyet a SETJMP.H fej definiál. A puffer beállítását a **setjmp( )** hívással végezhetjük a megfelelő **longjmp( )** előtt.

A *status*-t a **setjmp( )** visszatérési értéke kapja meg, vizsgálatával deríthetjük ki, a **longjmp( )** hívás helyét. Egyedül a nulla érték nem megengedett.

A **longjmp( )** függvényt leggyakrabban (és szinte kizárólag) akkor használjuk, amikor egy mélyen beágyazott rutinból kívánunk visszatérni hiba esetén.

Vonatkozó függvények: **setjmp( )**.

qsort

```
#include <stdlib.h>
void qsort(void *buf, size_t num, size_t size, int
          (*compare) const void *, const void *));
```

A **qsort( )** függvény a *buf* által mutatott tömb rendezését végzi az ún. Quicksort módszerrel (az algoritmus C. A. R. Hoare nevéhez fűződik). A Quicksortot tartják a leghatékonyabb általános célú rendező algoritmusnak. A függvényből való visszatéréskor a tömbünk rendezett lesz. A tömb elemeinek a számát a *num* paraméter tartalmazza, míg az egyes elemek méretét (bájtokban) a *size* határozza meg.

A *compare* pointer által mutatott függvény a tömb két elemének az összehasonlítását végzi (eszerint történik a rendezés). A *compare* függvény fejének a következőképpen kell kinézni:

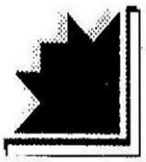
int *f_név*(const void **arg1*, const void **arg2*);

A visszatérési értékekhez rendelt jelentéseket az alábbi táblázat foglalja össze:

Visszatérési érték	*Jelentés*
Negatív szám	*arg1* kisebb, mint *arg2*
0	*arg1* egyenlő *arg2*-vel
Pozitív szám	*arg1* nagyobb, mint *arg2*

Az eredménytömb elemei növekvő (a *compare* szerint) sorrendben követik egymást (a legkisebb memóriacímen lesz a legkisebb elem).

Vonatkozó függvények: **bsearch( )**.



Programozási tipp

Amennyiben azt szeretnénk elérni, hogy a **qsort( )** *csökkenő sorrendbe rendezze az elemeket, elegendő az összehasonlító függvény visszatérési értékeit másképp megadni. A következő táblázat segíthet:*

Visszatérési érték	*Jelentés*
Negatív szám	*arg1* nagyobb, mint *arg2*
0	*arg1* egyenlő *arg2*-vel
Pozitív szám	*arg1* kisebb, mint *arg2*

Hasonlóképpen, ha a **bsearch( )** *függvényt olyan tömbre kívánjuk alkalmazni, amelyben az elemek csökkenő sorrendben vannak, akkor ugyancsak az összehasonlító függvényt ellentétét kell használnunk.*

■ *raise*

```
#include <signal.h>
int raise(int signal);
```

A **raise( )** függvény a *signal* paraméterben meghatározott szignált küldi a futtatóprogramnak. Sikeres végrehajtása esetén 0-t kapunk vissza, egyébként egy nullától különböző értéket. A SIGNAL.H fejállomány használata szükséges.

Az alábbi szignálok definiáltak a standard ANSI C-ben. Természetesen a fordító ezeken kívül további jeleket is támogathat.

Makró	*Jelentés*
SIGABRT	Terminálási hiba
SIGFPE	Lebegőpontos hiba
SIGNILL	Hibás utasítás
SIGINT	A felhasználó Ctrl-C-t nyomott
SIGSEGV	Memória-hozzáférési hiba
SIGTERM	Program terminálása

Vonatkozó függvények: **signal( )**.

■ *rand*

```
#include <stdlib.h>
int rand(void);
```

A **rand( )** függvény pszeudóvéletlen számok egy sorozatát generálja. Minden meghívása alkalmával egy 0 és **RAND_MAX** közötti egész számot kapunk vissza.

Vonatkozó függvények: **srand( )**.

■ *setjmp*

```
#include <setjmp.h>
int setjmp(jmp_buf envbuf);
```

A **setjmp( )** függvény elmenti a rendszerverem aktuális állapotát az *envbuf* nevű pufferba, a **longjmp( )** függvénnyel való későbbi használatra. A SETJMP.H fej importálását igényli.

A **setjmp( )** függvény 0-át ad vissza. A **longjmp( )** azonban végrehajtása során átad egy argumentumot a **setjmp( )**-nak, amelynek értéke (mindig nullától különböző) a **setjmp( )** értékének fog látszani. (További információk találhatók a **longjmp( )**-ról szóló részben.)

Vonatkozó függvények: **longjmp( )**.

■ *signal*

```
#include <signal.h>
void(*signal(int signal, void (*func) (int))) (int);
```

A **signal( )** függvény regisztrálja a *func* által mutatott függvényt a *signal* paraméterben megadott jel jelkezelőjeként. Ez más szóval azt jelenti, hogy ez a függvény kerül végrehajtásra, amikor a program a *signal* jelet kapja.

A *func* lehet egy programozó által definiált jelkezelő függvényre mutató pointer, vagy az alábbi SIGNAL.H fejben definiált makrók valamelyike:

Makró	*Jelentése*
SIG_DFL	Az alapértelmezett jelkezelő használata
SIG_IGN	Figyelmen kívül hagyja a jelet

Ha egy függvény címét adjuk át, akkor a jel megjelenésekor a megadott jelkezelő függvény kerül végrehajtásra.

Sikeres működéskor a **signal( )** az előzőleg definiált jelkezelő függvény címét adja vissza. Hiba esetén **SIG_ERR** -t (SIGNAL.H-ban definiált) kapunk vissza.

Vonatkozó függvények: **raise( )**.

■ *srand*

```
#include <stdlib.h>
void srand(unsigned seed);
```

A **srand( )** függvény a **rand( )** függvénnyel előállítandó véletlenszámsorozat kezdőpontját adja meg. (A **rand( )** függvény pszeudovéletlen számokat ad vissza.)

A **srand( )** lehetővé teszi, hogy több program egyidejű futtatásánál az egyes programok különböző pszeudovéletlen számsorozatot állítsanak elő azáltal, hogy más és más kezdőpontokat adunk meg. Vagy ellenkezőleg, ha ugyanazt a pszeudovéletlen számsorozatot kívánjuk generálni többször egymás után, akkor a **srand( )** függvényt ugyanazzal a *seed* paraméterrel kell meghívnunk, minden új sorozat kezdésével.

Vonatkozó függvények: **rand( )**.

■ *strtod*

```
#include <stdlib.h>
double strtod(const char *start, char **end);
```

A **strtod( )** függvény a *start* által mutatott stringként megadott számot **double** típusúvá konvertálja, és ennek eredményét visszaadja. A **strtod( )** függvény a következőképpen működik: Először is a *start* által mutatott

string elején található üres karakterekre a függvény nem érzékeny, azaz figyelmen kívül hagyja őket. Másodszor a számot alkotó karakterek beolvasásra kerülnek egészen addig, amíg egy olyan karakter nem kerül sorra, amely nem lehet része egy lebegőpontos számnak, ekkor a beolvasás befejeződik. Ilyen karakterek az írásjelek (kivéve a pont), az üres karakterek, és minden betű az „e" és „E" kivételével. Legvégül a függvény az *end* mutatót a string fel nem dolgozott részének első karakterére állítja. Például, ha a **strod( )**-ot a „100.00 macska" stringgel hívjuk meg, az *end* mutató a „macska" előtti szóközre mutat.

Ha nincs konverzió, a **strod( )** visszatérési értéke 0. Túlcsorduláskor a **HUGE_VAL** vagy **-HUGE_VAL** (attól függően, hogy pozitív vagy negatív túlcsordulás volt) értéket kapunk, és az **errno** globális változó **ERANGE** értéket kap, jelezve a túlcsordulási hibát. Alulcsordulás esetén 0-t kapunk vissza, a hibára most is az **errno ERANGE** értéke utal.

Vonatkozó függvények: **atof( )**.

■ *strtol*

```
#include <stlib.h>
long strtol(const char *start, char **end, int radix);
```

A **strtol( )** függvény a *start* által mutatott stringben tárolt számot **long** típusúvá konvertálja és az eredményt visszaadja. Azt az információt, hogy a szám milyen számrendszerben van ábrázolva, *radix* paraméteren (ez a számrendszer alapja) keresztül tudathatjuk a függvénnyel. Ha a *radix* értéke 0, akkor az alapot a konstansoknál látott szabályok határozzák meg. A *radix* lehet 0 vagy egy 2-től 36-ig terjedő szám.

A **strtol( )** függvény a következőképpen működik: először is a *start* által mutatott string elején található üreskarakterekre a függvény nem érzékeny, azaz figyelmen kívül hagyja őket. Másodszor a számot alkotó karakterek beolvasásra kerülnek egészen addig, amíg egy olyan karakter nem kerül sorra, amely nem lehet része egy hosszú egésznek, ekkor a beolvasás befejeződik. Ezek közé tartoznak pl. az írásjelek és az üres karakterek. Legvégül a függvény az *end* mutatót a string fel nem dolgozott részének első karakterére állítja. Például, ha a **strtol( )**-t a „100 macska" stringgel hívjuk meg, a 100L értéket kapjuk vissza, és az *end* mutató a „macska" előtti szóközre fog mutatni.

Ha a string nem konvertálható az előbbi módon, akkor a **strtol( )** 0-t ad vissza. Túlcsordulás, illetve alulcsordulás esetén **LONG_MAX**-ot, illetve **LONG_MIN**-t kapunk vissza, az **errno** globális változó pedig **ERANGE** lesz.

Vonatkozó függvények: **atol( )**.

■ *strtoul*

```
#include <stdlib.h>
unsigned long strtoul(const char *start, char **end,
                      int radix);
```

A **strtoul( )** függvény a *start* által mutatott stringben tárolt számot **unsigned long** típusúvá konvertálja és az eredményt visszaadja. Azt az információt, hogy a szám milyen számrendszerben van ábrázolva, *radix* paraméteren (ez a számrendszer alapja) keresztül tudathatjuk a függvénnyel. Ha a *radix* értéke 0, akkor az alapot a konstansoknál látott szabályok határozzák meg. A *radix* lehet 0 vagy egy 2-től 36-ig terjedő szám.

A **strtoul( )** függvény a következőképpen működik: Először is a *start* által mutatott string elején található üres karakterekre a függvény nem érzékeny, azaz figyelmen kívül hagyja őket. Másodszor a számot alkotó karakterek beolvasásra kerülnek egészen addig, amíg egy olyan karakter nem kerül sorra, amely nem lehet része egy hosszú egésznek, ekkor a beolvasás befejeződik. Ezek közé tartoznak pl. az írásjelek és az üres karakterek. Legvégül a függvény az *end* mutatót a string fel nem dolgozott részének első karakterére állítja. Például, ha a **strtoul( )**-t a „100 macska" stringgel hívjuk meg, a 100L értéket kapjuk vissza, és az *end* mutató a „macska" előtti szóközre fog mutatni.

Ha az eredmény nem reprezentálható előjel nélküli hosszú egészként, akkor a függvény az **ULONG_MAX** értékkel tér vissza és az **errno** globális változót **ERANGE**-ra állítja. Ha a stringen nem végezhető el a konverzió, 0-t kapunk vissza.

Vonatkozó függvények: **strtol( )**.

■ *system*

```
#include <stdlib.h>
int system(const char *str);
```

A **system( )** függvény a *str* által mutatott stringet parancsként átadja az operációs rendszer parancsértelmezőjének.

Ha a függvényt nullpointer argumentummal hívjuk meg, akkor a visszatérési értékéből a parancsértelmező jelenlétére, illetve hiányára következtethetünk. Ha 0-t kapunk, akkor van parancsértelmező, egyébként nincsen. (Vannak olyan rendszerek, amelyekben **system( )**-mel néhány C/C++ parancsot végre tudunk hajtatni anélkül, hogy rendelkeznének parancsértelmezővel, így maga az a tény, hogy néhány parancs működik, még nem jelenti azt, hogy lenne parancsértelmező.) A **system( )** visszatérési értéke implementációfüggő. A legtöbb fordítóval készített kód 0-t ad vissza, ha sikeres volt a parancs végrehajtása, egyébként nullától különböző értéket kapunk.

Vonatkozó függvények: **exit( )**.

■ *va_arg, va_start, va_end*

```
#include <stdarg.h>
type va_arg(va_list argptr, type);
void va_end(va_list argptr);
void va_start(va_list argptr, last_parm);
```

A **va_arg( )**, a **va_start( )** és a **va_end( )** makrók teszik lehetővé, hogy változó számú argumentum legyen átadható egy függvénynek. Az ilyen változó számú argumentumot elfogadó függvényekre egy jellemző példa az **fprint( )**. A **va_list** típust az STDARG.H definiálja.

Az általános recept ilyen függvények készítéséhez a következő: a függvénynek rendelkeznie kell legalább egy (lehet több is) ismert paraméterrel, amely megelőzi a változó hosszúságú paraméterlistát. A legutolsó ismert paraméter a *last_parm*, ezt használjuk a **va_start** második paramétereként. Ahhoz, hogy változó hosszú paraméterlistát használhassunk, először az *argptr* argumentummutatót szükséges inicializálnunk a

va_start( ) függvény meghívásával. Ezután a **va_arg( )** függvény hívásával megkaphatjuk sorra a paramétereket és a *type* a következő paraméter típusát fogja megadni. Végül a paraméterek kiolvasása után gondoskodni kell a verem visszaállításáról, amit a **va_end( )** függvénnyel végezhetünk. Az utóbbi lépés elmulasztása a program működésképtelenné válásához vezethet.

Vonatkozó függvények: **vprintf( )**.

Programozási tipp

A **va_start( )**, **va_end( )** *és a* **va_arg( )** *használatát legkönnyebben példákon keresztül sajátíthatjuk el. Az alább közölt példaprogram a* **sum_series( )** *függvényt definiálja, amely egy számsorozat elemeit adja össze. Első argumentumaként a többi argumentum számát adjuk át. A függvényt a program az alábbi sor n=5-höz tartozó részletösszegét számolja ki:*

$$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+ \quad \cdots \quad +\frac{1}{2^n}$$

A képernyőn a „0.968750" szám lesz látható.

```
/*Példa változó méretű argumentum
  listát használó függvényre: sorozat összege*/
#include <stdio.h>
#include <stdarg.h>

double sum_series(int, ...);

int main(void)
{
  double d;
  d= sum_series(5, 0.5, 0.25, 0.125, 0.0625,
                0.03125);

  printf("Az összeg %f\n",d);

  return 0;
}
```

```
double sum_series(int num, ...)
{
  double sum = 0.0, t;
  va_list argptr;

  /*argptr inicializálása*/
  va_start(argptr, num);

  /*összeg képzése*/
  for(; num; num--) {
    t = va_arg(argptr, double);
    sum += t;
  }

  /*befejezés*/
  va_end(argptr);
  return sum;
}
```

12. FEJEZET

A régi stílusú C++ I/O rendszer

Már említettük, hogy a C++ a C egész könyvtárát tartalmazza, így a C I/O rendszerét is. A C++-ban a C lehetőségei bővültek új osztályalapú, objektumorientált I/O rendszerrel, amelyet az *iostream* könyvtár fog össze. A legtöbb C++ programozó szívesebben használja az *iostream* könyvtárat, mint a régi C-alapú I/O rendszert.

E könyv születésekor az *iostream* könyvtárnak két változata volt forgalomban: egy régebbi, amely a C++ eredeti specifikációjának felel meg, és egy újabb, amely a ANSI C++ szabványhoz készült. Ma a legtöbb C++ fordító mind az új, mind pedig a régi stílusú *iostream* könyvtárat támogatja. Szerencsére a két könyvtár függvényei nagy részben egyformán működnek. Ha az egyik használatát megtanultuk, nem lehet gond a másik függvényeinek a megértése sem. A két könyvtár között azonban nem egy fontos különbség van.

Először, az eredeti *iostream* osztályok a globális névtér alatt voltak definiálva. Az új standard könyvtárban azokat az **std** névtér tartalmazza.

Másodszor, az új *iostream* könyvtár egy összefüggő, komplex generic-osztály- és függvénygyűjteményre épül. A régi típusú könyvtár egy kevésbé bonyolult, nem sablonosított osztályhierarchiát használ. Szerencsére a használt osztályok nevei nem különböznek a két változatban.

Harmadszor, az új *iostream* könyvtár sok új adattípust definiál.

Negyedszer, régi könyvtár használatához szükséges a .H-stílusú fejállomány importálása, úgymint IOSTREAM.H. Ezek a fejállományok definiálják a régi stílusú *iostream* osztályokat és teszik őket globálisan láthatóvá. Ezzel szemben, az új standard osztályok használatához elegendő az új stílusú **<iostream>** fej feltüntetése a programunkban.

A két könyvtár közti különbségek miatt célszerűnek találtuk, hogy a kézikönyv külön fejezeteiben tárgyaljuk azokat. Ez a rész a régi stílusú C++ I/O rendszert mutatja be.

■ *Az alapvető csatornaosztályok*

A régi stílusú *iostream* könyvtár szolgáltatásai az IOSTREAM.H fejállomány importálásával válik láthatóvá. Ez az állomány definiálja az osztályhierarchiát, amely lehetővé teszi az alapvető I/O műveletek használatát. Amennyiben állománykezelést is kívánunk végezni a programban, úgy az FSTREAM.H importálása is szükséges. Tömb alapú I/O műveletek a STRSTREA.H fejet igénylik.

A legalacsonyabb szinten lévő osztály a **streambuf**. Ez az alapvető input- és output-műveleteket biztosítja. Általában nem közvetlenül, hanem a leszármazottain keresztül használjuk. Szinte kizárólag akkor van szükségünk a **streambuf**-ra, ha saját I/O osztályt szeretnénk származtatni a **streambuf**-ból.

Az **ios** osztály már a hierarchia olyan szintjén található, amely alkalmassá teszi a közvetlen használatra; tipikus alaposztálya a C++ I/O rendszerének. Biztosít formátumkezelő, hibaellenőrző és állapotjelző funkciókat. Az **ios**-ból számos más fontos osztály származik – néha közbülső osztályokon keresztül. Ezek közül a tipikusabbakat (amelyekkel a legtöbb probléma megoldható) az alábbi táblázat foglalja össze:

Osztály	*Felhasználási kör*
istream	Általános bevitel
ostream	Általános kivitel
iostream	Általános be- és kivitel
ifstream	Fájl bevitel
ofstream	Fájl kivitel
fstream	Fájl be- és kivitel
istrstream	Tömb alapú bevitel
ostrstream	Tömb alapú kivitel
strstream	Tömb alapú be- és kivitel

■ *A C++ predefinit csatornái*

Amikor egy C++ program végrehajtása megkezdődik, négy csatorna automatikusan megnyitásra kerül. Ezek a következők:

Csatorna	*Jelentés*	*Alapértelmezett eszköz*
cin	Szabványos bemenet	Billentyűzet
cout	Szabványos kimenet	Képernyő
cerr	Szabványos hibakimenet	Képernyő
clog	A cerr pufferolt változata	Képernyő

Alapértelmezésben a standard csatornák szolgálják a konzollal való kapcsolattartást. Azokban a környezetekben azonban, amelyek támogatják a be- és kimenet átirányítását (úgymint a DOS, UNIX vagy Windows), a szabványos csatornák átirányíthatók más eszközökbe ill. fájlokba.

■ *A formátumjelző bitek*

A C++ I/O rendszerben minden csatornához tartozik egy formátumjelző bithalmaz, amely a csatornákba bekerülő, illetve a csatornákból szerzett információk formátumáért felelős. Az **ios**-ban az alábbi enumerációs értékek vannak definiálva, amelyekkel, be- illetve kikapcsolhatjuk az egyes formátumjelző biteket:

adjustfield	hex	scientific	stdio
basefield	internal	showbase	unitbuf
dec	left	showpoint	uppercase
fixed	oct	showpos	
floatfield	right	skipws	

Ezeket a jelzőbiteket az **ios** osztály definiálja, ezért használatukhoz szükséges az osztály feltüntetése. Például a **left** flagre így hivatkozhatunk: **ios::left**.

A **skipws** bit bekapcsolásával a vezető üres karaktereket (szóköz, tabulátor, új sor) hagyjuk figyelmen kívül egy csatornából történő beolvasáskor. A **skipws** jelzőbit törlésével az üres karakterek is beolvasásra kerülnek.

A **left** bit bekapcsolt állapotában a kimenet balra igazított formátumban jelenik meg. Hasonlóképpen a **right** jelzőbit bekapcsolása jobbra iga-

zításhoz vezet. Az **internal** bekapcsolásával a numerikus értékek a lehető legjobban széthúzva (szóközök segítségével) jelennek meg a kijelölt mezőben. Ha ezen bitek semelyike sincs bekapcsolva, az alapértelmezett jobbra igazított mód lép életbe.

Alapértelmezésben a numerikus adatok 10-es számrendszerben kerülnek a kimenetre. Természetesen lehetőségünk van ennek megváltoztatására. Az **oct** jelzőbit bekapcsolásával a számok nyolcas számrendszerbeli reprezentációihoz jutunk. Hasonlóképpen a **hex** bekapcsolásával az adatok 16-os számrendszerben jelennek meg a kimeneten. A **dec** flaggel a decimális reprezentációhoz térhetünk vissza.

A **showbase** beállításával elérhető, hogy a numerikus értékek az ábrázoló számrendszer alapjával együtt jelenjenek meg. Bekapcsolt állapotában például az 1F hexadecimális 0x1F-ként kerül kiírásra.

Tudományos formátumban megjelenített számokban alapértelmezésben az „e" kisbetűs. Hasonlóan, amikor hexadecimális értékeket jelenítünk meg, az „x" kisbetűs. Az **uppercase** bekapcsolásával, ezek a karakterek nagybetűkként jelennek meg.

A **showpos** bekapcsolása a pozitív számok elé plusz jelet helyez.

A **showpoint** bekapcsolásának eredményeként minden lebegőpontos szám (akár szükséges, akár nem) tizedesponttal és szükség esetén nullákkal kiegészítve jelenik meg a kimeneten.

A **scientific** jelzőbit bekapcsolásakor a lebegőpontos numerikus értékek tudományos formátumban jelennek meg. A **fixed** bit bekapcsolásával a normál jelölésmód állítható be. Amennyiben semelyik jelzőbit sincsen bekapcsolva, a fordító választja ki a megfelelő formátumot.

Az **unitbuf** bekapcsolása a puffer kiürítését eredményezi minden kimeneti művelet után.

Az **stdio** jelzőbit bekapcsolt állapotban az **stdout** és az **stderr** azonnali kimenetre ürítését teszi lehetővé.

Az **oct**, **dec** és **hex** mezőkre együtt **ios::basefield**-ként hivatkozhatunk. Hasonlóan, a **lef**, **right** és **internal** mezőkre **ios::adjustfield**-ként, a **scientific**, **fixed** párosra **ios:floated**-ként hivatkozhatunk.

A formátumjelző bitek tárolására általában egy **long integer** szolgál, beállításuk az **ios** osztály különböző függvénytagjaival történhet.

I/O manipulátorok

A formátum flagek közvetlen manipulálásán kívül más lehetőségek is rendelkezésünkre állnak a csatornaformátum jellemzőinek beállítására. A második módszert az ún. I/O manipulátorok biztosítják, ezek speciális függvények, amelyeket beépíthetünk I/O kifejezéseinkbe. A régi stílusú *iostream* könyvtár által definiált manipulátorokat az alábbi táblázat közli:

Manipulátor	*Cél*	*Bemenet/Kimenet*
dec	Decimális egész használata	Bemenet/Kimenet
endl	Újsor karakter kimentere küldése és a csatornapuffer ürítése	Kimenet
ends	Nulla küldése a kimenetre	Kimenet
flush	Csatornapuffer ürítése	Kimenet
hex	Hexadecimális egész használata	Bemenet/Kimenet
oct	Oktális szám használata	Bemenet/Kimenet
resetiosflags (long *f*)	Az *f*-beli I/O jelzőbitek törlése	Bemenet/Kimenet
setbase(int *base*)	*Base* alapú számrendszer használata	Kimenet
setfill(int *ch*)	A *ch* kitöltő karakter használata	Kimenet
setiosflags (long *f*)	Az *f*-beli jelzőbitek bekapcsolása	Bemenet/Kimenet
setprecision (int *p*)	A megjelenítendő tizedes jegyek megadása	Kimenet
setw(int *w*)	A mezőszélesség *w*-re állítása	Kimenet
ws	Vezető üres karakterek átugrása	Bemenet

A manipulátorok (pl. **setw( )**) használatához az IOMANIP.H fej importálása szükséges.

Programozási tipp

Látható, hogy kétféle, jól elkülöníthető manipulátortípus van: az egyikbe a paraméter nélküliek, a másikba a paramétereket elfogadók tartoznak. A paraméterezett manipulátorok létrehozása bonyolultsága miatt túlmutat e könyv célján, könnyűszerrel elkészíthetjük viszont saját paraméter nélküli manipulátorainkat.

A paraméter nélküli kimeneti manipulátorok az alábbi vázra épülnek:

```
ostream &manip_név(ostream &csatorna)
{
  //manipulátor gerinc
  return csatorna;
}
```

ahol a manip_név a manipulátor neve. Vegyük észre, hogy a függvény visszatérési értéke egy ***ostream*** *típusú adatfolyam. Ennek akkor van jelentősége, ha a manipulátor egy nagyobb I/O kifejezés részeként használjuk. Fontos azonban megértenünk, hogy habár a manipulátornak van egy formális argumentuma (amely referencia a manipulálandó csatornára), nem szerepeltetünk argumentumot, amikor a manipulátort egy kimenteti műveletbe ágyazzuk.*

A paraméter nélküli bemeneti manipulátorok az alábbi vázra épülnek:

```
istream &manip_név(istream &csatorna)
{
  //manipulátor gerinc
  return csatorna;
}
```

A bemeneti manipulátor paraméterként megkapja a manipulálandó csatornára mutató pointert. A manipulátor ugyanezt a csatornát adja vissza.

A következő példában definiálunk egy ***setup( )*** *nevű kimeneti manipulátort és bemutatjuk használatát. A* ***setup( )*** *bekapcsolja a balra igazított módot, a mezőszélességet 10 karakternyire állítja és a $ jelet adja meg kitöltő karakternek.*

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

ostream &setup(ostream &stream)
{
  stream.setf(ios::left);
  stream << setw(10) << setfill('$');
  return stream;
}
```

```
int main()
{
  cout << 10 << " " << setup << 10;

  return 0;
}
```

Figyelem! Fontos, hogy a manipulátor visszaadja azt az adatfolyamot, amelyre alkalmaztuk. Ha nem így járunk el, akkor a manipulátor nem lesz ágyazható input/output műveletek sorozatába.

■ *A régi stílusú iostream függvények*

Ebben a fejezetrészben leggyakrabban használt *iostream* függvényeket mutatjuk be.

bad

```
#include <iostream.h>
int bad() const;
```

A **bad( )** függvény az **ios** osztály tagja.

A **bad( )** nullától különböző értéket ad vissza, ha súlyos I/O hiba történt az adott csatorna használatakor. Minden más esetben nullát kapunk vissza.

Vonatkozó függvények: **good( )**.

clear

```
#include <iostream.h>
void clear(int flags = 0);
```

A **clear( )** függvény az **ios** osztály tagja.

A **clear( )** az adott adatfolyamhoz rendelt állapotjelző biteket törli. Ha a *flags* értéke 0 (alapértelmezett érték), akkor minden hibajelző bitet 0-ra

állít. Egyébként az állapotjelző bitek a *flags*-ben megadott érték bittérképe szerint lesznek beállítva.

Vonatkozó függvények: **rdstate( )**.

eatwhite

```
#include <iostream.h>
void eatwhite();
```

Az **eatwhite( )** függvény az **ios** osztály tagja.

A függvény az adott csatorna vezető üres karaktereit átolvassa és a get mutatót az első nem üres karakterre helyezi.

Vonatkozó függvények: **ignore( )**.

eof

```
#include <iostream.h>
int eof() const;
```

Az **eof( )** függvény az **ios** osztály tagja.

Az **eof( )** függvény nullától különböző értéket ad vissza, ha az adott adatfolyamhoz tartozó bemeneti fájl végéhez értünk. Egyébként visszatérési értéke 0.

Vonatkozó függvények: **bad( )**, **fail( )**, **good( )**, **rdstate( )**, **clear( )**.

fail

```
#include <iostream.h>
int fail() const;
```

A **fail( )** függvény az **ios** osztály tagja.

A **fail( )** függvény nullától különböző értékkel tér vissza, ha valamilyen I/O hiba történt. Egyébként nullát kapunk vissza.

Vonatkozó függvények: **good( )**, **eof( )**, **bad( )**, **clear( )**, **rdstate( )**.

fill

```
#include <iostream.h>
char fill() const;
char fill(char ch);
```

A **fill( )** függvény az **ios** osztály tagja.

Alapértelmezésben, egy mezőt az értékes adatot képező karakterek mellet szóközök töltik ki. A **fill( )** függvény segítségével a kitöltő karaktert választhatjuk meg. A kívánt karaktert a *ch* paraméterben kell megadnunk. Visszatérési értékként a régi kitöltőkaraktert kapjuk meg.

Ha csak az aktuális kitöltőkaraktert kívánjuk lekérdezni, akkor használjuk a **fill( )** paraméter nélküli változatát.

Vonatkozó függvények: **precision( )**, **width( )**.

flags

```
#include <iostream.h>
long flags() const;
long flags(long f);
```

A **flags( )** függvény az **ios** osztály tagja.

A **flags( )** első (paraméter nélküli) változata egyszerűen visszaadja az adott csatornához tartozó formátumjelző bit beállításokat.

A **flags( )** másik verziója az adott csatornához tartozó formátumjelző biteket állítja be az *f* paraméternek megfelelően. Visszatérési értékként itt az előző beállítást kapjuk meg.

Vonatkozó függvények: **unsetf( )**, **setf( )**.

flush

```
#include <iostream.h>
ostream &flush();
```

A **flush( )** függvény az **ostream** osztály tagja.

A **flush( )** függvény meghívásának eredményeként az adott csatornához tartozó puffer tartama a fizikai eszközre kerül. A függvény az aktuális adatfolyam-referenciát adja vissza.

Vonatkozó függvények: **put( )**, **write( )**.

fstream, ifstream, ofstream

```
#include <fstream.h>
fstream();
fstream(const char *filename, int mode,
        int access=filebuf::openprot);
fstream(int fd);
fstream(int fd, char, *buf, int size);
ifstream();
ifstream(const char *filename, int mode=ios::in,
         int access=filebuf::openprot);
ifstream(int fd);
ifstream(int fd, char *buf, int size);

ofstream();
ofstream(const char *filename, int mode=ios::out,
         int access=filebuf::openprot);
ofstream(int fd);
ofstream(int fd, char *buf, int size);
```

Az **fstream( )**, az **ifstream( )** és az **ofstream( )** rendre az **fstream**, **ifstream** és az **ofstream** osztályok konstruktor függvényei.

Az **fstream( )**, az **ifstream( )** és az **ofstream( )** paraméter nélküli változataival olyan csatornákat hozhatunk létre, amelyekhez nem kapcsolódnak fájlhoz. Ezeket az **open( )** függvénnyel később kapcsolhatjuk fájlokhoz.

Alkalmazásokban a leggyakrabban az **fstream( )**, az **ifstream( )** és az **ofstream( )** azon változatára van szükségünk, amelynek első paramétereként egy fájlnév adható meg. Habár teljesen helyénvaló az **open( )** használata fájlok megnyitására, a legtöbb esetben mégsem ezt a módszert választjuk, hanem a kézenfekvő megoldást, amelyet az **ifstream**, **ofstream** és az **fstream** konstruktorfüggvényei kínálnak, nevezetesen, hogy

automatikusan megnyitják a paraméterükként megadott fájlt a csatorna létrehozásakor. A konstruktorfüggvények paraméterei és azok alapértékei megegyeznek az **open( )**-ével. (Lásd az **open( )**-ről szóló részt.) A programokban leggyakrabban tehát az alábbi fájlmegnyitással találkozhatunk:

```
ifstream mystream("myfile");
```

Ha valamilyen oknál fogva a kívánt fájlt nem lehet megnyitni, akkor hozzárendelt adatfolyam objektumértéke 0 lesz. Függetlenül attól, hogy melyik módszert választjuk: a konstruktoron keresztüli vagy az **open( )**-nel történő megnyitást, célszerű mindig meggyőződnünk a visszakapott csatorna értékének megvizsgálásával, hogy a fájlt sikerült-e ténylegesen megnyitnunk.

Az **fstream( )**, az **ifstream( )** és az **ofstream( )** azon változat, amelyek csak egy paramétert fogadnak el (egy már érvényes állományleíró formájában), ugyancsak létrehoznak egy csatornát és azt a megadott *fd* állományleíróhoz rendelik.

Az **fstream( )**, az **ifstream( )** és az **ofstream( )** azon változatai, amelyek egy állományleíró, egy pufferre mutató pointer és egy méretparamétert fogadnak el, létrehoznak egy csatornát hozzárendelik az *fd*-ben megadott állományleíróhoz. A *buf* az adatfolyam pufferjét jelöli ki, amelynek méretét bájtokban a *size* paraméter adja meg. (Ha a *buf* nullpointer és/vagy *size* értéke 0, akkor nincs pufferolás.)

Vonatkozó függvények: **close( )**, **open( )**.

gcount

```
#include <iostream.h>
int gcount() const;
```

A **gcount( )** függvény az **iostream** osztály tagja.

A **gcount( )** a legutolsó beviteli művelet által olvasott karakterek számát adja vissza.

Vonatkozó függvények: **get( )**, **getline( )**, **read( )**.

get

```
#include <iostream.h>
int get();
istream &get(char &ch);
istream &get(char *buf, int num, char delim = '\n');
istream &get(streambuf &buf, char delim = '\n');
```

A **get( )** függvény minden változata a beviteli adatfolyamból olvas karaktereket. A **get( )** paraméter nélküli változata egy karaktert olvas az adott csatornáról és értékét visszaadja.

A **get( )** függvény azon verziója, amely egy karakter referenciaparaméterrel rendelkezik, az adott bemeneti csatornáról olvas egy karaktert, amelyet eltárol a *ch*-ba, és visszaad egy referenciát az adatfolyamra. (Megjegyzés: a *ch* lehet **unsigned char*** és **signed char*** típusú is.)

A **get( )** függvény harmadik változata, amely három paramétert vár, a következőképpen működik: Az adott csatornából karaktereket olvas a *buf* által mutatott tömbbe, egészen addig, míg vagy a *delim* paraméterben megadott karaktert nem olvasott, vagy a beolvasott karakterek száma el nem érte a *num*-ban meghatározott korlátot. A *buf* által mutatott string nulla végű lesz. Ha a *delim* paramétert nem adjuk meg, akkor az alapértelmezett újsor ('\n') karakter lesz az elválasztó karakter. A függvény, amikor egy elválasztó karakterhez ér (ami a művelet végét is jelenti), azt nem dolgozza fel (nem teszi a tömbbe) és nem is veszi ki a csatornából. Az elválasztó karakter a következő beviteli művelettel kerül ki az adatfolyamból. A függvény az aktuális adatfolyam-referenciát adja vissza. (Megjegyzés: a *buf* lehet **unsigned char*** és **signed char*** típusú is.)

A **get( )** két paraméteres változata a beviteli csatornából beolvasott karaktereket helyezi a megadott **streambuf** objektumba (vagy ennek egy leszármazottjába). A karakterek a megadott elválasztó karakter első előfordulásáig kerülnek beolvasásra. A függvény egy referenciát ad vissza a csatornára.

Vonatkozó függvények: **put( )**, **read( )**, **getline( )**.

getline

```
#include <iostream.h>
istream &getline(char *buf, int num, char delim = '\n');
```

A **getline( )** függvény az **istream** osztály tagja.

A **getline( )** függvény az adott csatornából karaktereket olvas a *buf* által mutatott tömbbe, egészen addig, míg vagy a *delim* paraméterben megadott karaktert nem olvasott, vagy a beolvasott karakterek száma el nem érte a *num*-ban meghatározott korlátot. A *buf* által mutatott string nulla végű lesz. Ha a *delim* paramétert nem adjuk meg, akkor az alapértelmezett újsor ('\n') karakter szolgál az elválasztó karakterként. Ha a függvény egy elválasztó karakterhez ér, akkor azt nem hagyja a csatornában, hanem kiolvassa, de a *buf* tömbbe nem kerül be. A függvény egy referenciát ad vissza a csatornára. (Megjegyzés: a *buf* lehet **unsigned char*** és **signed char*** típusú is.)

Vonatkozó függvények: **get( )**, **read( )**.

good

```
#include <iostream.h>
int good() const;
```

A **good( )** függvény az **ios** osztály tagja.

A **good( )** függvény nullától különböző értékkel tér vissza, ha valamilyen I/O hiba történt az asszociált adatfolyamban, különben visszatérési értéke 0.

Vonatkozó függvények: **bad( )**, **fail( )**, **eof( )**, **clear( )**, **rdstate( )**.

ignore

```
#include <iostream.h>
istream &ignore(int num = 1, int delim = EOF);
```

Az **ignore( )** függvény az **istream** osztály tagja.

Az **ignore( )** függvény segítségével a bemeneti csatornából karaktereket olvashatunk át. A függvény addig olvassa a karaktereket, amíg azok száma el nem éri a *num* paraméterben megadott korlátot, vagy amíg egy a *delim*-ben (alapértelmezésben EOF, azaz a fájlvége jel) megadott elválasztó karakterhez nem érünk. Ha elválasztó karakterhez ér, akkor azt kihagyja a bemeneti csatornából. A függvény egy referenciával tér vissza a csatornára.

Vonatkozó függvények: **get( )**, **getline( )**.

open

```
#include <fstream.h>
void open(const char *filename, int mode, int
          access=filebuf::openprot);
```

Az **open( )** függvény az **fstream**, **ifstream**, **ofstream** osztályok tagja.

Az **open( )** függvénnyel az adott csatornát egy konkrét fájlhoz rendelhetjük. A *filename* paraméterben adjuk meg a fájl nevét, amely tartalmazhatja a teljes elérési utat is. A *mode* értéke határozza meg a fájl megnyitásának módját. A *mode*-nak az alábbi értékek kombinációi közül kell kikerülnie :

ios::app
ios::ate
ios::binary
ios::in
ios::nocreate
ios::noreplace
ios::out
ios::trunc

Ezen értékek összeköthetők a | „bitenkénti vagy" művelettel.

Az **ios::app** szerepeltetésével megnyitott fájlhoz küldött adatok a fájl eredeti tartalmának végétől kerülnek kiírásra. Ez csak kimentként megnyitható fájlokra alkalmazható. Az **ios::ate** érték hatására megnyitáskor a

fájl végére kerül az állománymutató. Annak ellenére, hogy az **ios::ate** megkeresi a fájl végét, I/O műveletek a fájl bármely részén elképzelhetőek.

Az **ios::binary** érték a fájlt bináris I/O műveletekre nyitja meg. Alapértelmezésben a fájlok szöveges módban kerülnek megnyitásra.

Az **ios::in** értékkel a fájlt olvasásra nyitjuk meg. Az **ios::out** fájlt írásra nyitjuk meg. A **ifstream** osztály használata automatikusan input fájl használatát feltételezi, hasonlóképpen ha egy **ofstream** objektumból történik a hívás, a fájl kimenetként kezelődik, így az **ios::in** és **ios::out** értékek megadása fölösleges.

Az **ios::trunc** érték hatására a megadott nevű fájl korábbi tartalma megsemmisül, a rajta végzett első művelet egy üres fájlt talál.

Az **ios::nocreate** hatására az **open( )** hibát jelez, ha a megadott nevű fájl nem létezik. Az **ios::noreplace** hatására az **open( )** hibát jelez, ha már létezik a megadott nevű fájl és az **ios::app** vagy **ios::ate** nem komponense a *mode*-nak.

Az *access* értéke azt mondja meg, hogy a fájlhoz hogyan férhetünk hozzá, alapértelmezésben értéke **filebuf::openprot** (**filebuf** a fájlosztályok egy alaposztálya), amely egy szokásos fájlt jelent. Az *access* lehetséges értékeinek megismeréséhez forduljunk a fordító dokumentációjához.

Az **open( )** mind a *mode*, mind pedig az *access* paraméter számára biztosít alapértelmezett értéket. Input fájl megnyitásakor a *mode* alapértéke **ios::in**, míg output fájl megnyitásakor **ios::out**. Az *access* alapértéke mindkét esetben a szokásos fájloknak megfelelő **filebuf::openprot**. Az alábbi példasor egy TEST nevű állományt nyit meg írásra:

```
out.open("test");// alapértelmezésként írásra nyílik meg
```

Ha egy fájlt egyszerre írásra és olvasásra is meg szeretnénk nyitni, akkor a *mode* értékének képzésében mind az **ios:in**, mind az **ios::out** részt vesz. Ezt illusztrálja az alábbi példa:

```
mystream.open("test", ios::in | ios::out);
```

Vigyázat, a *mode* paraméter megadása nélkül nem lehet egy fájlt írásra és olvasásra megnyitni: nincsen ilyen alapértelmezés.

Ha **open( )** nem tudja megnyitni a kívánt fájlt, a csatorna értéke 0 lesz. Ezért, mielőtt egy fájllal dolgozni kezdenénk, vizsgáljuk meg, hogy az **open( )** sikeres volt-e.

Vonatkozó függvények: **close( )**, **fstream( )**, **ifstream( )**, **ofstream( )**.

Programozási tipp

Szöveges fájlból történő olvasáskor, illetve szöveges fájlba történő íráskor elegendő a << és a >> operátorokat használnunk a megnyitott csatornára. Az alábbi példaprogram a TEST nevű fájlba kiír egy egész számot, egy lebegőpontos értéket és egy karakterláncot, majd ezeket visszaolvassa.

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

int main()
{
  ofstream out("test");
  if(!out) {
    cout << "Az állományt nem tudom megnyitni.
         \n";
    return 1;
  }

// adat kiírás
 out << 10 << " " << 123.23 << "\n";
 out << "Ez egy rövid szöveges fájl.\n";
 out.close();

// most jön a visszaolvasás
  char ch;
  int i;
  float f;
  char str[80];

  ifstream in ("test");
  if(!in) {
```

```
        cout << "Az állományt nem tudom megnyitni.
                \n";
        return 1;
    }
    in >> i;
    in >> f;
    in >> ch;
    in >> str;
    cout << "az adatok: ";
    cout << i << " " << f << " " << ch << "\n";
    cout << str;

    in.close();

    return 0;

}
```

Fontos észben tartanunk, hogy amikor szöveges állományból olvasunk az >> operátorral, előfordulhatnak bizonyos karakterhelyettesítések. Például az üres karakterek átolvasásra kerülnek.
Ha ezt meg kívánjuk akadályozni, akkor nyissuk meg a fájlt binárisan és használjuk a C++ bináris I/O függvényeit.

peek

```
#include <iostream.h>
int peek();
```

A **peek( )** függvény az **iostream** osztály tagja.

A **peek( )** az adatfolyam következő karakterét adja vissza, kivéve ha a fájl végén vagyunk, ekkor az eredmény **EOF** lesz. A függvény semmilyen körülmények között nem veszi ki a beolvasott karaktert a csatornából.

Vonatkozó függvények: **get( )**.

precision

```
#include <iostream.h>
int precision() const;
int precision(int p);
```

A **precision( )** függvény az **ios** osztály tagja.

Alapértelmezésben a kimenetre hat tizedesjegy pontossággal kerülnek ki a lebegőpontos számok. A **precision( )** második változatát használva ezen változtathatunk. A *p* paraméter felelős azért, hogy hány tizedesjegy pontossággal jelennek meg adataink a kimeneten. Visszatérési értékként az előző beállítást kapjuk.

Vonatkozó függvények: **width( )**, **fill( )**.

put

```
#include <iostream.h>
ostream &put(char ch);
```

A **put( )** függvény az **ostream** osztály tagja.

A **put( )** a *ch* paraméterben megadott karaktert írja az adott kimeneti csatornába. Visszatérési értékként egy referenciát kapunk a csatornára.

Vonatkozó függvények: **write( )**, **get( )**.

putback

```
#include <iostream.h>
istream &putback(char ch);
```

A **putback( )** függvény az **istream** osztály tagja.

A **putback( )** függvény visszaadja a *ch* karaktert az asszociált bemeneti csatornának.

Megjegyzés
ch értéke kizárólag a csatornából a legutoljára beolvasott karakter lehet.

Vonatkozó függvények: **peek( )**.

rdstate

```
#include <iostream.h>
int rdstate() const;
```

Az **rdstate( )** függvény az **ios** osztály tagja.

Az **rdstate( )** függvény az adott csatorna állapotát adja vissza. A C++ I/O rendszere minden aktív csatornához állapotinformációkat rendel, amelyek az egyes I/O műveletek eredményét jellemzik. Az I/O rendszer aktuális állapotát egy integer tárolja, amely az alábbi jelzőbiteket tartalmazza:

Név	*Jelentés*
ios::goodbit	Nem történt hiba
ios::eofbit	Fájl végéhez értünk
ios::failbit	Nem súlyos I/O hiba történt
ios::badbit	Súlyos I/O hiba történt

Ezek a jelzőbitek az **ios**-ban vannak felsorolva.

rdstate( ) 0-t (**ios::goodbit**) ad vissza, amennyiben nem történt hiba. Minden más esetben valamilyen hibabit 1-esre vált.

Vonatkozó függvények: **eof( )**, **good( )**, **bad( )**, **clear( )**, **fail( )**.

read

```
#include <iostream.h>
istream &read(char *buf, int num);
```

A **read( )** függvény az **istream** osztály tagja.

A **read( )** függvény *num* számú bájtot olvas be az adott bemeneti csatornából és azokat a *buf* által mutatott pufferbe helyezi (a *buf*-nek egyaránt megadható **unsigned char*** és **signed char*** típusú érték is). Ha a **read( )** a fájl végét még a *num* számú karakter beolvasása előtt eléri, akkor a beolvasás abbamarad és a pufferbe az addig beolvasott karakterek lesznek találhatók (l. „**gcount**"). A **read( )** egy referenciát ad vissza a csatornára.

Vonatkozó függvények: **gcount( )**, **get( )**, **getline( )**, **write( )**.

seekg, seekp

```
#include <iostream.h>
istream &seekg(streamoff offset, ios::seek_dir origin)
istream &seekg(streampos position);

ostream &seekp(streamoff offset, ios::seek_dir origin);
ostream &seekp(streampos position);
```

A **seekg( )** függvény az **istream** a **seekp( )** az **ostream** osztály tagja.

A C++ I/O rendszerében a közvetlen fájlelérést a **seekg( )** és a **seekp( )** függvényekkel valósíthatjuk meg. A közvetlen hozzáférést a C++ úgy biztosítja, hogy minden fájlhoz két mutatót tart fenn, az egyik az ún. *get mutató*, amelyik a következő inputművelet helyét jelöli meg. A másik a *put mutató*, amely a következő outputművelet pozíciójára mutat. Minden alkalommal, amikor valamilyen be- vagy kiviteli művelet történik, a megfelelő pointer automatikusan előrébb állítódik. A **seekg( )** és a **seekp( )** függvényekkel lehetőségünk nyílik a fájlok nem szekvenciális elérésére.

A **seekg( )** kétparaméteres változata az *origin* által meghatározott helytől *offset* bájtnyira állítja a get mutatót. Hasonlóképpen a **seekp( )** kétparaméteres változata a az *origin* által meghatározott helytől *offset* bájtnyira állítja a put mutatót. Az *offset* paraméter **streamoff** típusú, amelyet az IOSTREAM.H definiál. A **streamoff** egy olyan objektum, amelynek maximális értékét az szabja meg, hogy mi a lehető legnagyobb megengedett értéke egy ilyen eltolásnak.

Az *origin* paraméter **ios::seek_dir** típusú, amely egy enumeráció. Az alábbi értékek valamelyikét kell, hogy felvegye:

Név	*Jelentés*
ios::beg	Fájl eleje
ios::cur	Aktuális pozíció
ios::end	Fájl vége

A **seekg( )** és a **seekp( )** egyparaméteres változatai az állománymutatókat a *position* által meghatározott helyre állítják. Ez az érték egy korábbi **tellg( )**, ill. **tellp( )** hívásból kell hogy származzon. A **streampos** típust az IOSTREAM.H fej definiálja. Maximális értékét az szabja meg, hogy mi a lehető legnagyobb megengedett értéke a *position*-nak. Ezek a függvények egy referenciát adnak vissza a csatornára.

Vonatkozó függvények: **tellg( )**, **tellp( )**.

setf

```
#include <iostream.h>
long setf(long flags);
long setf(long flags1, long flags2);
```

A **setf( )** függvény az **ios** osztály tagja.

A **setf( )** első változata a *flag*-ban megadottak szerint állítja be a formátumjelző biteket. (Más jelzőbitre nincs hatással.) Például a **showpos** flaget az alábbi utasítással kapcsolhatjuk be:

```
stream.setf(ios::showpos);
```

ahol a *stream* az az adatfolyam, amelyhez tartozó formátumjelző biteket meg akarjuk változtatni.

Fontos, hogy az **fstream( )**, csak egy adott csatornára vonatkozólag fejti ki hatását. Így nem lenne értelme a **setf( )** önmagában történő hívásának. Minden csatorna rendelkezik saját formátumállapottal.

Több flaget egyszerre a | („bitenkénti vagy") operátor segítségével állíthatunk.

A **setf( )** második verziójában a *flags2* paraméter az átállítható jelzőbiteket határozza meg. A függvény először törli ezeket a biteket, majd a *flag1*-nek megfelelően állítja be őket.

A **setf( )** mindkét változata a csatornához rendelt formátumjelző bitek előző állapotát adja vissza.

Vonatkozó függvények: **unset( )**, **flags( )**.

setmode

```
#include <fstream.h>
int setmode(int mode = filebuf::text);
```

A **setmode( )** függvény az **ofstream** és az **ifstream** osztályok tagja.

A **setmode( )** függvénnyel az adott csatorna manipulálásának módját állíthatjuk be binárisra vagy szövegesre. (A szöveges mód az alapértelmezés.) A *mode* lehetséges értékei: **filebuf::text** és **filebuf::binary**.

A függvény az előző beállítást adja vissza, ha nem történt hiba, különben –1-et.

Vonatkozó függvények: **open( )**.

str

```
#include <strstrea.h>
char *str();
```

A **str( )** függvény a **strstream** osztály tagja.

A **str( )** függvény egy dinamikusan allokált bemeneti tömböt „fagyaszt be" és visszaad egy, a tömbre mutató pointert. Befagyasztása után egy dinamikus tömb nem használható többet kimenetként. Ezért ne fagyasszunk be olyan tömböt, amelybe még további kimenő karaktereket várunk.

Megjegyzés
Ez a függvény a tömbalapú I/O műveleteknek csoportjába sorolandó.

Vonatkozó függvények **strstream( )**, **istrstream( )**, **ostrstream( )**.

strstream, istrstream, ostrstream

```
#include <strstrea.h>
strstream();
strstream(char *buf, int size, int mode);

istrstream (const char *buf);
istrstream (const char *buf, int size);

ostrstream ();
ostrstream (char *buf, int size, int mode=ios::out);
```

A **strstream** konstruktor a **strstream** osztály tagja, az **istrstream** konstruktor az **istrstream** osztály tagja, és az **ostrstream** konstruktor az **ostrstream** osztály tagja.

Ezek a konstruktorok tömbalapú csatornák létrehozását végzik, amelyek a C++ tömbalapú I/O függvényeit támogatják.

Az **ostrstream( )** függvény *buf* paramétere mutató egy olyan tömbre, amelybe a csatornába küldött karakterek kerülnek. A tömb méretét a *size* paraméteren keresztül adjuk meg. Alapértelmezésben a csatornát írásra nyitjuk meg, de ezen változtathatunk a *mode* paramétert használva. A *mode* érvényes értékei megegyeznek az **open( )** függvénynél látottakkal. A legtöbb esetben elegendő a *mode* alapértelmezésére hagyatkozni. Amennyiben az **ostrstream( )** paraméter nélküli változata automatikusan allokál számunkra egy dinamikus tömböt.

Az **istrstream( )** egy paraméteres változatában, a *buf* paraméter egy olyan tömbre mutató pointer, amely a csatornából történő olvasáskor a karakterek forrásaként szolgál. A *buf* által mutatott tömb tartalma nulla végű kell hogy legyen. Fontos azonban, hogy nullkarakter soha nem kerül beolvasásra.

Ha a string egy részét kívánjuk csak inputként használni, akkor az **istrstream** konstruktor két paraméteres változatát használjuk. Ekkor csak a *buf* által mutatott tömb az első *size* elemét használjuk. Ebben az esetben nem szükséges, hogy a string nullavégű legyen, hiszen a *size* megadja a használható tömb végét.

Egy ki- és bemenetként egyaránt használható tömbalapú csatorna létrehozását a **strstream( )** konstruktorral végezzük. Paraméteres változatában a *buf* a I/O műveletekre használt célstringre mutat. A *size*-zal a tömb méretét adhatjuk meg. A *mode* paraméter értéke a csatorna használatának

módját rögzíti. Normál I/O műveletekhez az **ios::in | ios::out**. Ha inputként használjuk, a tömb tartalma nullavégű string kell hogy legyen.

Ha a **strstream( )** paraméter nélküli változatát használjuk, be- és kivitelre használt puffer dinamikusan allokálódik, amelyen író és olvasó műveletek egyaránt használhatóak.

Vonatkozó függvények: **str( )** és **open( )**.

sync_with_stdio

```
#include <iostream.h>
static void sync_with_stdio();
```

A **sync_with_stdio( )** függvény az **ios** osztály tagja.

A **sync_with_stdio( )** hívásával lehetővé válik a C típusú I/O rendszer és a C++ osztályalapú I/O rendszerének egyidejű használata.

tellg, tellp

```
#include <iostream.h>
streampos tellg();
streampos tellp();
```

A **tellg( )** függvény az **istream**, míg a **tellp( )** az **ostream** osztály tagja.

A C++ I/O rendszere minden megnyitott fájlhoz két mutatót tart fenn. Az egyik az ún. *get mutató,* amelyik a következő inputművelet helyét jelöli meg a másik a *put mutató,* amely a következő outputművelet pozíciójára mutat. Minden alkalommal, amikor valamilyen be- vagy kiviteli művelet történik, a megfelelő pointer automatikusan előrébb állítódik. A get mutató, illetve a put mutató aktuális pozícióját rendre a **tellg( )** és a **tellp( )** függvényekkel kérdezhetjük le.

A **streampos** típust az IOSTREAM.H fej definiálja, úgy hogy képes legyen a lehető legnagyobb megengedett pozíciót is tárolni.

A **tellg( )** és a **tellp( )** függvényekkel nyerjük rendre a későbbi **seekg( )** és **seekp( )** hívások paramétereit is.

Vonatkozó függvények: **seekg( )**, **seekp( )**.

unsetf

```
#include <iostream.h>
long unsetf(long flags);
```

Az **unsetf( )** függvény az **ios** osztály tagja.

Az **unsetf( )** függvény a kívánt formátumjelző biteket törli, a többi jelzőbit nem változik.

A törlendő jelzőbiteket a *flag* paraméterben adhatjuk meg. A függvény az előzőleg használt jelzőbit-kombinációt adja vissza.

Vonatkozó függvények: **setf( )**, **flags( )**.

width

```
#include <iostream.h>
int width() const;
int width(int w);
```

A **width( )** függvény az **ios** osztály tagja.

Az aktuális mezőszélességet úgy kaphatjuk meg, ha **width( )** első változatát hívjuk meg, ekkor a visszatérési érték tartalmazza a mezőszélességet.

A mezőszélesség beállítása a második változattal történhet, ahol a *w* paraméter közvetítésével adjuk meg a kívánt szélességet. A függvény az előzőleg használt mezőszélességet adja vissza.

Vonatkozó függvények: **precision( )**, **fill ( )**.

write

```
#include <iostream.h>
ostream &write(const char *buf, int num);
```

A **write( )** függvény az **ostream** osztály tagja.

A **write( )** függvény *num* bájtnyit ír az adott kimeni csatornába a *buf* pointer által mutatott pufferből. (A *buf* értéke lehet **unsigned char*** vagy **signed char** típusú is.) A függvény egy referenciát ad vissza a csatornára.

Vonatkozó függvények: **read( )**, **put( )**.

13. FEJEZET

A szabványos C++ I/O rendszer

Amint azt az előző fejezet bevezető részében említettük, a C++ *iostream* könyvtárának két változata létezik. A régi típusú könyvtárral az előző fejezet foglalkozik. Az ANSI/ISO szabványnak megfelelő új változatot ebben a fejezetben mutatjuk be.

■ *Az új iostream könyvtár használata*

Az új és régi iostream könyvtár között két alapvető formai különbség van. Először a régi változat függvényei és típusai a globális névtér alatt voltak definiálva. Az új változat szolgáltatásaihoz a **std** névtér alatt férhetünk hozzá. Másodszor a régi típusú könyvtár a .H stílusú fejállományok által érhető el, míg az új verzió új stílusú fejeket használ.

Az iostream szabványos változatának használatához forrásainkban a **<iostream>** fej feltüntetése szükséges. Ezután kényelmességi okok miatt célszerű az aktuális névtérben is láthatóvá tenni a könyvtárat az alábbi utasítás segítségével:

```
using namespace std;
```

A **using** utasítás megadása után, az új és a régi stílusú könyvtár majdnem ugyanúgy működik. Ezért ha egy régi könyvtárban megírt kódot szeretnénk újra fordítani az új változat használatával, nem kell sokat (vagy egyáltalán nem) változtatnunk rajta.

A fenti **using** utasítás használata természetesen nem kötelező. Ehelyett használhatunk minden alkalommal névtér kvalifikátort is, amikor I/O osztályok valamelyikére hivatkozunk. Az alábbi kód explicit módon hivatkozik a **cout**-ra:

```
std::cout << "Ez egy próba";
```

Természetesen, az iostream gyakori használatakor a **using** használata sok gépeléstől kímélheti meg a programozót.

■ *Az I/O rendszer alapvető osztályai*

A standard iostream könyvtár egy meglehetősen komplex generic osztályhierarchiára épül. Az iostream standard változatában használatos könyvtárnevek nem egyeznek meg a régiekkel. Például a legalacsonyabb szintű osztályok a régi könyvtárban az **ios** és a **streambuf** nevet viselik. Az új könyvtárban ezeket **basic_streambuf**-nak és a **basic_ios**-nak nevezték. Szerencsére a legtöbb esetben a régi stílusú iostream könyvtárban definiált osztályneveket átmentették **typedef** utasítások segítségével, amelyekkel a I/O osztályok karakteralapú változatait hozták létre.

Genericosztály	*A megfelelő karakteralapú osztály*
basic_streambuf	streambuf
basic_ios	ios
basic_istream	istream
basic_ostream	ostream
basic_iostream	iostream
basic_fstream	fstream
basic_ifstream	ifstream
basic_ofstream	ofstream

A tömbalapú I/O osztályokat is támogatja a standard iostream, azonban ezek használatát nem ajánljuk, ehelyett inkább a következő fejezetben bemutatásra kerülő konténerek használata javasolt.

A karakteralapú osztályok mellett, az új könyvtár széleskarakter-alapú I/O osztályokat is támogat. Neveik megegyeznek a megfelelő karakteralapú osztályok neveivel, azzal a különbséggel, hogy „w"-vel kezdődnek. Például a **wios** az **ios** széleskarakter-változata.

Mivel a programozók többsége a karakteralapú I/O rendszert használja, ezért a könyv ezek ismertetésére szorítkozik. Ezért, amikor I/O osztá-

lyokra hivatkozunk, akkor egyszerűen azok **typedef**-fel definiált, karakter-alapú osztályokra utaló nevét használjuk inkább, mint a genericnevüket. Így a könyvben az **ios** név fog szerepelni és nem a **basic_ios**.

■ *A C++ predefinit csatornái*

Az új iostream könyvtár használatakor az alábbi csatornák automatikusan megnyitásra kerülnek.

Csatorna	*Jelentés*
cin	szabványos bemenet
cout	szabványos kimenet
cerr	szabványos hibakimenet
clog	a cerr pufferelt változata
wcin	a cin széles-karakter változata
wcout	a cout széles-karakter változata
wcerr	a cerr széles-karakter változata
wclog	a clog széles-karakter változata

Alapértelmezésben ezek a szabványos csatornák a konzollal történő kapcsolattartást szolgálják. Azokban a környezetekben azonban, amelyek támogatják a be- és kimenet átirányítását (úgymint a DOS, UNIX vagy Windows) a standard csatornák átirányíthatók más eszközökbe, illetve fájlokba.

■ *A streamsize típus*

Az új iostream könyvtár számos új adattípust definiál. Ezek közül a leggyakrabban használtak egyike a **streamsize**. Ez egy olyan egész típus, amelyben a legnagyobb tárolható érték megfelel az I/O műveletek során a legnagyobb egyszerre átvihető bájtsorozat hosszának.

■ *Az iostate típus*

Az I/O csatornák állapotát **iostate** típusú objektumok írják le. Az **iostate** az **ios** által definiált enumeráció, amely az alábbi tagokat tartalmazza:

Név	*Jelentés*
goodbit	Nem történt hiba
eofbit	Fájl végéhez értünk
failbit	Kisebb I/O hibára utal
badbit	Súlyos I/O hiba történt.

■ *A formátumjelző bitek és az fmtflags típus*

A C++ I/O rendszerben minden csatornához tartozik egy formátumjelző bithalmaz, amely a csatornákba bekerülő, illetve a csatornákból szerzett információk formátumáért felelős. Az új iostream könyvtár deklarálja az **fmtflags** nevű felsorolási típust, amelyben az alábbi értékek definiáltak:

adjustfield	floatfield	right	skipws
basefield	hex	scientific	unitbuf
boolalpha	internal	showbase	uppercase
dec	left	showpoint	
fixed	oct	showpos	

Ezekkel az értékekkel állíthatjuk be (kapcsolhatjuk be vagy ki) a formátum jelzőbiteket. A fenti konstansok az **ios** osztályból érhetők el. (Valójában az **fmtflags** az **ios_base** osztályban van definiálva, amely a **basic_ios** alaposztálya, de ez a ténynek nincs különösebb jelentősége a legtöbb programozási szituációban.)

A **skipws** bit bekapcsolásával a vezető üres karaktereket (szóköz, tabulátor, újsor) hagyjuk figyelmen kívül egy csatornából történő beolvasáskor. A **skipws** jelzőbit törlésével az üres karakterek is beolvasásra kerülnek.

A **left** jelzőbit bekapcsolt állapotában a kimenet balra igazított formátumban jelenik meg. Hasonlóképpen a **right** jelzőbit bekapcsolása jobbra igazításhoz vezet. Az **internal** bekapcsolásával a numerikus értékek a lehető

legjobban széthúzva jelennek meg a kijelölt mezőben. Ha ezen bitek semelyike sincs bekapcsolva, az alapértelmezett jobbra igazított mód lép életbe.

Alapértelmezésben a numerikus adatok 10-es számrendszerben kerülnek a kimenetre. Természetesen lehetőségünk van ennek megváltoztatására. Az **oct** jelzőbit bekapcsolásával a számok nyolcas számrendszerbeli reprezentációihoz jutunk. Hasonlóképpen a **hex** bekapcsolásával az adatok 16-os számrendszerben jelennek meg a kimeneten. A **dec** bit 1-re állításával a decimális reprezentációhoz térhetünk vissza.

A **showbase** beállításával elérhető, hogy a numerikus értékek az ábrázoló számrendszer alapjával együtt jelenjenek meg. Bekapcsolt állapotában, például az 1F hexadecimális 0x1F-ként kerül kiírásra.

Tudományos formátumban megjelenített számokban alapértelmezésben az „e" kisbetűs. Hasonlóan, amikor hexadecimális értékeket jelenítünk meg az „x" kisbetűs. Az **uppercase** bekapcsolásával, ezek a karakterek nagybetűkként jelennek meg.

A **showpos** bekapcsolása a pozitív számok elé pluszjelet helyez.

A **showpoint** bekapcsolásának eredményeként minden lebegőpontos szám (akár szükséges, akár nem) tizedesponttal és szükség esetén nullákkal kiegészítve jelenik meg a kimeneten.

A **scientific** jelzőbit bekapcsolásakor a lebegőpontos numerikus értékek tudományos formátumban jelennek meg. A **fixed** jelzőbit bekapcsolásával a normál jelölésmód állítható be. Amennyiben semelyik jelzőbit sincsen bekapcsolva, a fordító választja ki a megfelelő formátumot.

Az **unitbuf** bekapcsolása a puffer kiürítését eredményezi minden kimeneti művelet után.

A **boolalpha** bekapcsolásával, logikai értékeket írhatunk ki, ill. olvashatunk be a **true** és **false** kulcsszavak használatával.

Az **oct**, **dec** és **hex** mezőkre együtt **ios::basefield**-ként hivatkozhatunk. Hasonlóan, a **lef**, **right** és **internal** mezőkre **ios::adjustfield**-ként, a **scientific**, **fixed** párosra **ios:floated**-ként hivatkozhatunk.

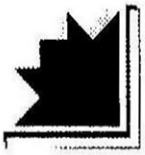
I/O manipulátorok

A formátumjelző bitek közvetlen manipulálásán kívül más lehetőségek is rendelkezésünkre állnak a csatornaformátum jellemzőinek beállítására. A második módszert az ún. I/O manipulátorok biztosítják, ezek speciális függvények, amelyeket beépíthetünk I/O kifejezéseinkbe. Az új *iostream*

könyvtár által definiált manipulátorokat az alábbi táblázat közli (a régebbi könyvtár nem támogatja mindegyiket).

Manipulátor	*Funkció*	*Input/Output*
boolalpha	Bekapcsolja a **boolalpha** bitet.	Bemenet/Kimenet
dec	Bekapcsolja a **dec** bitet.	Bemenet/Kimenet
endl	Újsor karakter kimenetre küldése és a csatornapuffer ürítése.	Kimenet
ends	Nulla küldése a kimenetre.	Kimenet
fixed	Bekapcsolja a **fixed** bitet.	Kimenet
flush	Csatornapuffer ürítése.	Kimenet
hex	Hexadecimális egész használat.	Bemenet/Kimenet
internal	Bekapcsolja az **internal** bitet.	Kimenet
left	Bekapcsolja a **left** bitet.	Kimenet
noboolalpha	Kikapcsolja a **boolalpha** bitet.	Bemenet/Kimenet
noshowbase	Kikapcsolja a **showbase** bitet.	Kimenet
noshowpoint	Kikapcsolja a **showpoint** bitet.	Kimenet
noshowpos	Kikapcsolja a **showpos** bitet.	Kimenet
noskipws	Kikapcsolja a **skipws** bitet.	Bemenet
nounitbuf	Kikapcsolja a **unitbuf** bitet.	Kimenet
nouppercase	Kikapcsolja a **uppercase** bitet.	Kimenet
oct	Bekapcsolja az **oct** bitet.	Bemenet/Kimenet
resetioflags (long *f*)	Az *f* által megjelölt jelzőbiteket kikapcsolja.	Bemenet/Kimenet
right	Bekapcsolja a **right** bitet.	Kimenet
scientific	Bekapcsolja a **scientific** bitet.	Kimenet
setbase (int *base*)	Base alapú számrendszer használat.	Kimenet
setfill (int *ch*)	A *ch* kitöltő karakter használata.	Kimenet
setiosflags (long *f*)	Az *f-beli* jelzőbitek bekapcsolása.	Bemenet/Kimenet
setprecision (int *p*)	A megjelenítendő tizedes jegyek megadása.	Kimenet
setw(int *w*)	A mezőszélesség *w*-re állítása.	Kimenet
showbase	Bekapcsolja a **showbase** bitet.	Kimenet
showpoint	Bekapcsolja a **showpoint** bitet.	Kimenet
showpos	Bekapcsolja a **showpos** bitet.	Kimenet
skipws	Bekapcsolja a **skipws** bitet.	Bemenet
unitbuf	Bekapcsolja az **unitbuf** bitet.	Kimenet
uppercase	Bekapcsolja az **uppercase** bitet.	Kimenet
ws	Vezető üres karakterek átugrása.	Bemenet

A paraméterrel rendelkező manipulátorok (pl. **setw( )**) eléréséhez, szükséges az #include **<iomanip>** programsor.



Programozási tipp

A standard iostream könyvtár új formátum jelzőbitjei közül az egyik legérdekesebb a ***boolalpha****. Ezt a jelzőbitet állíthatjuk közvetlenül, vagy a* ***boolalpha( )*** *és a* ***noboolapha( )*** *manipulátorokon keresztül.*
A ***boolalpha****-t az a tény teszi érdekessé, hogy segítségével lehetővé válik logikai értékek olvasása, illetve írása a* ***true*** *és* ***false*** *kulcsszavak használatával. Normális esetben a 0-t és 1-t használnánk, amelyeknek rendre a logikai igaz és a logikai hamis értékek felelnek meg. Tekintsük az alábbi példaprogramot.*

```
// A boolalpha formátumjelzőbit használata
#include <iostream>
using namespace std;

int main()
{
  bool b;

  cout << "A boolalpha jelzőbit bekapcsolása
          előtt: ";
  b = true;
  cout << b <<" ";
  b = false;
  cout << b << endl;

  cout << "A boolalpha jelzőbit bekapcsolása
          után: ";
  b = true;
  cout << b <<" ";
  b = false;
  cout << b << endl;

  cout <<"Írjon be egy logikai értéket
         (true/false): ";
  cin >> boolalpha >> b;
  cout << "Az előbb bevitt érték: " <<b;

  return 0;
}
```

Egy lefutáskor lehetséges képernyő:

```
A boolalpha jelzőbit bekapcsolása előtt: 1 0
A boolalpha jelzőbit bekapcsolása után: true false
Írjon be egy logikai értéket (true/false): true
Az előbb bevitt érték: true
```

■ *A standard iostream függvények*

Ebben a fejezetrészben a leggyakrabban használt új stílusú iostream függvények kerülnek bemutatásra.

bad

```
#include <iostream>
bool bad() const;
```

A **bad( )** függvény az **ios** osztály tagja.

A **bad( )** nullától különböző értéket ad vissza, ha súlyos I/O hiba történt az adott csatorna használatakor. Minden más esetben nullát kapunk vissza.

Vonatkozó függvények: **good( )**.

clear

```
#include <iostream>
void clear(iostate flags = goodbit);
```

A **clear( )** függvény az **ios** osztály tagja.

A **clear( )** az adott csatornához rendelt állapotjelző biteket törli. Ha a *flags* értéke 0 (alapértelmezett érték), akkor minden hibaflaget 0-ra állít. Egyébként az állapotjelző bitek a *flags*-ben megadott érték bittérképe szerint lesznek beállítva.

Vonatkozó függvények: **rdstate( )**.

eof

```
#include <iostream>
bool eof() const;
```

Az **eof( )** függvény az **ios** osztály tagja.

Az **eof( )** függvény nullától különböző értéket ad vissza, ha az adott csatornához tartozó input fájl végéhez értünk. Egyébként visszatérési értéke 0.

Vonatkozó függvények: **bad( )**, **fail( )**, **good( )**, **rdstate( )**, **clear( )**.

fail

```
#include <iostream>
bool fail() const;
```

A **fail( )** függvény az **ios** osztály tagja.

A **fail( )** függvény nullától különböző értékkel tér vissza, ha valamilyen I/O hiba történt. Egyébként nullát kapunk vissza.

Vonatkozó függvények: **good( )**, **eof( )**, **bad( )**, **clear( )**, **rdstate( )**.

fill

```
#include <iostream>
char fill() const;
char fill(char ch);
```

A **fill( )** függvény az **ios** osztály tagja.

Alapértelmezésben, egy mezőt az értékes adatot képező karakterek mellett szóközök töltik ki. A **fill( )** függvény segítségével a kitöltő karaktert választhatjuk meg. A kívánt karaktert a *ch* paraméterben kell megadnunk. Visszatérési értékként a régi kitöltőkaraktert kapjuk meg.

Ha csak az aktuális kitöltőkaraktert kívánjuk lekérdezni, akkor használjuk a **fill** paraméter nélküli változatát.

Vonatkozó függvények: **precision( )**, **width( )**.

flags

```
#include <iostream>
fmtflags flags() const;
fmtflags flags(fmtflags f);
```

A **flags( )** függvény az **ios** osztály tagja.

A **flags( )** első (paraméter nélküli) változata egyszerűen visszaadja az adott csatornához tartozó formátumjelzőbit-beállításokat.

A **flags( )** másik verziója az adott csatornához tartozó formátumjelző biteket állítja be az *f* paraméternek megfelelően. Visszatérési értékként itt az előző beállítást kapjuk meg.

Vonatkozó függvények: **unsetf( )**, **setf( )**.

flush

```
#include <iostream>
ostream &flush();
```

A **flush( )** függvény az **ostream** osztály tagja.

A **flush( )** függvény meghívásának eredményeként az adott csatornához tartozó puffer tartalma a fizikai eszközre kerül. A függvény az aktuális adatfolyamra ad vissza referenciát.

Vonatkozó függvények: **put( )**, **write( )**.

fstream, ifstream, ofstream

```
#include <fstream>
fstream();
fstream(const char *filename,
        openmode mode = ios::in | ios::out);

ifstream();
ifstream(const char *filename,
         openmode mode=ios::in);
```

```
ofstream();
ofstream(const char *filename,
         openmode mode=ios::out | ios::trunc);
```

Az **fstream( )**, **ifstream( )** és az **ofstream( )** függvények rendre az **fstream**, **ifstream** és az **ofstream** osztályok konstruktorai.

Az **fstream( )**, az **ifstream( )** és az **ofstream( )** paraméter nélküli változataival olyan csatornákat hozhatunk létre, amelyekhez nem kapcsolódnak fájlok. Ezeket az **open( )** függvénnyel később kapcsolhatjuk fájlokhoz.

Alkalmazásokban a leggyakrabban az **fstream( )**, az **ifstream( )** és az **ofstream( )** azon változatára van szükségünk, amelynek első paramétereként egy fájlnév adható meg. Habár teljesen helyénvaló az **open( )** használata fájlok megnyitására, mégis a legtöbb esetben nem ezt a módszert választjuk, hanem a kézenfekvő megoldást, amelyet az **ifstream**, **ofstream** és az **fstream** konstruktorfüggvényei kínálnak, nevezetesen, hogy automatikusan megnyitják a paraméterükként megadott fájlt az adatfolyam létrehozásakor. A konstruktorfüggvények paraméterei és azok alapértékei megegyeznek az **open( )**-ével. (Lásd az **open( )**-ról szóló részt.) A programokban leggyakrabban tehát az alábbi fájlmegnyitással találkozhatunk:

```
ifstream mystream("myfile");
```

Ha valamilyen oknál fogva a kívánt fájlt nem lehet megnyitni, akkor az asszociált stream változó értéke 0 lesz. Függetlenül attól, hogy melyik módszert választjuk: az a konstruktoron keresztüli vagy az **open( )**-nel történő megnyitást, célszerű mindig meggyőződnünk a visszakapott csatorna értékének megvizsgálásával, hogy a fájlt sikerült-e ténylegesen megnyitnunk.

Az **openmode** típust az **ios_base** osztály definiálja. További részletekért lásd még az **open( )** függvény leírását.

Vonatkozó függvények: **close( )**, **open( )**.

gcount

```
#include <iostream>
streamsize gcount() const;
```

A **gcount( )** függvény az **iostream** osztály tagja.

A **gcount( )** a legutolsó beviteli művelet által olvasott karakterek számát adja vissza.

Vonatkozó függvények: **get( )**, **getline( )**, **read( )**.

get

```
#include <iostream>
int get();
istream &get(char &ch);
istream &get(char *buf, streamsize num);
istream &get(char *buf, streamsize num, char delim);
istream &get(streambuf &buf);
istream &get(streambuf &buf, char delim);
```

A **get( )** függvény az **istream** osztály tagja.

A **get( )** függvény minden változata a beviteli csatornából olvas karaktereket. A **get( )** paraméter nélküli változata egy karaktert olvas az adott csatornáról és értékét visszaadja.

A **get(char &*ch*)** az adott csatornából olvas egy karaktert és értékül adja a *ch* paraméternek. A csatornára mutató pointerrel tér vissza.

A **get (char **buf*, streamsize *num*)** karaktereket olvas a *buf* által mutatott pufferbe addig, amíg a beolvasott karakterek száma el nem éri a *num-1*-et , vagy sorvégéhez nem ér. A pufferben megjelenő tömb „nullavégűségéről" a **get( )** gondoskodik. A beolvasás végét okozó újsor karakter nem kerül ki a csatornából a következő beolvasási műveletig. A függvény egy referenciát ad vissza a csatornára.

A **get(char **buf*, streamsize *num*, char *delim*)** függvény karaktereket olvas az adott csatornából a *buf* által mutatott tömbbe addig, amíg a beolvasott karakterek száma el nem éri a *num-1*-et, vagy a *delim* paraméterben megadott karakter nem kerül olvasásra vagy a fájl végéhez nem ér. A függvény biztosítja a pufferben megjelenő tömb nullavégűségét. A függvény, amikor egy elválasztó karakterhez ér (ami a művelet végét is jelenti), azt nem dolgozza fel (nem teszi a tömbbe) és nem is veszi ki a csatornából. Az elválasztó karakter a következő beviteli művelettel kerül ki az adatfolyamból. A függvény egy referenciát ad vissza a csatornára.

A **get(streambuf *_buf_)** függvény karaktereket olvas a bemeneti csatornáról a megadott **streambuf** objektumba. A beolvasás addig történik, amíg újsor vagy fájlvége jel nem kerül olvasásra. A függvény egy referenciát ad vissza a csatornára. A beolvasás végét okozó újsor karakter nem kerül ki a csatornából a következő beolvasási műveletig.

A **get(streambuf &_buf_, char _delim_)** függvény a bemeneti csatornából olvas karaktereket a megadott **streambuf** objektumba. A beolvasás addig történik, amíg a *delim* paraméterben megadott elválasztó jel beolvasásra nem kerül vagy el nem éri a fájl végét. A függvény egy referenciát ad vissza a csatornára. A beolvasás végét okozó elválasztó karakter nem kerül ki a csatornából a következő beolvasási műveletig.

Vonatkozó függvények: **put( )**, **read( )**, **getline( )**.

getline

```
#include <iostream>
istream &getline(char *buf, streamsize num);
istream &getline(char *buf, streamsize num, char
                 delim);
```

A **getline( )** függvény az **istream** osztály tagja.

A **getline(char *_buf_, streamsize _num_)** karaktereket olvas a bemeneti csatornából a *buf* által mutatott pufferbe addig, amíg a beolvasott karakterek száma el nem éri a *num-1*-et, vagy sorvégéhez nem ér. A pufferben megjelenő tömb „nullavégűségéről" a **getline( )** gondoskodik. Az újsor karakter szintén beolvasásra kerül, de a **getline( )** nem helyezi a pufferbe. A függvény egy referenciát ad vissza a csatornára.

A **getline(char *_buf_, streamsize _num_, char _delim_)** változat karaktereket olvas az adott csatornából a *buf* által mutatott tömbbe addig, amíg a beolvasott karakterek száma el nem éri a *num-1*-et, vagy a *delim* paraméterben megadott karakter nem kerül olvasásra vagy a fájl végéhez nem ér. A függvény biztosítja a pufferben megjelenő tömb nullavégűségét. Az elválasztó karakter szintén beolvasásra kerül, de a **getline( )** nem helyezi a pufferbe. A függvény egy referenciát ad vissza a csatornára.

Vonatkozó függvények: **get( )**, **read( )**.

good

```
#include <iostream>
bool good() const;
```

A **good( )** függvény az **ios** osztály tagja.

A **good( )** függvény nullától különböző értékkel tér vissza, ha valamilyen I/O hiba történt az asszociált adatfolyamban; különben visszatérési értéke 0.

Vonatkozó függvények: **bad( )**, **fail( )**, **eof( )**, **clear( )**, **rdstate( )**.

ignore

```
#include <iostream>
istream &ignore(streamsize num = 1, int delim = EOF);
```

Az **ignore( )** függvény az **istream** osztály tagja.

Az **ignore( )** függvény segítségével az bemeneti csatornából karaktereket olvashatunk át. A függvény addig olvassa a karaktereket, amíg azok száma el nem éri a *num* paraméterben megadott korlátot, vagy amíg egy a *delim*-ben (alapértelmezésben EOF, azaz a fájlvége jel) megadott elválasztó karakterhez nem érünk. Az elválasztó karakterhez ér, akkor azt kihagyja a csatornából. A függvény a csatornára mutató pointerrel tér vissza.

Vonatkozó függvények: **get( )**, **getline( )**.

open

```
#include <fstream>
void fstream::open(const char *filename,
        openmode mode = ios::in | ios::out);
void ifstream::open(const char *filename,
        openmode mode = ios::in);
void ofstream::open(const char *filename,
        openmode mode = ios::out | ios::trunc);
```

Az **open( )** függvény az **fstream**, **ifstream**, **ofstream** osztályok tagja.

Az **open( )** függvénnyel az adott csatornát egy konkrét fájlhoz rendelhetjük. A *filename* paraméterben adjuk meg a fájl nevét, amely tartalmazhatja a teljes elérési utat is. A *mode* értéke határozza meg a fájl megnyitásának módját. A *mode*-nak az alábbi értékek kombinációi közül kell kikerülnie:

ios::app
ios::ate
ios::binary
ios::in
ios::out
ios::trunc

Ezen értékek összeköthetők a | „bitenkénti vagy" művelettel.

Az **ios::app** szerepeltetésével megnyitott fájlhoz küldött adatok a fájl eredeti tartalmának végétől kerülnek kiírásra. Ez csak kimenetként megnyitható fájlokra alkalmazható. Az **ios::ate** érték hatására megnyitáskor a fájl végére kerül az állománymutató. Annak ellenére, hogy az **ios::ate** megkeresi a fájl végét, I/O műveletek a fájl bármely részén elképzelhetőek.

Az **ios::binary** érték a fájlt bináris I/O műveletekre nyitja meg. Alapértelmezésben a fájlok szöveges módban kerülnek megnyitásra.

Az **ios::in** értékkel a fájlt olvasásra nyitjuk meg. Az **ios::out** fájlt írásra nyitjuk meg. A **ifstream** osztály használata automatikusan input fájl használatát feltételezi, hasonlóképpen ha egy **ofstream** objektumból történik a hívás, a fájl kimenetként kezelődik, így az **ios::in** és **ios::out** értékek megadása fölösleges.

Az **ios::trunc** érték hatására a megadott nevű fájl korábbi tartalma megsemmisül, a rajta végzett első művelet egy üres fájlt talál.

Ha **open( )** nem tudja megnyitni a kívánt fájlt, a csatorna értéke 0 lesz. Ezért, mielőtt egy fájllal dolgozni kezdenénk, vizsgáljuk meg, hogy az **open( )** sikeres volt-e.

Vonatkozó függvények: **close( )**, **fstream( )**, **ifstream( )**, **ofstream( )**.

Programozási tipp

A régebbi iostream könyvtárban az ***fstream*** *konstruktor nem definiált alapértelmezési értéket a mode paraméternek, azaz a fájlt nem nyitotta meg automatikusan írásra vagy olvasásra. Ezért a régi stílusú könyvtár használatakor, egy csatornát írásra vagy olvasásra történő megnyitásához explicit módon meg kell adnunk az* ***ios::in*** *vagy az* ***ios::out*** *értékek valamelyikét. Az új típusú könyvtárban írt kód régivel való kompatibilitása az előzőek miatt nem áll fenn.*

peek

```
#include <iostream>
int peek();
```

A **peek( )** függvény az **iostream** osztály tagja.

A **peek( )** az adatfolyam következő karakterét adja vissza, kivéve ha a fájl végén vagyunk, ekkor az eredmény **EOF** lesz. A függvény semmilyen körülmények között nem veszi ki a beolvasott karaktert a csatornából.

Vonatkozó függvények: **get( )**.

precision

```
#include <iostream>
streamsize precision() const;
streamsize precision(streamsize p);
```

A **precision( )** függvény az **ios** osztály tagja.

Alapértelmezésben a kimenetre hat tizedesjegy pontossággal kerülnek ki a lebegőpontos számok. A **precision( )** második változatát használva ezen változtathatunk. A *p* paraméter felelős azért, hogy hány tizedesjegy pontossággal jelennek meg adataink a kimeneten. Visszatérési értékként az előző beállítást kapjuk.

Vonatkozó függvények: **width( )**, **fill( )**.

put

```
#include <iostream>
ostream &put(char ch);
```

A **put( )** függvény az **ostream** osztály tagja.

A **put( )** a *ch* paraméterben megadott karaktert írja az adott kimeneti csatornába. Visszatérési értékként egy referenciát kapunk a csatornára.

Vonatkozó függvények: **write( )**, **get( )**.

putback

```
#include <iostream>
istream &putback(char ch);
```

A **putback( )** függvény az **istream** osztály tagja.

A **putback( )** függvény visszaadja a *ch* karaktert az asszociált bemeneti csatornának.

Vonatkozó függvények: **peek( )**.

rdstate

```
#include <iostream>
iostate rdstate() const;
```

Az **rdstate( )** függvény az **ios** osztály tagja.

Az **rdstate( )** függvény az adott csatorna állapotát adja vissza. A C++ I/O rendszere minden aktív csatornához állapotinformációkat rendel, amelyek az egyes I/O műveletek eredményét jellemzik. Az I/O rendszer aktuális állapotát egy integer tárolja, amely az alábbi jelzőbiteket hordozza magában:

Név	*Jelentés*
ios::goodbit	Nem történt hiba
ios::eofbit	Fájl végéhez értünk
ios::failbit	Nem súlyos I/O hiba történt
ios::badbit	Súlyos I/O hiba történt

Ezek a jelzőbitek az **ios**-ban vannak felsorolva.

rdstate( ) 0-t (**ios::goodbit**) ad vissza, amennyiben nem történt hiba. Minden más esetben valamilyen hibabit 1-esre vált.

Vonatkozó függvények: **eof( )**, **good( )**, **bad( )**, **clear( )**, **fail( )**.

read

```
#include <iostream>
istream &read(char *buf, streamsize num);
```

A **read( )** függvény az **istream** osztály tagja.

A **read( )** függvény *num* számú bájtot olvas be az adott bemeneti csatornából és azokat a *buf* által mutatott pufferbe helyezi. Ha a **read( )** a fájl végét még a *num* számú karakter beolvasása előtt eléri, akkor a beolvasás abbamarad és a pufferbe az addig beolvasott karakterek lesznek találhatók (lásd „**gcount**"). A **read( )** egy referenciát ad vissza a csatornára.

Vonatkozó függvények: **gcount( )**, **get( )**, **getline( )**, **write( )**.

seekg, seekp

```
#include <iostream>
istream &seekg(off_type offset, ios::seekdir origin)
istream &seekg(pos_type position);

ostream &seekp(off_type offset, ios::seekdir origin);
ostream &seekp(pos_type position);
```

A **seekg( )** függvény az **istream** a **seekp( )** az **ostream** osztály tagja.

A C++ I/O rendszerében a közvetlen fájlelérést a **seekg( )** és a **seekp( )** függvényekkel valósíthatjuk meg. A közvetlen hozzáférést a C++ úgy biztosítja, hogy minden fájlhoz két mutatót tart fenn: az egyik az ún. *get mutató*, amelyik a következő inputművelet helyét jelöli meg, a másik a *put mutató*, amely a következő outputművelet pozíciójára mutat. Minden alkalommal, amikor valamilyen be- vagy kiviteli művelet történik, a meg-

felelő pointer automatikusan előrébb állítódik. A **seekg( )** és a **seekp( )** függvényekkel lehetőségünk nyílik a fájlok nem szekvenciális elérésére.

A **seekg( )** kétparaméteres változata az *origin* által meghatározott helytől *offset* bájtnyira állítja a get mutatót. Hasonlóképpen a **seekp( )** kétparaméteres változata az *origin* által meghatározott helytől *offset* bájtnyira állítja a put mutatót. Az *offset* paraméter **off_type** típusú, amelynek maximális értékét az szabja meg, hogy mi a lehető legnagyobb megengedett értéke egy ilyen eltolásnak.

Az *origin* paraméter **seekdir** típusú, amely egy enumeráció. Az alábbi értékek valamelyikét kell, hogy felvegye:

Név	*Jelentés*
ios::beg	Fájl eleje
ios::cur	Aktuális pozíció
ios::end	Fájl vége

A **seekg( )** és a **seekp( )** egyparaméteres változatai az állománymutatókat a *position* által meghatározott helyre állítják. Ez az érték egy korábbi **tellg( )**, illetve **tellp( )** hívásból kell hogy származzon. A **pos_type** egy olyan típus, amelynek maximális értékét az szabja meg, hogy mi a lehető legnagyobb megengedett értéke a *position*-nak. Ezek a függvények egy referenciát adnak vissza a csatornára.

Vonatkozó függvények: **tellg( )**, **tellp( )**.

setf

```
#include <iostream>
fmtflags setf(fmtflags flags);
fmtflags setf(fmtflags flags1, fmtflags flags2);
```

A **setf( )** függvény az **ios** osztály tagja.

A **setf( )** függvény mindegyik változata egy adott csatornaformátum jelzőbitjeinek manipulálására szolgál. Az egyes bitek szerepét e fejezet korábbi részében tárgyaltuk.

A **setf( )** első változata a *flag*-ban megadottak szerint állítja be a formátumjelző biteket. (Más jelzőbitre nincs hatással.) Például a **showpos** flaget az alábbi utasítással kapcsolhatjuk be:

```
stream.setf(ios::showpos);
```

ahol a *stream* az az adatfolyam, amelyhez tartozó formátumjelző biteket meg akarjuk változtatni.

Fontos, hogy az **fstream( )**, csak egy adott csatornára vonatkozólag fejti ki hatását. Így nem lenne értelme a **setf( )** önmagában történő hívásának. Minden csatorna rendelkezik saját formátumállapottal.

Több flaget egyszerre a | („bitenkénti vagy") operátor segítségével állíthatunk.

A **setf( )** második verziójában a *flags2* paraméter az átállítható jelzőbiteket határozza meg. A függvény először törli ezeket a biteket, majd a *flag1*-nek megfelelően állítja be őket.

A **setf( )** mindkét változata a csatornához rendelt formátumjelző bitek előző állapotát adja vissza.

Vonatkozó függvények: **unset( )**, **flags( )**.

sync_with_stdio

```
#include <iostream>
static bool sync_with_stdio(bool sync = true);
```

A **sync_with_stdio( )** függvény az **ios** osztály tagja.

A **sync_with_stdio( )** hívásával lehetővé válik a C típusú I/O rendszer és a C++ osztályalapú I/O rendszerének egyidejű használata. A szinkronizációt kikapcsolásához a függvényt **false** értékkel kell meghívni. Visszatérési értékként az előző állapotot kapjuk vissza (a **true** érték a szinkronizáltnak, a **false** a szinkronizálatlannak felel meg).

tellg, tellp

```
#include <iostream>
pos_type tellg();
pos_type tellp();
```

A **tellg( )** függvény az **istream**, míg a **tellp( )** az **ostream** osztály tagja.

A C++ I/O rendszere minden megnyitott fájlhoz két mutatót tart fenn. Az egyik az ún. *get mutató*, amelyik a következő inputművelet helyét jelöli meg, a másik a *put mutató*, amely a következő outputművelet pozíciójára mutat. Minden alkalommal, amikor valamilyen be- vagy kiviteli művelet történik, a megfelelő pointer automatikusan előrébb állítódik. A get mutató, illetve a put mutató aktuális pozícióját rendre a **tellg( )** és a **tellp( )** függvényekkel kérdezhetjük le.

A **pos_type** típust a könyvtár úgy definiálja, hogy képes legyen a lehető legnagyobb megengedett pozíciót is tárolni.

A **tellg( )** és a **tellp( )** függvényekkel nyerjük rendre a későbbi **seekg( )** és **seekp( )** hívások paramétereit is.

Vonatkozó függvények: **seekg( )**, **seekp( )**.

unsetf

```
#include <iostream>
void unsetf(fmtflags flags);
```

Az **unsetf( )** függvény az **ios** osztály tagja.

Az **unsetf( )** függvény a kívánt formátumjelző biteket törli, a többi jelzőbit nem változik.

A törlendő jelzőbiteket a *flag* paraméterben adhatjuk meg.

Vonatkozó függvények: **setf( )**, **flags( )**.

width

```
#include <iostream>
streamsize width() const;
streamsize width(streamsize w);
```

A **width( )** függvény az **ios** osztály tagja.

Az aktuális mezőszélességet úgy kaphatjuk meg, ha **width( )** első változatát hívjuk meg, ekkor a visszatérési érték tartalmazza a mezőszélességet.

A mezőszélesség beállítása a második változattal történhet, ahol a *w* paraméter közvetítésével adjuk meg a kívánt szélességet. A függvény az előzőleg használt mezőszélességet adja vissza.

Vonatkozó függvények: **precision( )**, **fill ( )**.

write

```
#include <iostream>
ostream &write(const char *buf, streamsize num);
```

A **write( )** függvény az **ostream** osztály tagja.

A **write( )** függvény *num* bájtnyit ír az adott kimeni csatornába a *buf* pointer által mutatott pufferből. A függvény egy referenciát ad vissza a csatornára.

Vonatkozó függvények: **read( )**, **put( )**.

14. FEJEZET

A C++ Szabványos Sablon Könyvtára

A C++ szabványosítási folyamata alatt a legnagyobb lépést a Szabványos Sablon Könyvtár vagy STL (az angol Standard Template Library) kifejlesztése és a C++ szerves részévé tétele jelentette. Az STL általános célú genericosztályokat (osztálysablonokat) és függvényeket tartalmaz, amelyek sok népszerű algoritmus és adatszerkezet implementációi. Támogatja a vektorokat, listákat, sorokat, vermeket és az ezeket kezelő rutinokat. Mivel az STL osztálysablon definíciókra épül, az algoritmusok és adatszerkezetek szinte bármilyen adattípussal használhatók.

Az STL nyújtotta szolgáltatások teljes ismertetése – a könyvtár mérete miatt – nem célja e könyvnek. Az itt az ANSI/ISO C++ szabványnak megfelelő STL könyvtár egyes lehetőségeit ismertetjük, előfordulhat azonban, hogy az olvasó által használt fordító ettől eltérő STL változatot támogat. Mindenesetre érdemes megvizsgálni a dokumentáció idevonatkozó részét.

■ *Konténerek, algoritmusok és iterátorok áttekintése*

Az STL három alapelemből áll: *konténerekből, algoritmusokból és iterátorokból.* Ezek elválaszthatatlanok egymástól, megfelelő használatukkal számos programozási problémára kész megoldást kínálnak. Ismertetésük ennek a fejezetrésznek a célja.

Konténerek

A *konténerek* bizonyos objektumok összefogását megvalósító adatszerkezetek. A konténereknek különböző típusai lehetnek: például a **vector** osztály, amely dinamikus tömbhasználatot tesz lehetővé, a **queue** osztály,

amellyel sorokat hozhatunk létre, vagy a **list** osztály, amellyel lineáris listákat készíthetünk és manipulálhatunk. Ezen alapkonténereken kívül az STL ún. asszociatív konténereket is definiál, amelyek kulcsalapú adatok manipulálását biztosítják. Ilyen például a **map** osztály, amellyel egyértelmű kulcsú adatok (kulcs/érték pár) tárolását és adott kulcsú adat kinyerését végezhetjük.

Minden konténertípus tartalmaz egy sor olyan függvényt, amely a típus példányain hat. Például egy lista konténer rendelkezik olyan függvényekkel, amelyekkel törölhetünk, beszúrhatunk és összefésülhetünk listaelemeket.

Algoritmusok

Az algoritmusok a konténereken végeznek műveleteket. Ezek közé tartozik a konténerek tartalmának inicializálása, rendezése és átalakítása. Sok algoritmus ún. sorozatokkal vagy szekvenciákkal dolgozik, amelyek a konténer elemeinek felsorolását tartalmazó struktúra.

Iterátorok

Az iterátorok mutatószerű objektumok. Segítségükkel férhetünk hozzá a konténerek tartalmához. Hasonlóképpen használjuk őket, mint a mutatókat, amikor egy tömb elemeit kívánjuk elérni. Öt különböző típusú iterátor van:

Iterátor	*Megengedett hozzáférés*
Közvetlen hozzáférésű	Értékek tárolásához és kiolvasásához; az elemek közvetlenül elérhetők.
Kétirányú	Értékek tárolásához és kiolvasásához; mozgatás a struktúra elején és végén megengedett.
Elölható	Értékek tárolásához és kiolvasásához; adatmozgatás a struktúra elején.
Bemeneti	Értékek kiolvasásához, tárolás nem megengedett, adatmozgatás a struktúra elején.
Kimeneti	Kizárólag tárolás megvalósítására, kiolvasás nem megengedett; adatmozgatás a struktúra elején.

Általában a több lehetőséget nyújtó iterátor használható a kevesebbet megengedő iterátorok helyett. Például egy elölható iterátorral helyettesíthetünk egy bemeneti iterátort.

Az iterátorokat ugyanúgy kezelhetjük, mint a pointereket. Inkrementálhatjuk és dekrementálhatjuk őket. Alkalmazhatjuk rájuk a * operátort.

Az iterátorokat az **iterator** típus segítségével deklarálhatunk. Az **iterator** típus az egyes konténerekben van definiálva.

A különböző iterátor típusokra a könyv az alábbi elnevezéseket használja:

Elnevezés	*Megfelel*
BiIter	Kétirányú
ForIter	Elölható
InIter	Bemeneti
OutIter	Kimeneti
RandIter	Közvetlen hozzáférésű

Az STL támogatja az ún. fordított irányú iterátorokat is. A fordított irányú iterátorok lehetnek kétirányú vagy közvetlen hozzáférésű iterátorok, amelyek egy lineáris struktúrát a normálissal ellenkező irányban járnak be. Így például, ha egy a struktúra végére mutató fordított irányú iterátort inkrementálunk, az az utolsó előtti elemre fog mutatni.

■ *Allokátorok*

Minden konténer rendelkezik egy rá jellemző *allokátorral*, amely egy **allocator** osztálybeli objektum. Az allokátorok a konténer létrehozásához szükséges memóriát allokálják.

■ *Predikátumok és összehasonlító függvények*

Számos algoritmus és konténer egy speciális típusú függvényt használ: ún. *predikátumot*. A predikátumoknak két változata van: unáris és bináris. Az unáris predikátumnak egy, míg a binárisnak két argumentuma van.

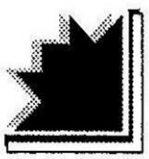
Ezek a függvények logikai értékeket adnak vissza. A pontos viselkedésüket, azaz, hogy mely esetekben adnak vissza igazat és mely esetekben hamisat, a programozó feladata meghatározni. A továbbiakban az unáris predikátum függvényre az **UnPred** típusspecifikátorral utalunk majd. Ha egy függvénynek bináris predikátum paraméterre van szüksége, azt majd a **BinPred** típusazonosító jelzi. A bináris predikátumoknál szükséges az operandusok helyes sorrendjének megtartása (gondoljunk pl. a „kisebb" relációra). Egy konténerhez tartozó predikátumok argumentumtípusa meg kell hogy egyezzen a konténer adatelemeinek típusával.

Vannak olyan algoritmusok és osztályok, amelyeknek azért van szükségük bináris predikátumra, hogy azzal két elem összehasonlítását elvégezhessék. Ezeket a speciális célú predikátumokat összehasonlító függvényeknek is nevezik. Egy összehasonlító függvény, akkor ad vissza logikai igaz értéket (true vagy 0), ha az első operandusa kisebb, mint a második. Az összehasonlító függvények összességére a **Comp** típussal utalunk.

■ *A <utility> és a <functional> fejek*

A különböző STL osztályok használatához szükséges fejek mellett, a C++ standard könyvtárában az STL kiegészítőjeként két fontos fej játszik még fontos szerepet: a **<utility>** és a **<functional>**. Az **<utility>** fejállományban van definiálva a **pair** osztály, amely értékpárok tárolását és manipulálását valósítja meg. A **less( )** generic függvény, amelynek deklarációja a **<functional>** fejben található, azt dönti el, hogy két objektum közül melyik a kisebb.

A **<functional>** fej sablonjai lehetővé teszik, hogy olyan objektumokat hozzunk létre, amelyek egy **operator( )** függvényt definiálnak. Ezeket *függvény objektumoknak* nevezzük és sok esetben függvénypointerek helyett használhatjuk őket. E kézikönyv céljain túlmutat ezen fejek részletes ismertetése.



Programozási tipp

Konténerek, algoritmusok és iterátorok egységet alkotnak, közösen fejtik ki működésüket. Működésüket legegyszerűbben egy példán keresztül érthetjük meg. Az alábbi program a ***vector*** *konténer működését illusztrálja. Egy* ***vector*** *objektum nagyon hasonlít egy tömbre. Az előbbi azonban rendelkezik azzal a jó tulajdonsággal, hogy kézben tudja*

tartani saját tárigényének változásával járó helyfoglalásokat: például mérete futás közben növekedhet. Egy ***vector*** *objektum biztosítja számunkra, hogy bármikor lekérdezzük aktuális méretét, valamint – természetesen – hogy elemeket vegyünk hozzá és töröljünk. Most nézzük az ígért programot.*

```
// a vektorok használatát bemutató rövid példa
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> v; // nulla hosszúságú vektor
                 létrehozása
  int i;

  // a v eredeti méretének kiíratása
  cout << "méret = " << v.size() << endl;

  /* adatok elhelyezése a vektor végébe, a
   vektor mérete megfelelően változik*/
  for(i=0; i<10, i++) v.push_back(i);

  // a v megváltozott méretének kiíratása
  cout << "megváltozott méret = " << v.size()
       << endl;

  // vektor tartalmának indexszel történő elérése
  for(i=0; i<10; i++) cout << v[i] << " ";
  cout << endl;

  // utolsó és első elemének kiíratása
  cout << "első elem = " << v.front() << endl;
  cout << "leghátsó elem = " << v.back() <<
          endl;
```

```
    // iterátor használata
    vector<int>::iterator p = v.begin();
    while(p != v.end()) {
    cout << *p << " ";
    p++;
    }

    return 0;
}
```

A program lefuttatása után a képernyőn az alábbiak lesznek láthatók:

```
méret = 0
megváltozott méret = 10
0 1 2 3 4 5 6 7 8 9
első elem = 0
leghátsó elem = 9
0 1 2 3 4 5 6 7 8 9
```

A program elején egy nulla hosszúságú vektor kerül elkészítésre. A ***push_back( )*** *függvénytaggal értékeket helyezünk a vektor végére. A vektor mérete ennek megfelelően változik. A* ***size( )*** *függvénnyel az aktuális méretet kérdezhetjük le. A vektor, egy normális tömbhöz hasonlóan indexelhető. Az elemek iterátor segítségével érhetők el. A* ***begin( )*** *függvény egy a vektor elejére mutató iterátort ad vissza. Az* ***end( )*** *a vektor végére beállított iterátorral tér vissza.*

Figyeljük meg, hogyan deklaráltuk a ***p*** *iterátort. Az* ***iterator*** *típust számos konténerosztály definiálja.*

■ *Konténerosztályok*

Az STL által definiált konténereket az alábbiakban soroljuk fel.

Konténer	*Leírás*	*Importálandó fej*
bitset	Bitek halmaza	<bitset>
deque	Kétvégű sor	<deque>
list	Lineáris lista	<list>
map	Kulcs/érték pár tárolása, ahol a kulcs egyértelmű.	<map>
multimap	Kulcs/érték pár tárolása, ahol egy kulcshoz több érték is tartozhat.	<map>
multiset	Olyan halmaz, amelyben lehetnek többször előforduló elemek.	<set>
pririty_queue	Prioritási sor	<queue>
queue	Sor	<queue>
set	Halmaz, amelyben minden elem csak egyszer kerül tárolásra.	<set>
stack	Verem	<stack>
vector	Dinamikus tömb	<vector>

A karakterláncokat kezelő **string** osztály is egy konténer, de ezt a 15. fejezet tárgyalja.

Az egyes konténerek működését a következő fejezetrészekben foglaljuk össze. Mivel a konténerek osztálysablonokként (genericosztályokként) vannak implementálva, azaz különböző típus adható meg az adatelemek típusaként (ilyen elemek lesznek a konténerben) és más típusok is szerepelhetnek paraméterként. A leírásokban ezt a típusparamétert a **T** reprezentálja, a **Size** paraméter valamilyen egész típusnak felel meg. Mivel a paraméterként megadott típusok nevei önkényesen megválaszthatók, az ebből származtatott, a tagfüggvények által használt típusok ezektől függnek. Azért, hogy mégis hivatkozni lehessen rájuk, a konténerosztályok deklarálnak néhány predefinit típusnevet a **typedef** segítségével. A leggyakrabban használt ilyen **typedef** típusok a következők:

Név	*Leírás*
size_type	A **size_t** -vel ekvivalens egész típus.
reference	Referencia egy elemre
const_reference	Konstans referencia egy elemre
iterator	Iterátor
const_iterator	Konstans iterátor
reverse_iterator	Fordított irányú iterátor
const_reverse_iterator	Konstans fordított irányú iterátor
value_type	A konténerben tárolt típus
allocator_type	Az allokátor típusa
key_type	Kulcs típusa
key_compare	Két kulcsot összehasonlító függvény típusa
value_compare	Két értéket összehasonlító függvény típusa.

bitset

A **bitset** osztály egy bithalmazok létrehozásához és kezeléséhez szükséges eszközöket biztosítja. Generic specifikációja:

```
template<size_t N> class bitset;
```

ahol *N* a bitset hosszát jelenti bitekben megadva. Az osztályhoz az alábbi konstruktorok tartoznak:

```
bitset( );
bitset(unsigned long bits);
explicit bitset(const string &s, size_t i=0, size_t num= -1);
```

Az első változat egy üres bithalmazt hoz létre. A második egy olyan bithalmazt készít el, amelynek a *bits*-nek megfelelő bitek az elemei. A harmadik konstruktor az *s* string *i*-ik elemétől kezdődően nyeri a kiindulási halmaz elemeit. A string csak 1-eseket és 0-kat tartalmazhat. A stringből csak *num* vagy s.**size( )**-*i* érték kerül felhasználásra, attól függően melyik a kisebb.

A **bitset**-tel használhatók a << és >> I/O operátorok.

A **bitset** osztályban a következő függvénytagok találhatók:

Tag	*Leírás*
bool any( ) const;	True értéket ad vissza, ha a hívó bithalmaz valamely bitje 1, különben false-ot ad vissza.
size_type count( ) const;	Az 1-es bitek számát adja vissza.
bitset<N>&flip( );	Minden bit állapotát ellenkezőre változatja és a ***this**-t adja vissza.
bitset<N>&flip(size_t *i*);	Az *i*-ik helyen található bit állapotát ellenkezőre változatja és visszaadja ***this**-t.
bool none( ) const;	True értéket ad vissza, ha a hívó bithalmazban minden bit 0.
bool operator != (const bitset<N>&op2) const;	True értéket ad vissza, ha a hívó bithalmaz különbözik a jobb oldali operandusként megadottól.
bool operator = = (const bitset<N>&op2) const;	True értéket ad vissza, ha a hívó bithalmaz megegyezik a jobb oldali operandusként megadottal.
bitset<N> &operator & = (const bitset<N>&op2);	Bitenkénti „és" műveletet hajt végre a hívó bithalmaz és a jobb oldali operandusként megadott bithalmaz összetartozó bitjein.
bitset<N> &operator ^ = (const bitset<N>&op2);	Bitenkénti „kizáró vagy" műveletet hajt végre a hívó bithalmaz és a jobb oldali operandusként megadott bithalmaz összetartozó bitjein.
bitset<N> &operator |= (const bitset<N>&op2);	Bitenkénti „vagy" műveletet hajt végre a hívó bithalmaz és a jobb oldali operandusként megadott bithalmaz összetartozó bitjein.
bitset<N>&operator ~=( ) const;	A hívó bithalmaz minden bitjét ellenkezőjére állítja és visszaadja ***this**-t.
bitset<N>&operator<<=(size_t *num*);	A hívó bithalmaz minden bitjét *num* hellyel balra lépteti az eredményt a hívó halmazban hagyja és visszaadja ***this**-t.
bitset<N>&operator>>=(size_t *num*);	A hívó bithalmaz minden bitjét *num* hellyel jobbra lépteti az eredményt a hívó halmazban hagyja és visszaadja ***this**-t.

Tag	*Leírás*
reference operator [ ](size_type *i*);	A hívó bithalmaz *i*-ik elemére ad vissza referenciát.
bitset<N>&reset( );	A hívó bithalmaz minden bitjét törli és visszaadja ***this**-t.
bitset<N>&reset(size_t *i*);	A hívó bithalmaz *i* pozícióján lévő bitet 0-ra állítja és visszaadja ***this**-t.
bitset<N>&set( );	A hívó bithalmaz minden bitjét 1-re állítja és visszaadja ***this**-t.
bitset<N>&set(size_t *i*, int *val*=1);	A hívó bithalmaz *i*-ik pozícióján lévő bitet *val*-ban megadott értékre állítja és visszaadja ***this**-t. A *val*-ban megadott nullától különböző érték 1-nek számít.
size_t size( ) const;	A halmaz által tárolható bitek számát adja vissza.
bool test(size_t *i*);	Az *i*-ik pozíción lévő bit állapotát adja vissza.
string to_string( ) const;	A hívó bithalmaz bitjeinek állapotát string reprezentációban adja vissza
unsigned long to_ulong( ) const;	A hívó bithalmazt konvertálja **unsigned long integer**-ré.

deque

A **deque** osztály kétvégű sorstruktúrák létrehozását és azok karbantartását teszi lehetővé. Generic specifikációja:

```
template <class T, class Allocator=allocator<T>> class deque
```

ahol **T** a **deque** által tárolt adattípus. A **deque** osztály az alábbi konstruktorokkal rendelkezik:

```
explicit deque(const Allocator &a = Allocator( ));
explicit deque(size_type num, const T &val = T ( ),
               const Allocator &a = Allocator( ));
```

```
deque(const deque<T, Allocator> &ob);
template <class InIter> deque(InIter start, InIter end,
                              const Allocator &a=Allocator( ));
```

Az első változat egy üres kétvégű sort hoz létre. A második változat egy olyan kétvégű sort hoz létre, amelynek *num* darab eleme van és mindegyik elemnek a *val* értéket adja. A harmadik konstruktor az *ob* kétvégű sor másolatát készíti el. A negyedik verzió olyan kétvégű sort hoz létre, amely a *start* és az *end* által megadott iterátorok közötti elemeket tartalmazza.

Az alábbi összehasonlító operátorok használhatók a **deque** osztállyal: ==, <, <=, !=, >, >=.

A **deque** az alábbi függvénytagokkal rendelkezik:

Tag	*Leírás*
template<class InIter> void assign(InIter *start*, InIter *end*);	A *start* és az *end* által meghatározott sorozat kerül a kétvégű sorba.
template<class Size, class T> void assign(Size *num*, const T &*val* = T( ));	A kétvégű sor első *num* elemébe *val* értéket másol.
reference at(size_type *i*); const_reference at(size_type *i*) const;	A kétvégű sor *i*-ik elemére ad vissza referenciát.
reference back( ); const_reference back( ) const;	A kétvégű sor utolsó elemére ad vissza referenciát.
iterator begin( ); const_iterator begin( ) const;	A kétvégű sor első elemére mutató iterátort ad vissza.
void clear();	Minden elemet töröl a kétvégű sorból.
bool empty( ) const;	Ha a hívó kétvégű sor üres, true értéket ad vissza, különben false-t.
const_iterator end( ) const; iterator end( );	A kétvégű sor utolsó elemére mutató iterátort ad vissza.
iterator erase(iterator *i*);	Törli az *i* által mutatott elemet és a következő elemre mutató iterátorral tér vissza.

Tag	*Leírás*
iterator erase(iterator *start*, iterator *end*);	Törli a *start* és *end* intervallumba eső elemeket és az utolsó törölt elemet követő elemre mutató iterátort adja vissza.
reference front( ); const_reference front( ) const;	A kétvégű sor első elemére mutató referenciát ad vissza.
allocator_type get_allocator( ) const;	A kétvégű sor allokátorát adja vissza.
iterator insert(iterator *i*, const & *val* = T( ));	Az *i* által meghatározott elem elé beszúrja a *val* elemet. A beszúrt elemre mutató iterátorral tér vissza.
void insert(iterator *i*, size_type *num*, const T & *val*);	Az *i* által mutatott elem elé *num* db. *val* elemet szúr be.
template<class InIter> void insert(iterator *i*, InIter *start*, InIter *end*);	A *start* és az *end* által közrefogott sorozatot szúrja be az *i* által mutatott elem elé.
size_type maxsize( ) const;	A kétvégű sor által tárolható elemek maximális számát adja vissza.
reference operator[ ](size_type *i*) const; const_reference operator[ ](size_type *i*) const;	Referenciát ad vissza az *i*-ik elemre.
void pop_back( );	Kiveszi a kétvégű sor utolsó elemét.
void pop_front( );	Kiveszi a kétvégű sor első elemét.
void push_back(const T &*val*);	Egy elemet helyez *val* értékkel a kétvégű sor végére.
void push_front (const T &*val*);	Egy elemet helyez *val* értékkel a kétvégű sor elejére.
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	A kétvégű sor végére mutató fordított irányú iterátort ad vissza.
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	A kétvégű sor elejére mutató fordított irányú iterátort ad vissza.
void resize(size_type *num*, T *val* = T( ));	A kétvégű sor méretét a *num*-ban megadottra változtatja. Ha ezáltal a méret nő, akkor a sor végére *val* értékkel rendelkező elemek kerülnek.

Tag	*Leírás*
size_type size() const;	A kétvégű sor elemeinek aktuális számát adja vissza.
void swap(deque<T, Allocator> &ob);	A hívó kétvégű sor és az *ob* által mutatott sor tartalmát cseréli ki.

list

A **list** osztály listák létrehozását és kezelését teszi lehetővé. Generic specifikációja:

```
template <class T, class Allocator = allocator<T>> class list
```

ahol a T a listaelemek típusa. A **list** osztály az alábbi konstruktorokkal rendelkezik:

```
explicit list(const Allocator &a = Allocator( ));

explicit list(size_type num, const T &val = T( ),
              const Allocator &a =  Allocator( ));

list(const list<T, Allocator> &ob);

template<class InIter>list(InIter start, InIter end,
                           const Allocator &a = Allocator( ));
```

Az első változat egy üres listát hoz létre. A második változat egy olyan listát hoz létre, amelynek *num* darab eleme van és mindegyik elemnek a *val* értéket adja. A harmadik konstruktor az *ob* lista másolatát készíti el. A negyedik verzió olyan listát hoz létre, amely a *start* és az *end* által megadott iterátorok közötti elemeket tartalmazza.

Az alábbi összehasonlító operátorok használhatók a **list** osztállyal: ==, <, <=, !=, >, >=.

A **list** osztály az alábbi függvénytagokat tartalmazza:

Tag	*Leírás*
template<class InIter> void assign(InIter *start*, InIter *end*);	A listába a *start* és az *end* közé eső sorozat kerül.
template<class Size, class T> void assign(Size *num*, const T &*val* = T( ));	A lista első *num* darab elemébe a *val* érték kerül.
reference back( ); const_reference back( ) const;	Visszaad egy referenciát a lista utolsó elemére.
iterator begin( ); const_iterator begin( ) const;	A lista első elemére mutató iterátort ad vissza.
void clear( );	Kiüríti a listát.
bool empty( ) const;	Ha lista üres true értéket ad vissza, különben false-t.
iterator end( ); const_iterator end( ) const;	A lista utolsó elemére mutató iterátort ad vissza.
iterator erase(iterator *i*);	Az *i* által mutatott elemet törli a listából, a törölt elem utáni elemre mutató iterátort ad vissza.
iterator erase(iterator *start*, iterator *end*);	Az *end* és a *start* iterátorok által meghatározott elemek között elhelyezkedő összes elemet törli a listából, és az utolsó törölt elem utáni elemre mutató iterátort ad vissza.
reference front( ); const_reference front( ) const;	Visszaad egy referenciát a lista első elemére.
allocator_type get allocator( ) const;	A lista allokátorát adja vissza.
iterator insert(iterator *i*, const T &*val* = T( ));	Az *i* által meghatározott elem elé szúr be egy listaelemet *val* értékkel, az erre mutató iterátorral tér vissza.
void insert(iterator i, size_type *num*, const T &*val*);	Az *i* által meghatározott elem elé szúr be *num* darab listaelemet *val* értékkel, az erre mutató iterátorral tér vissza.
template<class InIter> void insert(iterator *i*, InIter *start*, InIter *end*);	Az *i* által meghatározott elem elé szúrja be a *start* és az *end* által meghatározott sorozatot.

Tag	*Leírás*
size_type max_size( ) const;	A listában tárolható elemek maximális számát adja vissza.
void merge(list<T, Allocator>&*ob*); template<class Comp> void merge(list<T, Allocator>&*ob*, Comp *cmpfn*);	Két rendezett listát fésül össze: a hívó listát és az *ob* által mutatott listát. Az eredmény is rendezett lesz. Rendezés után az *ob* lista kiürül. A második változatban egy összehasonlító függvényt is megadhatunk, amellyel megmondhatjuk, hogy két elem közül, melyik a kisebb.
void pop_back( );	A lista utolsó elemét törli.
void pop_front( );	A lista első elemét törli.
void push_back(const T &*val*);	Egy elemet fűz a lista végére a *val*-ban megadott értékkel.
void push_front(const T &*val*);	Egy elemet fűz a lista elejére a *val*-ban megadott értékkel.
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	A lista végére mutató fordított irányú iterátort ad vissza.
void remove(const T &*val*);	A *val* értékkel rendelkező listaelemeket törli a listából.
template<class UnPred> void remove_if(UnPred *pr*);	Azon elemeket törli a listából, amelyekre az unáris *pr* predikátum igaz értéket vesz fel.
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	A lista első elemére mutató fordított irányú iterátort ad vissza.
void resize(size_type *num*, T *val* = T( ));	A lista méretét a *num* által meghatározottra változtatja. Ha ezáltal a lista nőtt, akkor az új elemek *val* értéket kapnak.
void reverse( );	Megfordítja a hívó listát.
size_type size( ) const;	A listában tárolt elemek aktuális számát adja vissza.

void sort( );
template<class Comp>
void sort(Comp *cmpfn*);
	Rendezi a listát.
	Rendezi a listát a *cmpfn* összehasonlító függvényt használva, amely megmondja, hogy két elem közül, melyik a kisebb.

void splice(iterator *i*, list<T, Allocator>&*ob*);	Az *ob* listát a hívó lista *i*-ik pozíciójára szúrja be. A művelet után *ob* lista üres lesz.

Tag	*Leírás*
void splice(iterator *i*, list<T, Allocator>&*ob*, iterator *el*);	Az *el* elemet törli az *ob* listából és befűzi a hívó lista *i*-ik pozíciójára.
void splice(iterator *i*, list<T, Allocator>&*ob*, iterator start, iterator *end*);	Az a *start* és *end* által definiált elemeket törli az *ob* listából és beszúrja a hívó lista *i*-ik pozíciójára.
void swap(list<T, Allocator>&*ob*);	A hívó lista tartalmát kicseréli az *ob* tartalmával.

void unique( );
template<class BinPred>
void unique(BinPred *pr*);
	A többször előforduló elemek közül csak egyet hagy meg. A második változatban megadhatjuk, hogy két elem mikor egyezik meg.

map

A **map** osztály kulcs/érték párokat tartalmazó asszociatív konténerek megvalósítását és kezelését teszi lehetővé. Gerincspecifikációja:

```
template<class Key, class T, class Comp = less<Key>,
   class Allocator = allocator<T>>class map
```

ahol **Key** a kulcsok adattípusát, **T** a tárolt értékek típusát jelöli. **Comp** két kulcs összehasonlítását végző függvény. A **map** osztály az alábbi konstruktorokkal rendelkezik:

```
explicit map(const Comp &cmpfn = Comp( ),
             const Allocator &a = Allocator( ));

map(const map<Key, T, Comp, Allocator>&ob;

template<class InIter>map(InIter start, InIter end,
                          const Comp &cmpfn =Comp( ),
                          const Allocator &a = Allocator( ));
```

Az első változat egy üres kulcs/érték táblázatot (továbbiakban táblázatot) hoz létre. A második konstruktor az *ob* által mutatott táblázatról

készít másolatot. A harmadik verzió olyan táblázatot hoz létre, amely a *start* és az *end* által megadott iterátorok közötti elemeket tartalmazza. Az *cmpfn* paraméterként megadott függvény a táblázat rendezéséért felelős.

Az alábbi összehasonlító operátorok használhatók a **map** osztállyal: ==, <, <=, !=, >, >=.

A **map** osztály függvénytagjait az alábbi táblázat foglalja össze. A prototípusokban a **key_type** a kulcs típusát jelenti, míg a **value_type** a **pair<Key, T>** pár típusa.

Tag	*Leírás*
iterator begin( ); const_iterator begin( ) const;	A táblázat első elemére mutató iterátort ad vissza.
void clear( );	Törli a táblázatot.
size_type count(const key_type &*k*) const;	Visszaadja, hogy a *k* kulcs hányszor fordul elő a táblázatban.
bool empty( ) const;	Ha a hívó táblázat üres, true értéket ad vissza, különben false-t.
iterator end(); const_iterator end( ) const;	A táblázat utolsó elemére mutató iterátort ad vissza.

pair<iterator, iterator>
equal_range(const key_type &*k*);
pair<const_iterator, const_iterator>
equal_range(const key_type &*k*) const;
	Egy iterátor párt ad vissza, amelyek az első ill. az utolsó olyan elemre mutatnak, amelyek kulcsai megegyeznek *k*-val.

void erase(iterator *i*);	Az *i* által mutatott elemet törli.
void erase(iterator *start*, iterator *end*);	Törli a *start* és *end* iterátorok által mutatott elemek között elhelyezkedő összes elemet.
size_type erase(const key_type &*k*);	A táblázatból törli a *k* kulcsértékkel rendelkező elemeket.
iterator find(const key_type &*k*); const_iterator find(const key_type &*k*) const;	A megadott *k* kulcsú elemet megkeresi a táblázatban és az arra mutató iterátorral tér vissza. Ha nem talál ilyen elemet, akkor a visszaadott iterátor a táblázat végére fog mutatni.

Tag	*Leírás*
allocator_type get_allocator( ) const;	A táblázat allokátorát adja vissza.
iterator insert(iterator *i*, const value_type &*val*);	A *val* értékű elemet szúrja be a táblázat *i* által megadott eleme után, a beszúrt elemre mutató iterátort adja vissza.
template<class InIter> void insert(InIter *start*, InIter *end*);	Elemek egy sorozatát szúrja be.
pair<iterator, bool> insert(const value_type &*val*);	A hívó táblázatba szúr be *val* értékkel rendelkező elemet. Az elemre mutató iterátort adja vissza. Az elem csak akkor kerül be a táblázatba, ha még nincs ilyen elem. Ha a beszúrás megtörtént, pair<iterator, true>-t kapunk vissza, különben pair<iterator, false>-t.
key_compare key_comp( ) const;	A kulcsok összehasonlítását végző függvényobjektumot adja vissza.

iterator lower_bound(const key_type &*k*);
const_iterator
lower_bound(const key_type &*k*) const;
	A táblázat első olyan elemére mutató iterátort adja vissza, amelynek kulcsértéke nagyobb vagy egyenlő, mint a megadott *k*.

size_type max_size() const;	A táblázatban tárolható elemek maximális számát adja vissza.
reference operator[ ](const key_type &*i*);	Az *i* által meghatározott elemre ad vissza referenciát, ha ez az elem nem létezik, akkor bekerül a táblázatba.
reverse_iterator rbegin(); const_reverse_iterator rbegin() const;	A táblázat végére mutató fordított irányú iterátort ad vissza.
reverse_iterator rend(); const_reverse_iterator rend() const;	A táblázat elejére mutató fordított irányú iterátort ad vissza.
size_type size() const;	A táblázatban található elemek aktuális számát adja vissza.
void swap(map<Key, T, Comp, Allocator>&*ob*);	Kicseréli a hívó táblázat és az *ob* által mutatott táblázat tartalmát.

iterator upper_bound(const key_type &*k*);
const_iterator
upper_bound(const key_type &*k*) const;
	A táblázat első olyan elemére mutató iterátort adja vissza, amelynek kulcsértéke kisebb, mint a megadott *k*.

value_compare value_comp() const;	Visszaadja az értékek összehasonlítását végző függvény-objektumot.

multimap

A **multimap** osztály olyan asszociatív konténerek megvalósítását és kezelését támogatja, amelyekben kulcs/érték párokat tárolhatunk és egy kulcshoz több érték tartozhat. Generic specifikációja:

```
template<class Key, class T, class Comp = less<Key>,
   class Allocator = allocator<T>>class multimap
```

ahol a **Key** a kulcsok, míg **T** a tárolt értékek adattípusa. **Comp** két kulcs összehasonlítását végző függvény. A **multimap** osztály az alábbi konstruktorokkal rendelkezik:

```
explicit multimap(const Comp &cmpfn = Comp( ),
                  const Allocator &a = Allocator( ));

multimap(const multimap<Key, T, Comp, Allocator>&ob;

template<class InIter>multimap(InIter start, InIter end,
                               const Comp &cmpfn =Comp( ),
                               const Allocator &a = Allocator( ));
```

Az első változat egy üres kulcs/érték táblázatot (továbbiakban táblázatot) hoz létre. A második konstruktor az *ob* által mutatott táblázatról készít másolatot. A harmadik verzió olyan táblázatot hoz létre, amely a *start* és az *end* által megadott iterátorok közötti elemeket tartalmazza. Az *cmpfn* paraméterként megadott függvény a táblázat rendezéséért felelős.

Az alábbi összehasonlító operátorok használhatók a **multimap** osztállyal: ==, <, <=, !=, >, >=.

A **multimap** osztály függvénytagjait az alábbi táblázat foglalja össze. A prototípusokban a **key_type** a kulcs típusát jelenti, míg a **value_type** a **pair<Key, T>** pár típusa.

Tag	*Leírás*
iterator begin( ); const_iterator begin( ) const;	A táblázat első elemére mutató iterátort ad vissza.
void clear( );	Törli a táblázatot.
size_type count(const key_type &*k*) const;	Visszaadja, hogy a *k* kulcs hányszor fordul elő a táblázatban.
bool empty( ) const;	Ha a hívó táblázat üres, true értéket ad vissza, különben false-t.
iterator end( ); const_iterator end( ) const;	A táblázat utolsó elemére mutató iterátort ad vissza.

pair<iterator, iterator>
equal_range(const key_type &*k*);
pair<const_iterator, const_iterator>
equal_range(const key_type &*k*) const;
	Egy iterátor párt ad vissza, amelyek az első ill. az utolsó olyan elemre mutatnak, amelyek kulcsai megegyeznek *k*-val.

void erase(iterator *i*);	Az *i* által mutatott elemet törli.
void erase(iterator *start*, iterator *end*);	Törli a *start* és *end* iterátorok által mutatott elemek között elhelyezkedő összes elemet.
size_type erase(const key_type &*k*);	A táblázatból törli a *k* kulcsértékkel rendelkező elemeket.
iterator find(const key_type &*k*); const_iterator find(const key_type &*k*) const;	A megadott *k* kulcsú elemet megkeresi a táblázatban és az arra mutató iterátorral tér vissza. Ha nem talál ilyen elemet, akkor a visszaadott iterátor a táblázat végére fog mutatni.
allocator_type get_allocator( ) const;	A táblázat allokátorát adja vissza.
iterator insert(iterator *i*, const value_type &*val*);	A *val* értékű elemet szúrja be a táblázat *i* által megadott eleme után, a beszúrt elemre mutató iterátort adja vissza.
template<class InIter> void insert(InIter *start*, InIter *end*);	Elemek egy sorozatát szúrja be.

Tag	*Leírás*
iterator insert(const value_type &*val*);	Az elemre mutató iterátort adja vissza. Az elem csak akkor kerül be a táblázatba, ha még nincs ilyen elem. Ha a beszúrás megtörtént, pair<iterator, true>-t kapunk vissza, különben pair<iterator, false>-t.
key_compare key_comp( ) const;	A kulcsok összehasonlítását végző függvény-objektumot adja vissza.

iterator lower_bound(const key_type &*k*);
const_iterator
lower_bound(const key_type &*k*) const;
	A táblázat első olyan elemére mutató iterátort adja vissza, amelynek kulcsértéke nagyobb vagy egyenlő, mint a megadott *k*.

size_type max_size( ) const;	A táblázatban tárolható elemek maximális számát adja vissza.
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	A táblázat végére mutató fordított irányú iterátort ad vissza.
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	A táblázat elejére mutató fordított irányú iterátort ad vissza.
size_type size( ) const;	A táblázatban található elemek aktuális számát adja vissza
void swap(map<Key, T, Comp, Allocator>&*ob*);	Kicseréli a hívó táblázat és az *ob* által mutatott táblázat tartalmát.

iterator upper_bound(const key_type &*k*);
const_iterator
upper_bound(const key_type &*k*) const;
	A táblázat első olyan elemére mutató iterátort adja vissza, amelynek kulcsértéke kisebb, mint a megadott *k*.

value_compare value_comp( ) const;	Visszaadja az értékek összehasonlítását végző függvényobjektumot.

multiset

A **multiset** osztály olyan halmaz létrehozását és manipulálását támogatja, amelyben az elemek többszörös multiplicitással szerepelhetnek. Megvalósítása nem egyértelmű kulcsok segítségével történik. A kulcsok itt a tárolt elemek szerepét játsszák, így amikor a kulcsokra történik utalás, a halmaz elemeire kell gondolnunk. Generic specifikációja:

```
template<class Key, class Comp = less<Key>,
   class Allocator = allocator<Key>>class multiset
```

ahol a **Key** a kulcsok adattípusa. A **Comp** két kulcs összehasonlítását végző függvény típusa. A **multiset** az alábbi konstruktorokkal rendelkezik:

```
explicit multiset(const Comp &cmpfn = Comp( ),
                  const Allocator &a = Allocator( ));

multiset(const multiset<Key, Comp, Allocator>&ob);

template<class InIter>multiset(InIter start, InIter end,
                               const Comp &cmpfn =Comp( ),
                               const Allocator &a = Allocator( ));
```

Az első változat egy üres halmazt hoz létre. A második konstruktor az *ob* által mutatott halmazról készít másolatot. A harmadik verzió olyan halmazt hoz létre, amely a *start* és az *end* által megadott iterátorok közötti elemeket tartalmazza. Az *cmpfn* paraméterként megadott függvény a halmaz rendezéséért felelős.

Az alábbi összehasonlító operátorok használhatók a **multiset** osztállyal: ==, <, <=, !=, >, >=.

A **multiset** osztály függvénytagjait az alábbi táblázat foglalja össze. A leírásokban mind a **key_type**, mind a **value_type** a **Key** név **typedef** utasítással átnevezett változatai.

Tag	*Leírás*
iterator begin( ); const_iterator begin( ) const;	A halmaz első elemére mutató iterátort ad vissza.
void clear( );	Kiüríti a halmazt.
size_type count(const key_type &*k*) const;	Visszaadja, hogy a *k* kulcs hányszor fordul elő a halmazban.
bool empty( ) const;	Ha a hívó halmaz üres, true értéket ad vissza, különben false-t.
iterator end( ); const_iterator end( ) const;	A halmaz utolsó elemére mutató iterátort ad vissza.
pair<iterator, iterator> equal_range(const key_type &*k*) const;	Egy iterátor párt ad vissza, amelyek az első ill. az utolsó olyan elemre mutatnak, amelyek a megadott kulccsal rendelkeznek.
void erase(iterator *i*);	Az *i* által mutatott elemet törli.
void erase(iterator *start*, iterator *end*);	Törli a *start* és *end* iterátorok által mutatott elemek között elhelyezkedő összes elemet.
size_type erase(const key_type &*k*);	A halmazból törli a *k* kulcsértékkel rendelkező elemeket. A törölt elemek számát adja vissza.
iterator find(const key_type &*k*) const;	A megadott *k* kulcsú elemet megkeresi a halmazban és az arra mutató iterátorral tér vissza. Ha nem talál ilyen elemet, akkor a visszaadott iterátor a halmaz végére fog mutatni.
allocator_type get_allocator( ) const;	A halmaz allokátorát adja vissza.
iterator insert(iterator *i*, const value_type &*val*);	A *val* elemet szúrja be a halmaz *i* által megadott eleme után, a beszúrt elemre mutató iterátort adja vissza.
template<class InIter> void insert(InIter *start*, InIter *end*);	Elemek egy sorozatát szúrja be.
iterator insert(const value_type&*val*);	A *val* elemet szúrja be a halmazba és visszaadja az arra mutató iterátort.
key_compare key_comp( ) const;	A kulcsok összehasonlítását végző függvényobjektumot adja vissza.

Tag	*Leírás*
iterator lower_bound(const key_type &*k*) const;	A halmaz első olyan elemére mutató iterátort adja vissza, amelynek kulcsértéke nagyobb vagy egyenlő, mint a megadott *k*.
size_type max_size( ) const;	A halmazban tárolható elemek maximális számát adja vissza.
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	A halmaz végére mutató fordított irányú iterátort ad vissza.
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	A halmaz elejére mutató fordított irányú iterátort ad vissza.
size_type size( ) const;	A halmazban található elemek aktuális számát adja vissza.
void swap(map<Key, Comp, Allocator>&*ob*);	Kicseréli a hívó halmaz és az *ob* által mutatott halmaz tartalmát.
iterator upper_bound(const key_type &*k*) const	A halmaz első olyan elemére mutató iterátort adja vissza, amelynek kulcsértéke kisebb, mint a megadott *k*.
value_compare value_comp( ) const;	Visszaadja az értékek összehasonlítását végző függvényobjektumot.

queue

A **queue** osztály sorok létrehozását és manipulálását teszi lehetővé. Generic specifikációja:

```
template<class T, class Container = deque<T>>class queue
```

ahol T a tárolt adat típusa. A **Container** a sort tartalmazó konténer típusa. A **queue** osztály az alábbi konsruktorral rendelkezik:

```
explicit queue(const Container &cnt = Container( ));
```

A **queue( )** konstruktor egy üres sort hoz létre. Alapértelmezésben konténerként a **deque**-t használja, de a sor elemihez csak az „első elem be,

első elem ki" (ezt gyakran FIFO-nak hívják az angol „first-in, first-out"-ból) séma szerint férhetünk hozzá. A **list** osztály is használható a sor konténereként. A paraméterként megadott konténer a **queue** osztály védett **c** mezőjébe kerül be.

Az alábbi összehasonlító operátorok használhatók a **queue** osztállyal: ==, <, <=, !=, >, >=.

A **queue** az alábbi tagfüggvényeket tartalmazza:

Tag	*Leírás*
value_type &back(); const value_type &back( ) const;	Referenciát ad vissza a sor utolsó elemére.
bool empty( ) const;	Ha a hívó sor üres, true értéket ad vissza, egyébként false-t.
value_type &front( ); const value_type &front( ) const;	Referenciát ad vissza a sor első elemére.
void pop( );	Kiszedi az első elemet a sorból.
void push(const T&*val*);	Egy elemet helyez a sorba a *val*-ban megadott értékkel.
size_type size( ) const;	A sorban található elemek aktuális számát adja vissza.

priority_queue

A **priority_queue** osztály prioritási sorok létrehozását és karbantartását teszi lehetővé. Generic specifikációja:

```
template<class T, class Container = vector<T>,
    class Comp = less<Container::value_type>>
    class priority_queue
```

ahol **T** a sorban tárolt adat típusa. A **Container** a sort tartalmazó konténer típusa.

A **Comp** típus azon függvényt adja meg, amely eldönti, hogy a prioritási sor két eleme közül, melyiknek alacsonyabb a prioritása. A **priority_queue** az alábbi konstruktorokkal rendelkezik:

```
explicit priority_queue(const Comp &cmpfn = Comp( ),
                        Container &cnt = Container( ) );

template<class InIter> priority_queue(InIter start, InIter end,
                        const Comp &cmpfn = Comp( ),
                        Container &cnt = Container( ));
```

Az első **priority_queue( )** konstruktor egy üres prioritási sort hoz létre. A második változat olyan prioritási sort készít, amely a *start* és az *end* által megadott iterátorok közötti elemeket tartalmazza. Alapértelmezésben **vector** objektumot használ konténerként, de megadható **deque** is. A paraméterként megadott konténer a **priority_queue** osztály védett **c** mezőjébe kerül be.

A **priority_queue** osztály az alábbi függvénytagokat tartalmazza:

Tag	*Leírás*
bool empty( ) const;	Ha a hívó prioritási sor üres true értéket ad vissza, különben false-t.
void pop( );	Az első elemet kiveszi a prioritási sorból.
void push(const T &*val*);	Egy elemet helyez a sorba a *val*-ban megadott értékkel.
size_type size( ) const;	A sorban található elemek aktuális számát adja vissza.
value_type &top( ); cost value_type &top( ) const;	A legmagasabb prioritással rendelkező elemet adja vissza. Az elemet nem veszi ki a sorból.

set

A **set** osztály olyan halmazok létrehozását és manipulálását támogatja, amelyekben egy elem csak egyszer fordul elő. Megvalósítása kulcsok segítségével történik. A kulcsok itt a tárolt elemek szerepét játsszák, így amikor a kulcsokra történik utalás, a halmaz elemeire kell gondolnunk. Generic specifikációja:

```
template<class Key, class Comp = less<Key>,
   class Allocator = allocator<Key>>class set
```

ahol a **Key** a kulcsok (vagyis az elemek) adattípusa. A **Comp** két kulcs összehasonlítását végző függvény típusa. A **set** az alábbi konstruktorokkal rendelkezik:

```
explicit set(const Comp &cmpfn = Comp( ),
             const Allocator &a = Allocator( ));
set(const set<Key, Comp, Allocator>&ob);

template<class InIter>set(InIter start, InIter end,
                          const Comp &cmpfn =Comp( ),
                          const Allocator &a = Allocator( ));
```

Az első változat egy üres halmazt hoz létre. A második konstruktor az *ob* által mutatott halmazról készít másolatot. A harmadik verzió olyan halmazt hoz létre, amely a *start* és az *end* által megadott iterátorok közötti elemeket tartalmazza. Az *cmpfn* paraméterként megadott függvény a halmaz rendezéséért felelős.

Az alábbi összehasonlító operátorok használhatók a **multiset** osztállyal: ==, <, <=, !=, >, >=.

A **multiset** osztály függvénytagjait az alábbi táblázat foglalja össze. A leírásokban mind a **key_type**, mind a **value_type** a **Key** név **typedef** utasítással átnevezett változatai.

Tag	*Leírás*
iterator begin( ); const_iterator begin() const;	A halmaz első elemére mutató iterátort ad vissza.
void clear( );	Kiüríti a halmazt.
size_type count(const key_type &*k*) const;	Visszaadja, hogy a *k* kulcs hányszor fordul elő a halmazban.
bool empty( ) const;	Ha a hívó halmaz üres, true értéket ad vissza, különben false-t.
const_iterator end( ) const; iterator end( );	A halmaz utolsó elemére mutató iterátort ad vissza.
pair<iterator, iterator> equal_range(const key_type &*k*) const;	Egy iterátor párt ad vissza, amelyek az első, ill. az utolsó olyan elemre mutatnak, amelyek a megadott kulccsal rendelkeznek.
void erase(iterator *i*);	Az *i* által mutatott elemet törli.
void erase(iterator *start*, iterator *end*);	Törli a *start* és *end* iterátorok által mutatott elemek között elhelyezkedő összes elemet.
size_type erase(const key_type &*k*);	A halmazból törli a *k* kulcsértékkel rendelkező elemeket. A törölt elemek számát adja vissza.
iterator find(const key_type &*k*) const;	A megadott *k* kulcsú elemet megkeresi a halmazban és az arra mutató iterátorral tér vissza. Ha nem talál ilyen elemet, akkor a visszaadott iterátor a halmaz végére fog mutatni.
allocator_type get_allocator( ) const;	A halmaz allokátorát adja vissza.
iterator insert(iterator *i*, const value_type &*val*);	A *val* elemet szúrja be a halmaz *i* által megadott eleme után, a beszúrt elemre mutató iterátort adja vissza.
template<class InIter> void insert(InIter *start*, InIter *end*);	Elemek egy sorozatát szúrja be.
pair<iterator, bool> insert(const value_type &*val*);	A hívó halmazba szúr be *val* értékkel rendelkező elemet. Az elemre mutató iterátort adja vissza. Az elem csak akkor kerül be a halmazba, ha még nincs ilyen elem. Ha a beszúrás megtörtént pair<iterator, true>-t kapunk vissza, különben pair<iterator, false>-t.

Tag	*Leírás*
iterator lower_bound(const key_type &*k*) const;	A halmaz első olyan elemére mutató iterátort adja vissza, amelynek kulcsértéke nagyobb vagy egyenlő, mint a megadott *k*.
key_compare key_comp( ) const;	A kulcsok összehasonlítását végző függvényobjektumot adja vissza.
size_type max_size( ) const;	A halmazban tárolható elemek maximális számát adja vissza.
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	A halmaz végére mutató fordított irányú iterátort ad vissza.
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	A halmaz elejére mutató fordított irányú iterátort ad vissza.
size_type size( ) const;	A halmazban található elemek aktuális számát adja vissza.
void swap(map<Key, Comp, Allocator>&*ob*);	Kicseréli a hívó halmaz és az *ob* által mutatott halmaz tartalmát.
iterator upper_bound(const key_type &*k*) const	A halmaz első olyan elemére mutató iterátort adja vissza, amelynek kulcsértéke kisebb, mint a megadott *k*.
value_compare value_comp( ) const;	Visszaadja az értékek összehasonlítását végző függvényobjektumot.

stack

A **stack** osztállyal vermeket hozhatunk létre. Generic specifikációja:

template<class T, class Container= deque<T>>class stack

ahol T a veremben tárolt adatok típusa. A **Container** a vermet megvalósító konténer típusát jelöli, ez alapértelmezésben a **deque**, de a verem elemeihez a csak az „utolsó elem be, első elem ki" (ezt gyakran LIFO-nak hívják az angol „last -in, first-out"-ból) séma szerint férhetünk hozzá. A konténer a **stack** védett **c** mezejében jelenik meg.

Az alábbi összehasonlító operátorok használhatók a **stack** osztállyal: ==, <, <=, !=, >, >=.

A **stack** az alábbi tagfüggvényeket tartalmazza:

Tag	*Leírás*
bool empty( ) const;	Ha a hívó verem üres true értéket ad vissza, különben false-t.
void pop( );	A kiveszi a legfelső elemet a veremből.
void push(const T &*val*);	A *val* értékű elemet a verembe teszi.
size_type size( ) const;	A veremben található elemek aktuális számát adja vissza.
value_type &top( ); const value_type &top( ) const;	A verem tetején lévő elemre ad vissza egy referenciát, ez egyben a konténer utolsó eleme. A verem tartalma nem változik.

vector

A **vector** osztály lehetővé teszi a dinamikus tömbök használatát. Generic specifikációja:

```
template<class T, class Allocator = allocator<T>>
  class vector
```

ahol **T** a vektor elemeinek típusát, az **Allocator** a allokátor típusát jelöli. A **vector** osztály az alábbi konstruktorokat definiálja:

```
explicit vector (const Allocator &a = Allocator( ));

explicit vector (size_type num, const T &val = T ( ),
                 const Allocator &a = Allocator( ));

vector(const vector<T, Allocator> &ob);

template<class InIter> vector(InIter start, InIter end,
                              const Allocator &a = Allocator( ));
```

Az első változat egy üres vektort hoz létre. A második egy olyan *num* hosszúságú vektort készít el, amelynek minden eleme *val* értékkel rendelkezik. A harmadik konstruktor változat lemásolja az *ob* által mutatott vektort. Végül a negyedik konstruktor egy olyan vektort állít elő, amely a *start* és az *end* iterátorok által meghatározott sorozatot másolja a vektorba.

Az alábbi összehasonlító operátorok definiáltak a **vector** osztályban: ==, <, <=, !=, >, >=.

A **vector** osztály az alábbi függvénytagokat tartalmazza:

Tag	*Leírás*
template<class InIter> void assign(InIter *start*, InIter *end*);	A *start* és az *end* iterátorok által közrefogott sorozatot adja értékül a vektornak.
template<class Size, class T> void assign(Size *num*, const T &*val* = T ( ));	A hívó vektor értéke egy *num* darab csupa *ch* karakterből álló vektor lesz.
reference at(size_type *i*); const_reference at(size_type *i*) const;	Az *i*-ik elemre ad vissza referenciát.
reference back( ); const_reference back( ) const;	A vektor utolsó elemére ad vissza referenciát.
iterator begin( ); const_iterator begin( ) const;	A vektor első elemére mutató iterátort ad vissza.
size_type capacity( ) const;	A vektor aktuális kapacitását adja vissza. Ez nem más, mint azon elemek maximális száma, amelyek újabb allokáció nélkül tárolhatók a vektorban
void clear( );	A vektor összes elemének törlése és null-hosszúságúvá tétele.
bool empty( ) const;	Ha a vektor üres true értéket ad vissza, egyébként false-t.
iterator end( ); const_iterator end( ) const;	A vektor végére mutató iterátort ad vissza.
iterator erase(iterator *i*);	A *i* iterátor által mutatott elemet veszi ki a vektorból.
iterator erase(iterator *start*, iterator *end*);	A *start* és *end* iterátorok közti elemeket szedi ki a vektorból.

Tag	*Leírás*
reference front( ); const_reference front( ) const;	A vektor első elemére ad vissza referenciát.
allocator_type get_allocator( ) const;	A vektor allokátorát adja vissza.
iterator insert(iterator *i*, const T &*val*= T( ));	A *i* iterátor által képviselt pozíció elé egy *val* értékkel rendelkező elemet szúr be.
void insert(iterator *i*, size_type *num*, const T &*val*);	A *i* iterátor által képviselt pozíció elé *num* darab *val* értékkel rendelkező elemet szúr be.
template<class InIter> void insert(iterator *i*, InIter *start*, InIter *end*);	A *start* és az *end* iterátorok által közrefogott sorozatot szúrja be az *i* iterátornak megfelelő pozíció elé.
size_type max_size( ) const;	A vektor által tárolható elemek maximális számát adja vissza.
reference operator[ ](size_type *i*) const; const_ reference operator[ ](size_type *i*) const;	Az *i* által meghatározott elemre ad vissza referenciát
void pop_back( );	Az utolsó elemet törli a vektorból.
void push_back(const T &*val*);	A vektor végére *val* értékű elemet szúr be.
reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	A vektor végére mutató fordított irányú iterátort ad vissza.
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	A vektor elejére mutató fordított irányú iterátort ad vissza.
void reserve(size_type *num*);	A vektor kapacitását állítja legalább *num*-ra.
void resize(size_type *num*, T *val* = T( ));	A vektor méretét *num* hosszúságúra változtatja. Ha ezáltal a vektor nőtt, akkor a vektor végén keletkező új helyek *val* értékkel töltődnek fel.
size_type size( ) const;	A vektorban tárolt aktuális elemek számát adja vissza.
void swap(vector<T, Allocator>&*ob*);	A hívó vektor és az *ob* vektor elemeit cseréli ki.

Az STL az előbb felsorolt függvényeken túl, további két művelettel támogatja a logikai értékeket tartalmazó vektorokat:

Tag	*Leírás*
void flip( );	Minden bitet átforgat (0-ból 1-et és 1-ből 0-át csinál).
static void swap(reference *i*, reference *j*);	A i-ik és a j-ik helyen lévő biteket cseréli ki.

Algoritmusok

Ebben a részben a szabványos algoritmusokat mutatjuk be.
Minden algoritmus egy függvénysablon. A leírást egyszerűsítendő a generic specifikációkat nem közöljük. Ehelyett mindvégig az alábbi generic típusneveket fogjuk használni:

Generic név	*Aminek megfelel*
BiIter	Kétirányú iterátor
ForIter	Előre irányú iterátor
InIter	Bemeneti iterátor
OutIter	Kimeneti iterátor
T	Valamilyen adattípus
Size	Valamilyen egésztípus
Func	Valamilyen függvénytípus
Generator	Objektumokat előállító függvény
BinPred	Bináris predikátum
UnPred	Unáris predikátum
Comp	Összehasonlító függvény, amely az arg1 <arg2 eredményét adja vissza

adjacent_find

```
ForIter adjacent_find(ForIter start, ForIter end);
ForIter adjacent_find(ForIter start, ForIter end,
                      BinPred pfn);
```

Az **adjacent_find( )** algoritmus egyező szomszédos elemeket keres a *start* és az *end* iterátorok által meghatározott szekvenciában és a talált pár első elemére mutató iterátort ad vissza. Ha nincs ilyen szomszédos pár, akkor az *end* iterátort kapjuk vissza. Az első verzió ugyanolyan elempárokat keres, míg a másodikban mi adhatjuk meg a *pfn* függvényen keresztül az egyezés kritériumait.

binary_search

```
bool binary_search(ForIter start, ForIter end,
                   const T &val);
bool binary_search(ForIter start, ForIter end,
                   const T &val, Comp cmpfn);
```

A **binary_search( )** algoritmus logaritmikus keresést végez a *start* és az *end* iterátorok által közrefogott rendezett sorozaton, a keresett értéket a *val* paraméterben adjuk meg. Visszatérési értéke true, ha a sorozatban van ilyen érték, találat híján false értéket ad vissza. Az első változatban a találat megegyezést jelent. A második változat engedi saját összehasonlító függvény használatát.

copy

```
OutIter copy(InIter start, InIter end, OutIter result);
```

A **copy( )** algoritmus a *start* és az *end* iterátorok által közrefogott sorozatot másolja a *result* iterátor által mutatott helyre. Az átmásolandó sorozat nem és az eredmény nem fedhetik át egymást.

copy_backward

```
BiIter2 copy_backward(BiIter1 start, BiIter1 end,
                      BiIter2 result);
```

A **copy_backward( )** algoritmus csak abban különbözik a **copy( )**-tól, hogy az elemek először a sorozat végéről kerülnek át az eredménysorozatba.

count

```
size_t count(InIter start, InIter end, const T &val);
```

A **count( )** algoritmus a start és end operátorok által közrefogott sorozat azon elemeinek számát adja vissza, amelyek megegyeznek *val*-al.

count_if

```
size_t count_if(InIter start, InIter end, UnPred pfn);
```

A **count_if( )** algoritmus a *start* és az *end* iterátorok által közrefogott sorozat azon elemeinek a számát adja vissza, amelyekre a *pfn* unáris predikátum igaz.

equal

```
bool equal(InIter1 start1, InIter1 end1, InIter2 start2);
```

Az **equal( )** algoritmus eldönti, hogy két szekvencia megegyezik-e. A *start1* és *end1* iterátorok által közrefogott sorozat kerül összehasonlításra a *start2*-ben kezdődő sorozattal. Egyezés esetén true értéket kapunk vissza, különben false-t.

equal_range

```
pair<ForIter, ForIter> equal_range(ForIter start,
                                   ForIter end,
                                   const T &val);
pair<ForIter, ForIter> equal_range(ForIter start,
                                   ForIter end,
                                   const T &val,
                                   Comp cmpfn);
```

Az **equal_range( )** algoritmus egy olyan részsorozatot reprezentáló iterátor párt ad vissza (nevezetesen a részsorozat elejére és végére mutató iterátorokat), amelybe a megadott elem beszúrható úgy, hogy közben ne változzon meg a *start, end* iterátorok által közrefogott sorozat

rendezettsége. Az elemet a *val* tartalmazza. A második változatban az összehasonlító függvényt magunk adhatjuk meg a *cmpfn* paraméteren keresztül.

A **pair** osztálysablon a <utility> könyvtárból származik, segítségével objektum párokat kapcsolhatunk össze, amelyeket külön-külön a **first**, illetve a **second** mezőkön keresztül érhetünk el.

fill, fill_n

```
void fill(ForIter start, ForIter end, const T &val);
void fill_n(ForIter start, Size num, const T &val);
```

A **fill( )** és a **fill_n( )** algoritmusok egy megadott sorozatot töltenek fel a *val* paraméterben megadott értékkel. A **fill( )** esetében a sorozatot a *start* és az *end* iterátorokkal adjuk meg, míg a **fill_n( )**-ben a *start* mellett a sorozat hosszát (azaz, hogy hány elemből áll) kell megadnunk a *num* paraméterben.

find

```
InIter find(InIter start, InIter end, const T &val);
```

A **find( )** algoritmus a *val* paraméterként megadott értéket keresi meg *start* és az *end* iterátorok által közrefogott sorozatban. Az első találatra mutató iterátorral tér vissza. Ha a *val* nem eleme a sorozatnak, akkor az *end* iterátort kapjuk vissza.

find_end

```
FwdIter1 find_end(ForIter1 start1, ForIter1 end1,
                  ForIter2 start2, ForIter2 end2);
FwdIter1 find_end(ForIter1 start1, ForIter1 end1,
                  ForIter2 start2, ForIter2 end2,
                  BinPred pfn);
```

A **find_end( )** algoritmus a *start1* és *end1* iterátorok által meghatározott sorozatban keresi a *start2* és *end2* iterátorok által közrefogott részsoroza-

tot, ha talált ilyet, akkor a talált részsorozat utolsó elemére mutató iterátort ad vissza, különben az *end1*-et kapjuk vissza.

A második változatban az „egyezést" is definiálhatjuk a *pfn* függvényen keresztül.

find_first_of

```
FwdIter1 find_first_of(ForIter1 start1, ForIter1 end1,
                       ForIter2 start2, ForIter2 end2);
FwdIter1 find_first_of(ForIter1 start1, ForIter1 end1,
                       ForIter2 start2, ForIter2 end2,
                       BinPred pfn);
```

A **find_first_of( )** algoritmus a *start1* és *end1* iterátorok által meghatározott sorozatban megkeresi az első olyan elemet – és az arra mutató iterátort adja vissza –, amelyhez van a *start2* és *end2* iterátorok által meghatározott sorozatban vele megegyező elem. Ha ilyen nincs, akkor az *end1* iterátort kapjuk vissza.

A második változatban definiálhatjuk az „egyezést" a *pfn* bináris predikátumon keresztül.

find_if

```
InIter find_if(InIter start, InIter end, UnPred pfn);
```

A **find_if( )** algoritmus megkeresi az első olyan a *start* és az *end* iterátorok által közrefogott részsorozatban, amelyre a *pfn* unáris predikátum igaz és a találatra mutató iterátorral tér vissza. Ha nincs találat az *end*-et kapjuk vissza.

for_each

```
Func for_each(InIter start, InIter end, Func fn);
```

A **for_each( )** algoritmus a *start* és *end* iterátorok közti sorozat minden elemére alkalmazza az *fn* függvényt. Az *fn* függvényt adja vissza.

generate, generate_n

```
void generate(ForIter start, ForIter end, Generator
              fngen);
void generate_n(OutIter start, Size num, Generator
                fngen);
```

A **generate( )** és a **generate_n( )** algoritmusok egy sorozat elemeinek adnak egy generátor függvény által értéket. A **generate( )** esetében a sorozatot a két végére mutató iterátorral (*start* és *end*) adjuk meg. A **generaten( )** esetében a sorozatot a az első elemére mutató itererátor és elemeinek száma adja meg. A generátor függvény az *fngen* paraméteren keresztül adjuk át. A függvénynek nincsenek paraméterei.

includes

```
bool includes(InIter1 start1, InIter1 end1, InIter2 start2,
              InIter2 end2);
bool includes(InIter1 start1, InIter1 end1, InIter2 start2,
              InIter2 end2, Comp cmpfn);
```

Az **includes( )** algoritmus eldönti, hogy a *start1, end1* iterátorok által közrefogott sorozat tartalmazza-e a *start2* és *end2* iterátorok által meghatározott sorozat minden tagját. Ha minden elem megtalálható a sorozatban, akkor true értéket ad vissza, egyébként false-t.

A második változatban az „egyezést" magunk definiálhatjuk a *cmpfn* összehasonlító függvényen keresztül.

inplace_merge

```
void inplace_merge(BiIter start, BiIter mid, BiIter end);
void inplace_merge(BiIter start, BiIter mid, BiIter end,
                   Comp cmpfn);
```

Az **inplace_merge( )** algoritmus egy sorozat két részét fésüli össze. A két részt a *start* és a *mid* valamint a *mid* és az *end* iterátorok által meghatározott részsorozatok jelentik. Mindkét résznek rendezettnek kell

lennie (növekvő sorrendben). Az eredmény sorozat szintén rendezett lesz.

A második változatban magunk definiálhatunk egy összehasonlító függvényt, amely eldönti, hogy két elem közül melyik a kisebb.

iter_swap

```
void iter_swap(ForIter1 i, ForIter2 j);
```

Az **iter_swap( )** algoritmus az argumentumaként megadott két iterátor által mutatott értéket cseréli ki.

lexicographical_compare

```
bool lexicographical_compare(InIter1 start1, InIter1 end1,
                             InIter2 start2, InIter2 end2);
bool lexicographical_compare(InIter1 start1, InIter1 end1,
                             InIter2 start2, InIter2 end2,
                             Comp cmpfn);
```

A **lexicographical_compare( )** algoritmus lexikografikusan összehasonlítja a *start1*, *end1* és a *start2*, *end2* iterátorpárokkal megadott sorozatokat. True értéket ad vissza, ha az első sorozat lexikografikusan kisebb, mint a második (azaz ha az első sorozat egy szótárban előbb fordulna elő, mint a második).

A második változatban megadhatunk egy összehasonlító függvényt, amely eldönti, hogy két elem közül melyik a kisebb.

lower_bound

```
ForIter lower_bound(ForIter start, ForIter end,
                    const T &val);
ForIter lower_bound(ForIter start, ForIter end,
                    const T &val, Comp cmpfn);
```

A **lower_bound( )** algoritmus megkeresi a *start* és *end* iterátorok által közrefogott sorozatban az első olyan elemet, amely nem kisebb, mint *val* és a találatra mutató iterátort ad vissza.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

make_heap

```
void make_heap(RandIter start, RandIter end);
void make_heap(RandIter start, RandIter end, Comp
               cmpfn);
```

A **make_heap( )** algoritmus a *start* és az *end* iterátorok által közrefogott sorozatól készít heap-et.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

max

```
const T &max(const T &i, const T &j);
const T &max(const T &i, const T &j, Comp cmpfn);
```

A **max( )** algoritmus a megadott két érték maximumát adja vissza.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

max_element

```
ForIter max_element(ForIter start, ForIter last);
ForIter max_element(ForIter start, ForIter last,
                    Comp cmpfn);
```

A **max_element( )** algoritmus a *start* és a *last* iterátorok által közrefogott sorozat maximumát választja ki és ad vissza egy arra mutató iterátort.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

merge

```
OutIter merge(InIter1 start1, InIter1 end1, InIter2 start2,
              InIter2 end2, OutIter result);
OutIter merge(InIter1 start1, InIter1 end1, InIter2 start2,
              InIter2 end2, OutIter result, Comp cmpfn);
```

A **merge( )** algoritmus két rendezett sorozatot fésül össze az eredményt, egy harmadik sorozatba helyezve. Az összefésülendő sorozatokat a *start1*, *end1* és a *start2*, *end2* iterátor párok jelölik ki. Az eredmény helyét a *result* iterátorral adjuk meg. A **merge** az eredménysorozat végére mutató iterátort ad vissza.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

min

```
const T &min(const T &i, const T &j);
const T &min(const T &i, const T &j, Comp cmpfn);
```

A **max( )** algoritmus a megadott két érték minimumát adja vissza.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

min_element

```
ForIter min_element(ForIter start, ForIter last);
ForIter min_element(ForIter start, ForIter last,
                    Comp cmpfn);
```

A **min_element( )** algoritmus a *start* és a *last* iterátorok által közrefogott sorozat minimumát választja ki és ad vissza egy arra mutató iterátort.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

mismatch

```
pair<InIter1, InIter2> mismatch(InIter1 start1,
                                InIter1 end1,
                                InIter2 start2);

pair<InIter1, InIter2> mismatch(InIter1 start1,
                                InIter1 end1,
                                InIter2 start2,
                                BinPred pfn);
```

A **mismatch( )** algoritmus két sorozat első eltérő elemét keresi meg. Az eltérő két elemre kapunk vissza egy iterátorpárt. Ha a két sorozat megegyezik, akkor az utolsó elemekre mutató iterátorokat kapunk vissza.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem egyenlő-e.

A **pair** osztálysablon a <utility> könyvtárból származik, segítségével objektumpárokat kapcsolhatunk össze, amelyeket külön-külön a **first** ill. a **second** mezőkön keresztül érhetünk el.

next_permutation

```
bool next_permutation(BiIter start, BiIter end);
bool next_permutation(BiIter start, BiIter end, Comp
                      cmpfn);
```

A **next_permutation( )** algoritmus egy sorozat következő permutációját állítja elő. A sorozat első permutációját a rendezett sor adja. Egy permutáció rákövetkezőjét lexikografikusan kell érteni. Ha a megadott sorozatra a következő permutáció nem létezik, akkor a rendezett sorrendet kapjuk, és a függvény false értékkel tér vissza, különben true-t ad vissza.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

nth_element

```
void nth_element(RandIter start, RandIter element,
                 RandIter end);
void nth_element(RandIter start, RandIter element,
                 RandIter end, Comp cmpfn);
```

Az **nth_element( )** algoritmus úgy rendezi el az elemeket *start* és *end* iterátorok által közrefogott sorozatban, hogy minden az *element*-nél kisebb elem az *element* előtt szerepeljen és minden nála nagyobb pedig utána.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a nagyobb.

partial_sort

```
void partial_sort(RandIter start, RandIter mid
                  RandIter end);
void partial_sort(RandIter start, RandIter mid
                  RandIter end, Comp cmpfn);
```

A **partial_sort( )** algoritmus úgy mozgatja a *start* és az *end* iterátorok által közrefogott elemeket, hogy az eredmény a *mid* iterátor által mutatott elemig rendezett legyen.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

partial_sort_copy

```
RandIter partial_sort_copy(InIter start, InIter end,
                           RandIter res_start,
                           RandIter res_end);

RandIter partial_sort_copy(InIter start, InIter end,
                           RandIter res_start,
                           RandIter res_end,
                           Comp cmpfn);
```

A **partial_sort_copy( )** algoritmus a *start* és *end* iterátorok által közrefogott sorozatot rendezi, majd ebből átmásol annyi elemet a *res_start* és *res_end* által meghatározott sorozatba, amennyi elfér. Az utolsó átmásolt elemre mutató iterátort adja vissza.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

partition

```
BiIter partition(BiIter start, BiIter end,
                 UnPred pfn);
```

A **partition( )** algoritmus algoritmus a *start* és *end* iterátorok által közrefogott sorozatot úgy alakítja, hogy azok az elemek, amelyre a *pfn* unáris predikátum igaz megelőzzék azokat, amelyekre a *pfn* hamis. A **partition( )** az első olyan elemre mutató iterátort ad vissza, amelyre a predikátum hamis.

pop_heap

```
void pop_heap(RandIter start, RandIter end);
void pop_heap(RandIter start, RandIter end, Comp
              cmpfn);
```

A **pop_heap( )** algoritmus kicseréli a heap tulajdonságú sorozat első és utolsó előtti elemét, majd újra heappé alakítja.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

prev_permutation

```
bool prev_permutation(BiIter start, BiIter end);
bool prev_permutation(BiIter start, BiIter end,
                      Comp cmpfn);
```

A **prev_permutation( )** algoritmus a megadott sorozat előző permutációját. A sorozat első permutációját a rendezett sorrend adja. Egy per-

mutáció megelőzőjét lexikografikusan kell érteni. Ha a megadott sorozatra a következő permutáció nem létezik, akkor a rendezett sorrendet kapjuk, és a függvény false értékkel tér vissza, különben true-t ad vissza.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

push_heap

```
void push_heap(RandIter start, RandIter end);
void push_heap(RandIter start, RandIter end, Comp
               cmpfn);
```

A **push_heap( )** algoritmus a heap végére helyez egy elemet. A *start* és az *end* által megadott sorozat heap tulajdonságú kell hogy legyen.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

random_shuffle

```
void random_shuffle(RandIter start, RandIter end);
void random_shuffle(RandIter start, RandIter end,
                    Generator rand_gen);
```

A **random_shuffle( )** algoritmus a *start* és az *end* iterátorok által közrefogott sorozat elemeit keveri össze véletlenszerűen.

A második változatban megadhatjuk a keveréskor használt véletlenszám-generáló függvényt. Ennek az alábbi formájúnak kell lennie:

rand_gen(*num*);

Visszatérési értéke egy 0 és *num* közti szám kell hogy legyen.

remove, remove_if, remove_copy, remove_copy_if

```
ForIter remove(ForIter start, ForIter end, const T &val);
ForIter remove_if(ForIter start, ForIter end, UnPred pfn);
OutIter remove_copy(InIter start, InIter end,
                    OutIter result, const T &val);
```

```
OutIter remove_copy_if(InIter start, InIter end,
                       OutIter result, UnPred pfn);
```

A **remove( )** algoritmus a megadott sorozatból törli a *val* értékű elemeket és a megmaradó elemek végére mutató iterátort ad vissza.

A **remove_if( )** algoritmus törli az összes olyan elemet a megadott sorozatból, amelyre a *pfn* predikátum igaz és a megmaradó elemek végére mutató iterátort ad vissza.

A **remove_copy( )** algoritmus a megadott sorozat azon elemeit törli, amelyek értéke megegyezik *val* a paraméter értékével, az eredmény sorozatot a *result* iterátor által mutatott pozícióval kezdve helyezi el. Az eredménysorozat végére mutató iterátort ad vissza.

A **remove_copy_if ( )** algoritmus a megadott sor azon elemeit másolja a *result* iterátor által mutatott pozíciótól kezdve, amelyre a *pfn* predikátum igaz. Az eredménysorozat végére mutató iterátort ad vissza.

replace, replace_copy, replace_if, replace_copy_if

```
void replace(ForIter start, ForIter end, const T &old,
             Const T &new);
void replace_if(ForIter start, ForIter end, UnPred pfn,
                Const T &new);
OutIter replace_copy(InIter start, InIter end,
                     OutIter result, const T &old,
                     Const T& new);
OutIter replace_copy_if(InIter start, InIter end,
                        OutIter result, UnPred pfn,
                        Const T &new);
```

A megadott sorozatban a **replace( )** algoritmus az összes *old* értékkel rendelkező elemet helyettesíti *new* értékűvel.

A megadott sorozatban a **replace_if( )** algoritmus az összes olyan elemet helyettesíti *new* értékűvel, amelyre a *pfn* predikátum igaz.

A megadott sorozatban a **replace_copy( )** algoritmus az összes *old* értékkel rendelkező elemet helyettesíti *new* értékűvel, az eredmény a *result* iterátor által mutatott helyre kerül. Az eredeti sorozat változatlan marad. Az eredménysorozat végére mutató iterátort kapunk vissza.

A **replace_copy_if( )** algoritmus a megadott sorozat azon elemeit másolja – miközben értéküket *new*-ra változtatja – a *result* iterátor által kijelölt helytől kezdődően, amelyekre a *pfn* unáris predikátum igaz. Az eredeti sorozat változatlan marad. Az eredménysorozat végére mutató iterátort kapunk vissza.

reverse, reverse_copy

```
void reverse(BiIter start, BiIter end);
OutIter reverse_copy(BiIter first, BiIter last,
                     OutIter result);
```

A **reverse( )** algoritmus a *start* és az *end* iterátorok által közrefogott sorozatot fordítja meg.

A **reverse_copy( )** algoritmus ugyanazt csinálja, mint a **reverse( )** azzal a különbséggel, hogy az eredményt a *result* iterátor által mutatott pozícióra helyezi, miközben az eredeti sorozat változatlan marad. A **reverse_copy( )** az eredménysorozat végére mutató iterátort ad vissza.

rotate, rotate_copy

```
void rotate(ForIter start, ForIter mid, ForIter end);
OutIter rotate_copy(ForIter start, ForIter mid,
                    ForIter end, OutIter result);
```

A **rotate( )** algoritmus balra forgatja a *start* és az *end* iterátorok által közrefogott sorozatot úgy, hogy a *mid* iterátor által mutatott elem lesz az új első elem.

A **rotate_copy( )** algoritmus ugyanazt csinálja, mint a **rotate( )** azzal a különbséggel, hogy az eredményt a *result* iterátor által mutatott pozícióra helyezi, miközben az eredeti sorozat változatlan marad. A **rotate_copy( )** az eredmény végére mutató iterátort ad vissza.

search

```
ForIter1 search(ForIter1 start1, ForIter1 end1,
                ForIter2 start2, ForIter2 end2);
```

```
ForIter1 search(ForIter1 start1, ForIter1 end1,
                ForIter2 start2, ForIter2 end2,
                BinPred pfn);
```

A **search( )** algoritmus egy részsorozatot keres a megadott *start1*, *end1* iterátorok által közrefogott sorozaton belül. A keresendő részsorozat elejét és végét rendre a *start2* és az *end2* iterátorok jelölik. Ha van ilyen részsorozat, akkor a **search( )** annak az elejére mutató iterátort ad vissza.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem egyenlő-e.

search_n

```
ForIter search_n(ForIter start, ForIter end, Size num,
                 Const T &val);
ForIter search_n(ForIter start, ForIter end, Size num,
                 Const T &val, BinPred pfn);
```

A **search_n( )** algoritmus egy sorozatban egy *num* hosszúságú egyforma elemekből álló részsorozatot keres. A sorozatot a *start* és *end* iterátorok jelölik ki. Ha van ilyen részsorozat, a **search_n( )** ennek elejére mutató iterátort ad vissza. Különben az *end* iterátort kapjuk vissza.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem egyenlő-e.

set_difference

```
OutIter set_difference(InIter1 start1, InIter1 end1,
                       InIter2 start2, InIter2 end2,
                       OutIter result);
OutIter set_difference(InIter1 start1, InIter1 end1,
                       InIter2 start2, InIter2 end2,
                       OutIter result, Comp cmpfn);
```

A **set_difference( )** algoritmus egy olyan sorozatot állít elő, amely a két megadott sorozat (amelyek a *start1*, *end1* és *start2*, *end2* iterátorok jelöl-

nek ki), mint két rendezett halmaz különbségének elemeit tartalmazza. Az eredmény rendezett lesz, és a *result* iterátor által mutatott helyre kerül. A **set_difference( )** az eredmény végére mutató iterátort adja vissza.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

set_intersection

```
OutIter set_intersection(InIter1 start1, InIter1 end1,
                         InIter2 start2, InIter2 end2,
                         OutIter result);
OutIter set_intersection(InIter1 start1, InIter1 end1,
                         InIter2 start2, InIter2 end2,
                         OutIter result, Comp cmpfn);
```

A **set_intersection( )** algoritmus egy olyan sorozatot állít elő, amely a két megadott sorozat (amelyek a *start1, end1* és *start2, end2* iterátorok jelölnek ki), mint két rendezett halmaz metszetének elemeit tartalmazza. Az eredmény rendezett lesz, és a *result* iterátor által mutatott helyre kerül. A **set_intersection ( )** az eredmény végére mutató iterátort adja vissza.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

set_symmetric_difference

```
OutIter set_symmetric_difference(InIter1 start1,
                                 InIter1 end1,
                                 InIter2 start2,
                                 InIter2 end2,
                                 OutIter result);
OutIter set_symmetric_difference(InIter1 start1,
                                 InIter1 end1,
                                 InIter2 start2,
                                 InIter2 end2,
                                 OutIter result,
                                 Comp cmpfn);
```

A **set_symmetric_difference( )** algoritmus egy olyan sorozatot állít elő, amely a két megadott sorozat (amelyek a *start1, end1* és *start2, end2* iterátorok jelölnek ki), mint két rendezett halmaz szimmetrikus differenciájának az elemeit tartalmazza. Az eredmény rendezett lesz, és a *result* iterátor által mutatott helyre kerül. A **set_symmetric_difference( )** az eredmény végére mutató iterátort adja vissza.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

set_union

```
OutIter set_union(InIter1 start1, InIter1 end1,
                  InIter2 start2, InIter2 end2,
                  OutIter result);
OutIter set_union(InIter1 start1, InIter1 end1,
                  InIter2 start2, InIter2 end2,
                  OutIter result, Comp cmpfn);
```

A **set_union( )** algoritmus egy olyan sorozatot állít elő, amely a két megadott sorozat (amelyek a *start1, end1* és *start2, end2* iterátorok jelölnek ki), mint két rendezett halmaz unionjának elemeit tartalmazza. Az eredmény rendezett lesz, és a *result* iterátor által mutatott helyre kerül. A **set_union( )** az eredmény végére mutató iterátort adja vissza.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

sort

```
void sort(RandIter start, RandIter end);
void sort(RandIter start, RandIter end, Comp cmpfn);
```

A **sort( )** algoritmus a *start* és az *end* iterátorok által közrefogott sorozatot rendezi.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

sort_heap

```
void sort_heap(RandIter start, RandIter end);
void sort_heap(RandIter start, RandIter end, comp
               cmpfn);
```

A **sort_heap( )** algoritmus a *start* és az *end* iterátorok által megadott heapet rendezi.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

stable_partition

```
BiIter_partition(BiIter start, BiIter end);
                 BinPred pfn);
```

A **stable_partition( )** algoritmus a *start* és *end* iterátorok által közrefogott sorozatot úgy alakítja, hogy azok az elemek, amelyre a *pfn* unáris predikátum igaz megelőzzék azokat, amelyekre a *pfn* hamis. A partícionálás stabil, ami azt jelenti, hogy a két részben az elemek relatív sorrendje nem változik. A **stable_partition( )** az első olyan elemre mutató iterátort ad vissza, amelyre a predikátum hamis.

stable_sort

```
void stable_sort(RandIter start, RandIter end);
void stable_sort(RandIter start, RandIter end, Comp
                 cmpfn);
```

A **stable_sort( )** algoritmus a *start* és az *end* iterátorok által közrefogott sorozatot rendezi. A rendezés stabil, ami azt jelenti, hogy egyenlő elemek nem cserélődnek fel.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

swap

```
void swap(T &i, T &j);
```

A **swap( )** algoritmus az *i* és a *j* által mutatott elemeket cseréli ki.

swap_ranges

```
ForIter2 swap_ranges(ForIter1 start1, ForIter1 end1,
                     ForIter2 start2);
```

A **swap_ranges( )** algoritmus a *start1* és az *end1* iterátorok által közrefogott sorozat elemeit cseréli fel a *start2*-ben kezdődő sorozat elemeivel. A *start2*-ben kezdődő sorozat végére mutató iterátort ad vissza.

transform

```
OutIter transform(InIter start, InIter end,
                  OutIter result, Func unaryfunc);
OutIter transform(InIter1 start1, InIter1 end1,
                  InIter2 start2, OutIter result,
                  Func binaryfunc);
```

A **transform( )** algoritmus egy függvényt alkalmaz a megadott sorozat minden tagjára és az így nyert eredménysorozatot a *result* iterátor által mutatott helytől kezdődően helyezi el. Az első esetben a kiindulási sorozatot a *start end* iterátorpár határozza meg. Az elemeken ható függvény az *unaryfunc*. Ez a függvény az elem értékét kapja meg és egy ebből származtatott értéket ad vissza.

A második változatban a transzformátor függvény kétváltozós, első argumentumaként az első sorozat elemei szolgálnak, míg második argumentumát a második sorozat elemeiből veszi. Az elemek sorban kerülnek feldolgozásra (először mindkét sorozat első eleme, aztán a második elemek, harmadik elemek stb.).

Mindkét változat az eredménysorozat végére mutató iterátort ad vissza.

Programozási tipp

*A **transform( )** egyike az érdekesebb algoritmusoknak, mert az adott sorozat minden elemét az általunk megadott függvény szerint módosítja. Az alábbi példaprogram egy **xform( )** nevű egyszerű transzformációs függvényt definiál, amely a lista tartalmát emeli négyzetre.*

```
// Példa a transform algoritmus használatára
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

// Egy egyszerű transzformációs függvény
int xform(int i){
  return i * i; // Eredeti érték négyzetre emelése
}

int main()
{
  list<int> x1;
  int i;

  // berakja az értékeket a listába
    for(i=0; i<10; i++) x1.push_back(i);

  cout << "Az x1 eredeti tartalma: ";
  list<int>::iterator p = x1.begin();
  while(p !=x1.end()) {
    cout <<*p << " ";
    p++;
  }

  cout << endl;

  //x1 transzformálása
  p = transform(x1.begin(), x1.end(),
                x1.begin(), xform);
```

```
    cout << "Az x1 transzformált tartalma: ";
    p = x1.begin();
    while(p !=x1.end()) {
      cout << *p << " ";
      p++;
    }

    return 0;
}
```

A program által produkált kimenet:

```
Az x1 eredeti tartalma: 0 1 2 3 4 5 6 7 8 9
Az x1 transzformált tartalma:0 1 4 9 16 25 36
                             49 64 81
```

Látható, hogy x1 minden elemét negyzetre emelte a program.

unique, unique_copy

```
ForIter unique(ForIter start, ForIter end);
ForIter unique(ForIter start, ForIter end, BinPred pfn);
OutIter unique_copy(ForIter start, ForIter end,
                    OutIter result);
OutIter unique_copy(ForIter start, ForIter end,
                    OutIter result, BinPred pfn);
```

A **unique( )** a megadott sorozatban kiküszöböli a többszörös elem előfordulásokat. A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem egyenlő-e. A **unique( )** az új sorozat végére mutató iterátort ad vissza.

A **unique_copy( )** algoritmus a *start1* és az *end1* iterátorok által közrefogott sorozatban küszöböli ki a többszörös elem előfordulásokat. Az eredmény a *result* iterátor által mutatott hely mögé kerül. Az eredeti sorozat változatlan marad. A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem egyenlő-e. A **unique_copy( )** az új sorozat végére mutató iterátort ad vissza.

upper_bound

```
ForIter upper_bound(ForIter start, ForIter end,
                    const T &val);
ForIter upper_bound(ForIter start, ForIter end,
                    const T &val, Comp cmpfn);
```

Az **upper_bound( )** algoritmus az utolsó olyan pozíciót keresi meg a *start* és az *end* közötti intervallumban, amelyen lévő elem értéke nem nagyobb, mint *val* és erre az elemre mutató iterátort ad vissza.

A második változat lehetővé teszi egy saját összehasonlító függvény megadását, amely eldönti, hogy két elem közül melyik a kisebb.

15. FEJEZET

C++ karakterláncok és kivételek

Az STL, az iostream könyvtárakon kívül, egyes programozási szituációk kezelésének a megkönnyítésére, a C++ szabványos könyvtára további más predefinit osztályokat is támogat. Ezek közé tartoznak pl. olyan osztályok, amelyek numerikus módszereket, komplex aritmetikát támogatnak. Amíg sok osztályra ezek közül csak ritkán van szüksége a programozónak, a string- és a kivételekezelést megvalósító két osztályról ez nem mondható el. Ezek ismertetése következik most.

■ *Karakterláncok*

A C++ karakterláncok használatát kétféleképpen támogatja: nulla végű karaktertömbökön keresztül, amelyekre a szakirodalom gyakran C-stringként hivatkozik, illetve a **basic_string** osztály által. A **basic_string** osztálynak két specializációja van a **string**, amely az ASCII karakterstringeket, és a **wstring**, amely a széles karakterekből álló stringeket kezeli. Ebben a fejezetrészben a **string** osztályt tárgyaljuk.

A **basic_string** osztály a **char_traits** leszármazottja, amely a stringet alkotó karakterek számos tulajdonságát definiálja. Nem árt tudnunk, hogy míg a megszokott stringeket vagy **char** vagy **wchar_t** típusú objektumok alkotják, addig a **basic_string** bármilyen objektummal működik, amely használható szöveges karakter reprezentálására.

A **basic_string** osztály alapvetően nem más, mint egy konténer. Ez azt jelenti, hogy támogatja az STL-ről szóló részben ismertetett algoritmusokat. Ezen felül a stringek további szolgáltatásokat is nyújtanak. A **string** osztály használatához, szükségünk van az #include <string> sorra.

A **basic_string** generic osztály specifikációja a következő:

```
template<class charT, class Traits = char_traits<charT>,
        class Allocator = allocator<T>>class basic_string
```

ahol **charT** a használni kívánt karaktertípus. **Traits** a karaktereket leíró osztály és az **Allocator** az allokátort adja meg. A **basic_string** az alábbi konstruktorokat tartalmazza:

```
explicit basic_string(const Allocator &a = Allocator( ));

basic_string(size_type len, charT ch,
             const Allocator &a = Allocator( ));

basic_string(const charT *str; const Allocator &a = Allocator( ));

basic_string(const charT *str; size_type len,
             const Allocator &a = Allocator( ));

basic_string(const basic_string &str, size_type indx = 0,
             size_type len, const Allocator &a = Allocator( ));

template<class InIter> basic_string(InIter start,
        InIter end, const Allocator &a = Allocator( ));
```

Az első konstruktor egy üres stringet hoz létre. A második változat egy olyan *len* hosszúságú stringet állít elő, amely csupa *ch* karakterből áll. A harmadik konstruktor olyan stringet készít, amely ugyanazokat az elemeket tartalmazza, mint a *str* string. A negyedik verzió a *str* string *len* hosszúságú kezdőszeletét másolja le. Az ötödik egy másolatot készít egy másik **basic_string** típusú string *indx*-ben kezdődő *len* karakter hosszú szeletéről. A hatodik konstruktor változat olyan stringet hoz létre, amely a *start* és az *end* iterátorok által közrefogott string elemeit tartalmazza.

Az alábbi összehasonlító műveletek definiáltak a **basic_string** osztályra:

```
==, <, <=, !=, > , >=.
```

A + is az osztályműveletek közé tartozik, segítségével két stringet konkatenálhatunk össze. A >> és a << I/O operátorok input és output stringek esetén használhatók. A + operátor egyaránt használható két karakterlánc-objektum, valamint egy karakterlánc-objektum és egy C-stílusú string egymásba fűzésére. Az alábbi művelet sémák lehetségesek:

string + string
string + C-string
C-string + string

A + operátorral lehetséges egyetlen karakter string végéhez toldása is.

A **basic_string** osztály definiálja az **npos** konstanst, amely általában –1. Ez a konstans reprezentálja a lehetséges legnagyobb string hosszát.

Az ismertetésben a generic **charT** típus a stringet alkotó karakterek típusát jelöli. Mivel a paraméterként megadott típusok nevei önkényesen megválaszthatók, az ebből származtatott, a tagfüggvények által használt típusok ezektől függnek. Azért, hogy mégis hivatkozni lehessen rájuk, a konténerosztályok deklarálnak néhány predefinit típusnevet a **typedef** segítségével. A **basic_string** által leggyakrabban használt ilyen **typedef** típusok a következők:

Típus	*Leírás*
size_type	A **size_t** -vel ekvivalens egész típus
reference	Referencia egy elemre
const_reference	Konstans referencia egy elemre
iterator	Iterátor
const_iterator	Konstans iterátor
reverse_iterator	Fordított irányú iterátor
const_reverse_iterator	Konstans fordított irányú iterátor
value_type	A konténerben tárolt típus
allocator_type	Az allokátor típusa
pointer	A string valamely elemére mutató pointer
const_pointer	A string valamely elemére mutató **const** pointer

A **basic_string** az alábbi függvénytagokat tartalmazza:

Tag	*Leírás*
basic_string &append(const basic_string &*str*);	A hívó string végéhez fűzi az *str* karakterláncot, ***this**-t ad vissza.
basic_string &append(const basic_string &*str*, size_type *indx*, size_type *len*);	Az *str* string *indx* elemétől kezdődő *len* hosszúságú szeletét fűzi a hívó stringhez, ***this**-t ad vissza.
basic_string &append(const charT **str*);	A hívó string végéhez fűzi az *str* karakterláncot, ***this**-t ad vissza.
basic_string &append(const char T **str*, size_type *num*);	A hívó string végéhez fűzi az *str* karakterlánc első *num* karaktere által alkotott szeletét, ***this**-t ad vissza.
basic_string &append(size_type *len*, char T *ch*);	A hívó string végéhez fűz *len* darab *ch* karaktert, ***this**-t ad vissza.
template<class InIter> basic_string &append(InIter *start*, InIter *end*);	A *start* és *end* iterátorok által meghatározott sorozatot fűzi a hívó string végéhez, ***this**-t ad vissza.
basic_string &assign(const basic_string &*str*);	A hívó stringnek *str* értéket ad, ***this**-t ad vissza.
basic_string &assign(const basic_string &*str*, size_type *indx*, size_type *len*);	A hívó stringnek a *str indx* helytől kezdődő *len* hosszúságú szeletét adja értékül, ***this**-t ad vissza.
basic_string &assign(const charT **str*);	A hívó stringnek *str* értéket ad, ***this**-t ad vissza.
basic_string &assign(const charT **str*, size_type *len*);	A hívó stringnek a *str len* hosszúságú kezdőszeletét adja értékül, ***this**-t ad vissza.
basic_string &assign(size_type *len*, charT *ch*);	A hívó stringnek *len* hosszúságú *ch* karakterből álló karakterláncot ad értékül, ***this**-t ad vissza.
template<class InIter> basic_string &append(InIter *start*, InIter *end*);	A hívó string a *start* és az *end* által meghatározott sorozatot kapja értékül, ***this**-t ad vissza.

Tag	*Leírás*
reference at(size_type *indx*); const_reference at(size_type *indx*) const;	Az *indx* pozíciójú karakterre ad vissza referenciát.
iterator begin( ); const_iterator begin( ) const;	A string első elemére mutató iterátort ad vissza.
const charT *c_str( ) const;	A hívó string C-stílusú (nullavégű) változatát készíti el, és az arra mutató pointert adja vissza.
size_type capacity( ) const;	A string aktuális kapacitását adja vissza. Ez az újabb memória allokálás nélkül a stringben tárolható karakterek maximális száma.
int compare (const basic_string &*str*) const;	A hívó stringet hasonlítja össze *str*-rel. Visszatérési értékei az alábbiak lehetnek: Negatív, ha ***this** < *str* 0, ha ***this** == *str* Pozitív, ha ***this** > *str*
int compare(size_type *indx*, size_type *len*, const basic_string &*str*) const;	A hívó string *indx* pozíciójától kezdődő *len* hosszúságú szeletét hasonlítja össze *str*-rel. Visszatérési értékei az alábbiak lehetnek: Negatív, ha ***this** < *str* 0, ha ***this** == *str* Pozitív, ha ***this** > *str*
int compare(size_type *indx*, size_type *len*, const basic_string &*str*, size_type *indx2*, size_type *len2*) const;	Az *str* egy szeletét hasonlítja össze a hívó string egy részstringjével. Az előbbit az *indx1* indextől kezdődő *len1* darab elem, az utóbbit az *indx2* indextől kezdődő *len2* darab elem határozza meg. Visszatérési értékei az alábbiak lehetnek: Negatív, ha ***this** < *str* 0, ha ***this** == *str* Pozitív, ha ***this** > *str*
int compare(const char T **str*) const;	A hívó stringet hasonlítja össze a *str* karakterlánccal. Visszatérési értékei alábbiak lehetnek: Negatív, ha ***this** < *str* 0, ha ***this** == *str* Pozitív, ha ***this** > *str*

Tag	*Leírás*
int compare(size_type *indx*, size_type *len*, const char T **str*, size_type *len2* = npos) const;	Az hívó string egy szeletét hasonlítja össze a *str* egy részstringjével. Az előbbit az *indx* indextől kezdődő *len* darab elem, az utóbbit a 0 indextől kezdődő *len2* darab elem határozza meg. Visszatérési értékei az alábbiak lehetnek: Negatív, ha ***this** < *str* 0, ha ***this** == *str* Pozitív, ha ***this** > *str*
size_type copy(charT **str*, size_type *len*, size_type *indx* = 0) const;	A hívó string *indx* indextől kezdődő *len* hosszúságú szeletét másolja a *str* stringbe. A bemásolt karakterek számát adj vissza.
const charT *data( ) const;	A hívó string első karakterére mutató pointert ad vissza.
bool empty( ) const;	Ha a hívó string üres true értéket ad vissza, különben false-t.
iterator end( ); const_iterator end( ) const;	A hívó string végére mutató iterátort ad vissza.
iterator erase(iterator *i*);	Az *i* iterátor által mutatott karaktert törli a stringből.
iterator erase(iterator *start*, iterator *end*);	A *start* és az *end* iterátorok által közrefogott részstringet törli a hívó stringből.
basic_string &erase(size_type *indx* = 0, size_type *len* = npos);	Az *indx* -től kezdve *len* karaktert töröl a stringből.
size_type find(const basic_string &*str*, size_type *indx* = 0) const;	Megkeresi a *str* string részstringet a hívó stringben. Az első előfordulás első karakterének indexét adja vissza. Ha nincs találat, **npos**-t kapunk vissza.
size_type find(const charT **str*, size_type *indx* = 0) const;	Megkeresi a *str* részstringet a hívó string *indx* indexétől indulva. Az első előfordulás első karakterének indexét adja vissza. Ha nincs találat, **npos**-t kapunk vissza.
size_type find(const charT **str*, size_type *indx*, size_type *len*) const;	Megkeresi a *str* string *len* hosszúságú kezdőszeletének előfordulását a hívó stringben. A keresés az *indx* indextől indul. Ha nincs találat, **npos**-t kapunk vissza.

Tag	*Leírás*
size_type find(charT *ch*, size_type *indx* = 0) const;	A *ch* karakter első előfordulását keresi meg a hívó karakterláncban. Ha van találat, akkor az indexét adja vissza, különben **npos**-t.
size_type find_first_of (const basic_string &*str*, size_type *indx* = 0) const;	Az első olyan karakter indexét adja vissza a hívó stringnek, amely megegyezik a *str* valamely karakterével. A keresés *indx* indextől kezdődik. Ha nincs találat, **npos**-t kapunk vissza.
size_type find_first_of(const charT **str*, size_type *indx* = 0) const;	A hívó string első olyan karakterének indexét adja vissza, amely megegyezik a *str* valamely karakterével. A keresés *indx* indextől kezdődik. Ha nincs találat, **npos**-t kapunk vissza.
size_type find_first_of(const charT **str*, size_type *indx*, size_type *len*) const;	A hívó string első olyan karakterének indexét adja vissza, amelyen a *str* string első *len* karakterének valamelyike szerepel. A keresés az *indx* indextől kezdődik. Ha nincs találat, **npos**-t kapunk vissza.
size_type find_first_of(charT *ch*, size_type *indx* = 0) const;	A *ch* karakter első előfordulását keresi meg a hívó stringben. A keresés az *indx* indextől kezdődik. Ha van találat, annak indexét adja vissza, különben **npos**-t.
size_type find_first_not_of(const basic_string &*str*, size_type *indx* = 0) const;	A hívó string első olyan karakterének az indexét adja vissza, amely nem szerepel az *str* stringben. A keresés az *indx* indextől kezdődik. Ha nincsen ilyen, akkor **npos**-t kapunk vissza.
size_type find_first_not_of(const charT **str*, size_type *indx* = 0) const;	A hívó string első olyan karakterének az indexét adja vissza, amely nem szerepel az *str* stringben. A keresés az *indx* indextől kezdődik. Ha nincsen ilyen, akkor **npos**-t kapunk vissza.
size_type find_first_not_of(const charT* *str*, size_type *indx*, size_type *len*) const;	A hívó string első olyan karakterének az indexét adja vissza, amely nem szerepel az *str* string *len* hosszúságú kezdőszeletében. A keresés az *indx* indextől kezdődik. Ha nincs találat, **npos**-t kapunk vissza.
size_type find_first_not_of(charT *ch*, size_type *indx* = 0) const;	A hívó string első nem *ch* karakterének indexét adja vissza. A keresés az *indx* indextől kezdődik. Ha nincs találat, **npos**-t kapunk vissza.

Tag	*Leírás*
size_type find_last_of(const basic_string &*str*, size_type *indx* = npos) const;	A hívó string utolsó olyan karakterének az indexét adja vissza, amely benne van a *str* stringben. A keresés az *indx* indextől kezdődik. Ha nincs találat, **npos**-t kapunk vissza.
size_type find_last_of(const charT **str*, size_type *indx* = npos) const;	A hívó string utolsó olyan karakterének az indexét adja vissza, amely benne van a *str* stringben. A keresés az *indx* indextől kezdődik. Ha nincs találat **npos**-t kapunk vissza.
size_type find_last_of(const charT **str*, size_type *indx*, size_type *len*) const;	A hívó string utolsó olyan karakterének az indexét adja vissza, amely benne van az *str* string *len* hosszú kezdőszeletében. A keresés az *indx* indextől kezdődik. Ha nincs találat, **npos**-t kapunk vissza.
size_type find_last_of(charT *ch*, size_type *indx* = npos) const;	A *ch* karakter utolsó előfordulását keresi meg a hívó stringben. Ha van ilyen, akkor annak indexét adja vissza, különben **npos**-t kapunk vissza.
size_type find_last_not_of(const basic_string &*str*, size_type *indx* = npos) const;	A hívó string utolsó olyan karakterének az indexét adja vissza, amely nem található meg az *str* stringben. A keresés az *indx*-ik indextől kezdődik. Ha nincs találat, **npos**-t kapunk vissza.
size_type find_last_not_of(const charT **str*, size_type *indx* = npos) const;	A hívó string utolsó olyan karakterének az indexét adja vissza, amely nem található meg az *str* stringben. A keresés az *indx*-ik indextől kezdődik. Ha nincs találat, **npos**-t kapunk vissza.
size_type find_last_not_of(const charT **str*, size_type *indx*, size_type *len*) const;	A hívó string utolsó olyan karakterének az indexét adja vissza, amely nem található meg az *str* string *len* hosszú kezdőszeletében. A keresés az *indx*-ik indextől kezdődik. Ha nincs találat, **npos**-t kapunk vissza.
size_type find_last_not_of(charT *ch*, size_type *indx* = npos) const;	A hívó string utolsó nem *ch* karakterének az indexét adja vissza. A keresés az *indx* indextől kezdődik. Ha nincs találat, **npos**-t kapunk vissza.
allocator_type get_allocator( ) const;	A string allokátorát adja vissza.

Tag	*Leírás*
iterator insert(iterator *i*, const charT &*ch* = charT( ));	Az *i* iterátor által mutatott karakter elé beszúrja a *ch* karaktert. A karakterre mutató iterátort ad vissza.
basic_string &insert(size_type *indx*, const basic_string &*str*);	A hívó string *indx* indexére szúrja be a *str* stringet, ***this**-t ad vissza.
basic_string &insert(size_type *indx1*, const basic_string &*str*, size_type *indx2*, size_type *len*);	A *str* string *indx2* indexétől kezdődő *len* hosszúságú szeletét beszúrja a hívó string *indx1* indexétől kezdve, ***this**-t ad vissza.
basic_string &insert(size_type *indx*, const charT **str*);	A hívó string *indx* indexére szúrja be a *str* stringet, ***this**-t ad vissza.
basic_string &insert(size_type *indx*, const charT **str*, size_type *len*);	A hívó string *indx* indexére szúrja be az *str* string első *len* karakterét, ***this**-t ad vissza.
basic_string &insert(size_type *indx*, size_type *len*, charT *ch*);	A hívó string *indx* indexére *len* darab *ch* karaktert szúr be, ***this**-t ad vissza.
void insert(iterator *i*, size_type *len*, const charT &*ch*);	Az *i* iterátor által mutatott elem elé a *ch* karakter *len* példányát szúrja be.
template<class InIter> void insert(iterator *i*, InIter *start*, InIter *end*);	A *start* és az *end* iterátorok által közrefogott karaktersorozatot szúrja be az *i* által meghatározott elem elé.
size_type length( ) const;	A stringben tárolt karakterek aktuális számát adja vissza.
size_type max_size( ) const;	A stringben tárolható elemek maximális számát adja vissza.
reference operator [ ](size_type *indx*) const; const_reference operator[ ] (size_type *indx*) const;	Az *indx* indexű karakterre ad vissza referenciát.

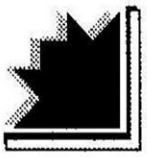
basic_string &operator=(const basic_string &*str*);
basic_string &operator=(const charT **str*);
basic_string &operator=(charT *ch*);
	A hívó stringet a megadott stringgel vagy karakterrel (pontosabban a karakter string változatával) teszi egyenlővé, ***this**-t ad vissza.

Tag	*Leírás*

basic_string &operator+=(const basic_string &*str*);
basic_string &operator+=(const charT **str*);
basic_string &operator+=(charT *ch*);
	A megadott stringet vagy karaktert a hívó string végéhez fűzi, ***this**-t ad vissza.

reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	A string végére mutató fordított irányú iterátort ad vissza.
reverse_iterator rend( ); const_reverse_iterator rend( ) const;	A string elejére mutató fordított irányú iterátort ad vissza.
basic_string &replace(size_type *indx*, size_type *len*, const basic_string &*str*);	A hívó string *indx* indexétől számított *len* karakterét helyettesíti a *str* string megfelelő hosszú kezdőszeletével, ***this**-t ad vissza.
basic_string &replace(size_type *indx1*, size_type *len1*, const basic_string &*str*, size_type *indx2*, size_type *len2*);	A *str* string *indx2* indexétől kezdődő *len2* hosszú szeletével helyettesíti a hívó string *indx1* indextől kezdődő részstringjét, úgyhogy max. *len1* helyettesítés történik, ***this**-t ad vissza.
basic_string &replace(size_type *indx*, size_type *len*, const charT **str*);	A hívó string *indx* indexétől számított *len* karakterét helyettesíti a *str* string megfelelő hosszú kezdőszeletével, ***this**-t ad vissza.
basic_string &replace(size_type *indx1*, size_type *len1*, const charT **str*, size_type *len2*);	A *str* string *indx2* indexétől kezdődő *len2* hosszú szeletével helyettesíti a hívó string *indx1* indextől kezdődő részstringjét, úgyhogy max. *len1* helyettesítés történik, ***this**-t ad vissza.
basic_string &replace(size_type *indx*, size_type *len1*, size_type *len2*, charT *ch*);	A hívó string *indx* indexében kezdődő max *len* hosszú részstringjét helyettesíti *len2* hosszú, csupa *ch* karakterből álló stringgel, ***this**-t ad vissza.
basic_string &replace(iterator *start*, iterator *end*, const basic_string &*str*);	A *start* és *end* iterátorok által közrefogott sorozatot helyettesíti a *str* stringgel, ***this**-t ad vissza.
basic_string &replace(iterator *start*, iterator *end*, const charT **str*);	A *start* és az *end* iterátorok által közrefogott sorozatot helyettesíti a *str* stringgel, ***this**-t ad vissza.
basic_string &replace(iterator *start*, iterator *end*, const charT **str*, size_type *len*);	A *start* és az *end* iterátorok által közrefogott sorozatot helyettesíti az *str* string első *len* karakterével, ***this**-t ad vissza.

Tag	*Leírás*
basic_string &replace(iterator *start*, iterator *end*, size_type *len*, charT *ch*);	A *start* és az *end* iterátorok által közrefogott sorozatot helyettesíti *len* darab *ch* karakterrel, ***this**-t ad vissza.
template<class InIter> basic_string &replace(iterator *start1*, iterator *end1*, InIter *start2*, InIter *end2*);	A *start1* és az *end1* iterátorok által közrefogott karaktersorozatot helyettesíti a *start2* és az *end2* iterátorok által meghatározott sorozattal, ***this**-t ad vissza.
void reserve(size_type num = 0);	A string kapacitását minimum *num*-ra állítja be.
void resize(size_type *num*); void resize(size_type *num*, charT *ch*);	A string méretét változtatja a *num*-ban megadottra. Ha a string nő, akkor a végére került új helyeket a *ch* karakterrel tölti fel.
size_type rfind(const basic_string &*str*, size_type *indx* = npos) const;	A *str* string utolsó előfordulását keresi meg a hívó stringben. A keresés az *indx* indextől kezdődik. Ha van találat, akkor annak elejét megadó indexet adja vissza, ha nincs, **npos**-t kapunk vissza.
size_type rfind(const charT **str*, size_type *indx* = npos) const;	A *str* string utolsó előfordulását keresi meg a hívó stringben. A keresés az *indx* indextől kezdődik. Ha van találat, akkor annak elejét megadó indexet adja vissza, ha nincs, **npos**-t kapunk vissza.
size_type rfind(const charT **str*, size_type *indx*, size_type *len*) const;	A *str* string *len* hosszúságú kezdőszeletének utolsó előfordulását keresi meg a hívó stringben. A keresés az *indx* indextől kezdődik. Ha van találat, akkor annak elejét megadó indexet adja vissza, ha nincs, **npos**-t kapunk vissza.
size_type rfind(charT *ch*, size_type *indx* = npos) const;	A *ch* karakter utolsó előfordulását keresi meg a hívó stringben. A keresés az *indx* indextől kezdődik. Ha van találat, akkor annak indexét kapjuk vissza, ha nincs, **npos**-t kapunk.
size_type size( ) const;	A stringben található karakterek aktuális számát adja vissza.
basic_string substr(size_type *indx* = 0, size_type *len* = npos) const;	A hívó string *indx* pozíciójától kezdődő *len* hosszú részstringjét adja vissza.
void swap(basic_string &*str*);	A hívó string és az *str* string tartalmát cseréli ki.



Programozási tipp

A hagyományos C-stílusú karakterláncok használata sem volt nagyon bonyolult, de a C++ stringosztályok a stringkezelést rendkívül egyszerűvé teszik. Elég csak arra gondolnunk, hogy egy stringobjektumnak az = operátor használatával egyszerűen értékül tudunk adni egy idézőjelek közé zárt stringkonstanst, vagy arra, hogy relációs operátorok használhatók stringek összehasonlítására, nem is beszélve azokról a függvényekről, amelyek az részstringekkel történő manipulációkat teszik kényelmessé. Az alábbi program ezeket igyekszik alátámasztani:

```
// Példa stringekkel
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string str1 = "abcdefghijklmnopqrstuvwxyz";
  string str2;
  string str3(str1);

  str2 = str1.substr(10, 5);

  cout << "str1: "<< str1 << endl;
  cout << "str2: "<< str2 << endl;
  cout << "str3: "<< str3 << endl;

  str1.replace(5, 10, "");
  cout << "str1.replace(5, 10, \"\"): "
       << str1 << endl;

  str1 = "one";
  str2 = "two";
  str3 = "three";

  cout << "str1.compare(str2): ";
  cout << str1.compare(str2) << endl;
```

```
    if(str1<str2) cout << "str1 kisebb, mint
                          str2\n";

    string str4 = str1 + " " + str2 + " " +
                  str3;
    cout << "str4: " << str4 << endl;

    int i = str4.find("wo");
    cout << "str4.substr(i): " << str4.substr(i);

    return 0;
}
```

A program az alábbi kimenetet produkálja:

```
str1: abcdefghijklmnopqrstuvwxyz
str2: klmno
str3: abcdefghijklmnopqrstuvwxyz
str1.replace(5, 10, ""): abcdepqrstuvwxyz
str1.compare(str2): -1
str1 kisebb, mint str2
str4: one two three
str4.substr(i) wo three
```

Szembetűnő a könnyedség, amellyel az egyes manipulációkat véghez vittük. Például a kézenfekvő + jelet használtuk két string konkatenálására és a < operátorral hasonlítottunk össze két stringet. Ha ugyanezeket C-stílusú stringekkel szerettük volna megcsinálni, akkor a kevésbé kényelmes ***strcat( )*** *és* ***strcomp( )*** *függvényhívásokhoz kellett volna folyamodnunk. Mivel a C++* ***string*** *objektumai vegyesen használhatók a C-stílusú stringekkel, semmilyen hátrány nem származhat programjainkban történő használatukból – a látottak alapján szembetűnő előnyeiket fölösleges hangsúlyozni.*

■ *Kivételek*

A szabványos C++ könyvtár két fejállományt is tartalmaz, amelyek a kivételek használatához kapcsolódnak: az **<exception>** és az **<stdexcept>** fejeket. A kivételek hiba vagy nem normális állapotok jelzését szolgálják. A két könyvtár szolgáltatásait ebben a részben ismertetjük.

<exception>

Az **<exception>** fej olyan osztályokat, típusokat és függvényeket definiál, amelyek a kivételkezeléshez kapcsolódnak. Az **<exception>** osztályai a következők:

Osztály	*Szerepe*
exception	Minden kivétel alaposztálya a C++ standard könyvtára által definiált.
bad_exception	Az **unexpected( )** függvény által dobott kivételek típusa.

Az **<exception>** típusai:

Típus	*Jelentés*
terminate_handler	typedef void (*terminate_handler) ( );
unexpected_handler	typedef void (*unexpected_handler) ( );

A függvényeket az alábbi táblázatban foglaltuk össze:

Függvény	*Leírás*
terminate_handler set_terminate (terminate_handler *fn*) throw( );	A az *fn*-t terminálló függvénnyé teszi meg. A régi terminálló függvényre mutató pointert ad vissza.
unexpected_handler set_unexpected (unexpected_handler *fn*) throw( );	Az *fn* függvényt teszi meg nem várt hibaesemények kezelőjévé.
void terminate( );	Meghívja terminátor függvényt, ha valamilyen súlyos lekezeletlen hiba történt, az alapértelmezésben az **abort( )** függvényt hívja.
bool uncaught_exception( );	Ha a kivétel kezeletlen true értéket ad vissza.
void unexpected( )	Meghívja a nem várt kivételeket kezelő függvényt, amikor a függvény egy nem megengedett kivételt dob. Alapértelmezésben egy **terminate( )** függvényhívás történik.

<stdexcept>

A **<stdexcept>** fej számos szabványos kivételt definiál, amelyeket a C++ könyvtári függvényei és/vagy a futtató rendszer dobhatnak. Az **<stdexcept>** két általános típusú kivételt különít el: logikai hibákat és futásidejű hibákat. Logikai hibák a programozói hibákból fakadóan jönnek létre. Futásidejű hibák könyvtári függvények hibáiból, illetve a futtató rendszer hibáiból adódnak, ezek a programozó hatáskörén kívül esnek.

A szabványos C++ kivételek, amelyeket logikai hiba okoz, a **logic_error** alaposztály leszármazottai. Ezeket soroljuk fel itt:

Kivétel	*Jelentés*
domain_error	Értelmezési tartomány megsértése.
invalid_argument	Nem megengedett argumentum szerepeltetése függvényhíváskor.
length_error	Kísérlet történt egy túl nagy objektum létrehozására.
out_of_range	A függvény argumentum nem esett az előírt intervallumba.

Az alábbi futásidejű kivételek a **runtime_error** alaposztály leszármazottai:

Kivétel	*Jelentés*
overflow_error	Aritmetikai túlcsordulás történt.
range_error	Belső értéktartomány-megsértés történt.
underflow_error	Alulcsordulás történt.

Tárgymutató

Egyetemi Nyomda — 98.5381 Budapest, 1998
Felelős vezető: Sümeghi Zoltán igazgató