Írta:

FERENC RUDOLF

FEJLETT PROGRAMOZÁS

Egyetemi tananyag

2011

COPYRIGHT: 🅭 2011–2016, Dr. Ferenc Rudolf, Szegedi Tudományegyetem Természettudományi és Informatikai Kar Szoftverfejlesztés Tanszék

LEKTORÁLTA: Dr. Porkoláb Zoltán, Eötvös Loránd Tudományegyetem Informatikai Kar Programozási Nyelvek és Fordítóprogramok Tanszék

Creative Commons NonCommercial-NoDerivs 3.0 (CC BY-NC-ND 3.0)
A szerző nevének feltüntetése mellett nem kereskedelmi céllal szabadon másolható, terjeszthető, megjelentethető és előadható, de nem módosítható.

TÁMOGATÁS:
Készült a TÁMOP-4.1.2-08/1/A-2009-0008 számú, „Tananyagfejlesztés mérnök informatikus, programtervező informatikus és gazdaságinformatikus képzésekhez” című projekt keretében.

ISBN 978-963-279-498-3

KÉSZÜLT: a Typotex Kiadó gondozásában

FELELŐS VEZETŐ: Votisky Zsuzsa

AZ ELEKTRONIKUS KIADÁST ELŐKÉSZÍTETTE: Sosity Beáta

KULCSSZAVAK:
generikus programozás, C++, template, STL.

ÖSSZEFOGLALÁS:
A jegyzet fő célja, hogy az olvasó számára bemutassa a generikus programozási paradigmát. A könnyebb érthetőség kedvéért a bevezetésben egy rövid áttekintést nyújt az objektum-orientált programozásról, illetve a C++ nyelvről, majd ezután mutatja be a generikus programozást, valamint a legismertebb generikus programozással készült osztálykönyvtárat, a Standard Template Library-t (STL). A jegyzet betekintést nyújt az STL generikus algoritmusok és tárolók belső implementációjába és a tipikus használatába is. A jegyzet célja, hogy a teljesség igénye nélkül minél több területtel megismertesse az olvasót, ezzel megfelelő alapokat biztosítva a generikus programozási paradigma megértéséhez és elsajátításához.

TARTALOMJEGYZÉK

Bevezetés 6
Objektum-orientált programozás 8
  Interfész és implementáció 8
  Újrafelhasználhatóság 9
    Asszociáció, aggregáció 9
    Öröklődés 10
    Polimorfizmus 11
    Többszörös öröklődés 12
  Absztrakt osztályok 14
  Névterek 14
  Kivételkezelés 15
  Az objektumok élete 16
  Operáció-kiterjesztés 17
  This 18
  Operátor-kiterjesztés 19
Generikus programozás 20
  Sablonok 20
  Osztálysablonok 20
  Függvénysablonok 24
Standard Template Library (STL) 26
  A standard könyvtár szerkezete 26
  String osztály 27
  Saját sztring osztály 31
Folyamok 33
  Adatfolyamok 33
  Saját adatfolyam operátorok 35
  Fájlfolyamok 37
  Adatfolyam pufferezés 38
  Keresés az adatfolyamban 38
  Sztring folyamok 39
  Kimenő folyam formázása 41
  Manipulátorok 43
  Saját manipulátorok 44

Generikus programozási idiómák 46
Traits (jellemvonások)..46
Policy (eljárásmód) 49
Curiously recurring template pattern („szokatlan módon ismétlődő” saját ősosztály) 51
Template metaprogramozás 54
Kifejezés sablonok 56
A feladat 56
Egy egyszerű megoldás 56
Egy jobb megoldás 58
Egy teljes megoldás 61
Generikus algoritmusok összetevői 67
Generikus algoritmus használata 67
Predikátumok 70
Függvény objektumok 73
Függvény objektum adapterek 74
Adaptálható függvény objektumok 75
Függvény pointer adapterek 77
Generikus algoritmusok 79
Iterátorok 79
Feltöltés és generálás 80
Számlálás 82
Sorozatok manipulálása 83
Keresés és csere 85
Összehasonlítás 87
Elemek törlése 88
Rendezés 90
Keresés rendezett sorozatokban 91
Műveletek sorozat elemeken 92
Generikus konténerek 93
Példa konténer és iterátor használatára 93
Konténer kategóriák 95
Egyszerű sorozat konténerek 95
Vector 95
List 96
Deque 96
Származtatás STL konténerből 97

 © Ferenc Rudolf, SzTE

Iterátorok 99
Fordított iterátorok 99
Beszúró iterátorok 99
Egyszerű sorozat konténerek hasznos tagfüggvényei 101
Konténer adapterek 102
Stack 102
Queue 104
Priority_queue 105
Asszociatív konténerek 107
Map....108
Multimap 109
Set és multiset....110
Asszociatív konténerek hasznos tagfüggvényei 110
Köszönetnyilvánítás 112
Felhasznált irodalom 113

BEVEZETÉS

Jelen jegyzet a Fejlett Programozás tárgy írásos előadásjegyzete, a generikus programozási paradigmát mutatja be a C++ programozási nyelv segítségével, a Standard Template Library (STL) megvalósításán és használatán keresztül.
A C++ programozási nyelvet Bjarne Stroustrup fejlesztette ki az AT&T Bell Labs-nál, az 1980-as évek elején. Ez a C nyelv továbbfejlesztése, ami a következő lényeges dologgal egészült ki:

- támogatja az objektum-orientált tervezést és programozást (támogatja az adatabsztrakciót, az öröklődést, polimorfizmust és kései kötést),
- támogatja a generikus programozást, algoritmusokat,
- különböző hasznos kiegészítéseket biztosít a C nyelvi eszközeihez,

Feltételezzük, hogy az olvasó az objektum-orientált paradigmát jól ismeri, továbbá a C++ programozás alapvető fogásait a Programozás II. kurzus során elsajátította.
A jegyzet három fő részre bontható: C++ objektum-orientált programozás alapjainak átismétlése, generikus programozás és a Standard Template Library (STL) megvalósítása és használata.
Az ismétlés során szóba kerülnek olyan alapfogalmak, mint:

- osztályok - új típusok létrehozása, mezők, metódusok, kiterjesztés (*overloading*),
- implementáció elrejtése, névterek,
- újrafelhasználhatóság - kompozíció, aggregáció, öröklődés,
- felüldefiniálás (*overriding*), polimorfizmus, kései kötés,
- absztrakt és interfész osztályok, többszörös öröklődés, virtuális öröklődés,
- hibakezelés kivételekkel.

A jegyzet ezután ismerteti a generikus programozás alapjait a következő fogalmakon keresztül:

- sablonok (*template*-k),
- generikus programozási idiómák (*traits*, *policy*, *curiously recurring template pattern*),
- metaprogramozás,
- kifejezés sablonok (*expression templates*).

A *Standard Template Library* (STL) megvalósításának és használatának ismertetése során a következő fogalmak kerülnek áttanulmányozásra:

- STL alapok,
- sztringek, adatfolyamok,
- manipulátorok, effektorok,
- generikus algoritmusok, predikátumok,
- függvény objektumok, függvény objektum és pointer adapterek,
- iterátorok, rendezés, keresés, módosítás,
- generikus konténerek és adapterek,

A C++ standard könyvtár bemutatásának célja megértetni, hogyan használható a könyvtár: általános tervezési és programozási módszereket szemléltetni és megmutatni, hogyan bővíthető a könyvtár.

© Ferenc Rudolf, SzTE

A bemutatott fogalmak megértését egyszerű példák segítik, amelyek a már megismert információkra épülnek és a konkrét fogalom megértésére összpontosítanak. Általában a példaprogramokhoz egy futtatható tesztkörnyezet is társul, amely esetén a várt kimenet is ismertetésre kerül.

OBJEKTUM-ORIENTÁLT PROGRAMOZÁS

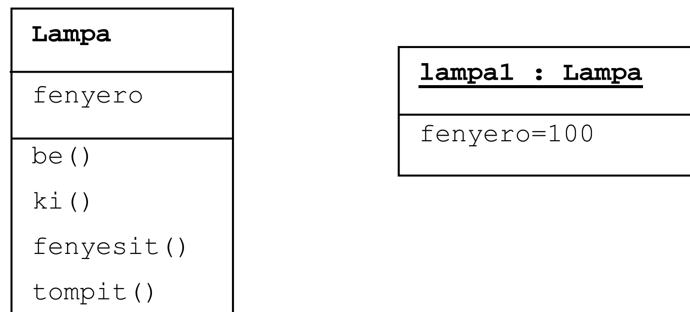
Az *objektum-orientált programozás* (OOP) fokozatosan felváltotta az elavulttá vált, klasszikusnak mondható *strukturált programozás*t. Az OOP hatékonyabban képes ábrázolni a való világot. Minden valóságos tárgyat nemcsak a rá jellemző adatok jellemeznek, hanem az is, hogyan viselkednek bizonyos körülmények között. Így a való világ elemei minden jellemzőivel együtt komplex egészként tekinthetők.
Vezessük be az OOP legfontosabb elemeit! A *program* egymással kommunikáló objektumok összessége. Az *objektum* a probléma egy elemének alkalmazhatóság-független absztrakciójaként tekinthető. Információkat tárol, és kérésre feladatokat hajt végre. Adatok és metódusok összessége, mely felelős feladatai elvégzéséért. Egyértelműen azonosítható, azonossága független az állapotától. Egy tisztán objektum-orientált programban minden objektum. Minden objektumot egyéb objektumokból állítunk össze, amelyek lehetnek alaptípusok is.
Az *osztály* az objektum típusa, egy absztrakt adattípus. A sok egyedi objektum között vannak olyanok, melyeknek közös tulajdonságai és viselkedési módjai vannak, vagyis egyazon családba – osztályba – tartoznak. Az objektum az osztály egy példánya. Ugyanolyan típusú objektumok ugyanolyan üzeneteket fogadhatnak. C++-ban a *class* kulcsszóval definiáljuk őket.
Az első objektum-orientált programozási nyelv a Simula-67 volt 1967-ből. A Simula-67 szimulációs célokra lett kifejlesztve, itt lett először az osztály fogalma bevezetve, mint az adatok és a rajta végezhető műveletek egységbezárása (*encapsulation*), valamint az öröklődés is megjelent.

Interfész és implementáció

Az objektum két különálló részre bontható: megkülönböztethetjük az objektum *interfész*ét és az *implementáció*ját. Az interfész maga a deklaráció, az implementáció pedig a megvalósítás, a definíció.
Célszerű a két rész külön kezelése, az implementáció elrejtése, hogy az osztály használója ne ismerje mi történik a háttérben, hogy van megvalósítva az egyes funkció.
Az információ elrejtése (láthatóság korlátozása) céljából háromféle elérés vezérlés (*access specifier*) állítható be: *public*, *private*, *protected*. A *public* (nyilvános) a legmagasabb szintű hozzáférést biztosítja. Az általa megjelölt típusok és tagok a program bármely pontjából elérhetők, használhatók. A *private* módosító a legalacsonyabb szintű hozzáférési módosító. A *private* típusok és tagok csak azokban az osztályokban használhatók, amelyben deklarálva lettek. A *protected* (védett) nagyon hasonlít a *private*-hoz. A különbség annyi, hogy a *protected* szintű típusok és tagok a származtatott osztályokon belül is láthatóak. A *friend kulcsszó* segítségével megadhatunk olyan barát osztályokat és függvényeket, amelyek hozzáférhetnek az adott osztály nem publikus típusaihoz, attribútumaihoz és metódusaihoz.
Egy osztály alapvetően *attribútum*okból és *operáció*kból van felépítve. Az attribútumok felelősek az osztály tulajdonságaiért. Az attribútum szinonimája az adattag vagy mező. Az operációk felelősek az osztály viselkedéséért. Az operáció szinonimája a metódus, tagfüggvény.
Az osztály implementációja tartalmazza az operációk tényleges megvalósítását.

 © Ferenc Rudolf, SzTE

Készítsünk egy *Lampa* osztályt, amelynek egy tulajdonsága van, a *fenyero*, valamint négy operációja van: ami kikapcsolja (*ki*), ami bekapcsolja (*be*) a lámpát, több fényt biztosít (*fenyesit*), illetve kevesebb fényt biztosít (*tompit*).
A *Lampa* osztály és a *lampa1* objektum egyszerűsített UML diagramja a következőképpen néz ki:



Az osztály az UML osztálydiagram alapján a következőképpen valósítható meg:

```
class Lampa {
      int fenyero;
public:
      Lampa() : fenyero(100) {}
      void be() {fenyero = 100;}
      void ki() {fenyero = 0;}
      void fenyesit() {fenyero++;}
      void tompit() {fenyero--;}
};
```

A *Lampa* osztály példányosítása pedig az alábbi módokon történhet:

```
Lampa lampa1;
lampa1.be();

Lampa *lampa1 = new Lampa();
lampa1->be();
```

Az első esetben lokális vagy tag objektumot hozunk létre közvetlen névvel, a második esetben a heap-en hozzuk létre az objektumot és pointer-rel hivatkozunk rá. Az objektum tagjainak elérése az első esetben a „.”, míg pointer esetén a „->” operátor segítségével történik.

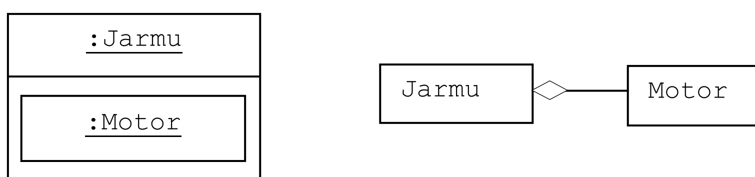
Újrafelhasználhatóság

Az újrafelhasználhatóság az OOP egyik legfontosabb előnye. Az újrafelhasználhatóság háromféleképpen történhet: *asszociáció*, *aggregáció* és *öröklődés* segítségével.

Asszociáció, aggregáció

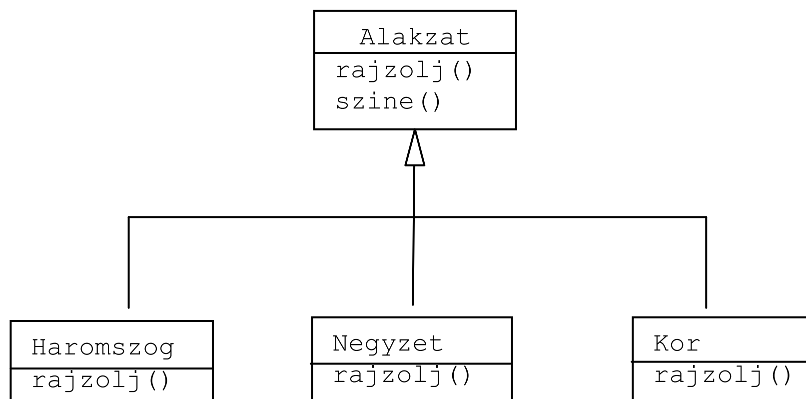
Az aggregáció az osztályok olyan kapcsolata, amely az egész és részeinek viszonyrendszerét fejezi ki. Az *asszociáció* az osztályok közötti kétirányú általános összeköttetés. Ez egy használati kapcsolat, létük általában egymástól független, de legalább az egyik ismeri és/vagy használja a másikat. Szemantikus összefüggést mutat. Általában az osztályokból létrejövő objektumok között van összefüggés.

Az *aggregáció* az asszociáció egy speciális formája, rész-egész kapcsolat, amely erősebb, mint az asszociáció. Itt az egyik objektum fizikailag tartalmazza vagy birtokolja a másikat. A rész-objektumok léte az egész objektumtól függ. Kétféle aggregációt különböztethetünk meg: az egyik a gyenge tartalmazás, azaz az általános aggregáció, a másik az erős tartalmazás, azaz a *kompozíció*, ahol a részek élettartama szigorúan megegyezik az egészével.
Nézzünk egy példát az aggregációra! Tegyük fel, hogy van egy *Jarmu* osztályunk. A *Jarmu* bizonyára rendelkezik motorral, tehát az osztály része lesz a *Motor* osztály. Ha kivesszük a járműből a motort, akkor az még jármű marad, bár elveszti funkcióját, tehát a jármű és a motor között aggregációs kapcsolat áll fenn. Ezt a kapcsolatot a következő UML diagramokkal ábrázolhatjuk:



Öröklődés

Az öröklődés egy olyan módszer, amely alkalmas már létező osztály újrafelhasználására. Célja, hogy hasonló osztályokat ne kelljen mindig újra implementálni. A közös rész kiemelésével létrejön az ősosztály, majd az ebből történő származtatással létrejönnek a speciális funkciókat ellátó leszármazott osztályok. A származtatással létrehozott osztály örökli az ősosztály tulajdonságait és funkcióit. Ezen kívül definiálhat új adattagokat és metódusokat, amelyek bővítik az ősosztály viselkedését. Egy osztály őse egy másik osztálynak, ha belőle lett az osztály leszármaztatva. Az öröklődés több szintű is lehet, így öröklődési hierarchia építhető fel. Az öröklődési hierarchiában felfelé haladva egyre általánosabb osztályokat találunk (*generalization*), míg lefelé haladva egyre speciálisabb viselkedésű osztályokat, azaz gyerekosztályokat találunk (*specialization*). Egy öröklődési kapcsolat két pontja az ős, szülő, alap (*base, super*) és a gyerek, leszármazott *(derived, child*). Öröklődés esetén a származtatott osztály egy új típus lesz. Ha az ős változik, a származtatott is „módosul". Abban az esetben, ha az ősosztály adattagja és/vagy metódusa *private* elérhetőséggel rendelkezik, a leszármazott osztály része lesz, de nem érheti el őket. Az ősosztály *protected* és *public* adattagjai és metódusai esetén a leszármazott osztály eléri az örökölt elemeket, azonban azok láthatóságát az öröklődés láthatósága határozza meg. Ha az öröklődés *public*, akkor az örökölt *protected* és *public* adattagok és metódusok láthatósága nem változik, ha az öröklődés *protected* vagy *private*, akkor az örökölt *protected* és *public* adattagok és metódusok láthatósága *protected* vagy *private* lesz, az öröklődés láthatóságának megfelelően.
Nézzünk egy példát az öröklődésre! Az alakzat egy általános fogalom, minden alakzatnak van színe, meg lehet rajzolni, stb. Azt azonban nem tudjuk definiálni, hogy hogyan kell egy alakzatot megrajzolni, mert minden alakzatot máshogyan kell. Ha egy konkrét alakzatra gondolunk, például egy háromszögre, akkor konkrétan meg lehet mondani, hogyan kell megrajzolni. Ha azonban egy körre gondolunk, akkor a rajzolás módja különbözik a háromszögétől. Tehát van egy általános funkciónk, hogy az alakzat rajzolható, de az, hogy hogyan, az a konkrét (specializált) alakzatok esetén mondható csak meg. A következő UML diagram ábrázolja az öröklődést és az örökölt metódus, a *rajzolj* más és más implementációját. A *szine* metódust nem szükséges specializálni, mivel ez csak egy tulajdonság lekérdezése minden alakzat esetén és nem függ az alakzat konkrét alakjától.



A származtatott osztály bővítését (specializálását) kétféleképpen tehetjük meg:

- attribútumokat és teljesen új operációkat veszünk fel, illetve
- átírjuk az őstől örökölt operációk működését, vagyis módosítjuk az ős viselkedését (az interfész marad). Ezt felüldefiniálásnak (*overriding*) nevezzük.

Polimorfizmus

A fenti példában a *rajzolj* metódus specializálásra került a leszármazott osztályokban. A felüldefiniálás *(overriding*) úgy módosítja az őstől örökölt viselkedést, hogy közben az interfészt nem módosítja. Egy metódus több megvalósításban is megjelenhet a leszármazott osztályokban. Ezeket a metódusokat a *virtual* kulcsszóval jelöljük meg, ez mutatja, hogy a leszármazott osztályokban felüldefiniálhatják az ősosztály egy metódusát. A *virtual* kulcsszó egy ún. *kései kötés* (*late binding*) mechanizmust aktivizál, ami lényegében azt jelenti, hogy a fordítóprogram a futási időre halasztja annak eldöntését, hogy ezen hívások során mely megvalósítás fog lefutni valójában. Ez a kései kötés mechanizmus teszi lehetővé az objektumok felcserélhetőségét (*polimorfizmus*) bizonyos szituációkban. A polimorfizmust ügyesen használva általánosabb és egyszerűbb programkód írható, melyet könnyebb a későbbiekben karbantartani.
Nem OOP esetében (hagyományos strukturális programozás pl. C nyelven) korai kötésről beszélhetünk, ahol már fordításkor biztosan eldől, hogy melyik meghívott operáció fut majd le, itt a hívott eljárás abszolút címe már fordítási időben megadásra kerül.
Nézzük meg a fenti UML diagram alapján az *Alakzat* osztály és a leszármazottai implementációjának főbb vonalát!

```
class Alakzat {
public:
	virtual void rajzolj() {/*...*/}
};
class Haromszog : public Alakzat {
public:
	virtual void rajzolj() {/*...*/}
};
class Negyzet : public Alakzat {
public:
	virtual void rajzolj() {/*...*/}
};
class Kor : public Alakzat {
public:
```

```
	virtual void rajzolj() {/*...*/}
};

void csinald(Alakzat& a) {
	// ...
	a.rajzolj();
}
```

Definiáljuk az *Alakzat* osztályt és származtatunk belőle három másik osztályt: *Haromszog*, *Negyzet*, *Kor*. Az, hogy egy osztály származik egy másik osztályból, onnan látható, hogy a „*class osztálynév*” és kettőspont után felsorolásra kerül(nek) az ősosztály(ok). A *csinald* metódus egy *Alakzat* típusú objektum hivatkozást vár, amelyre meghívja a *rajzolj* operációt (helyesebben fogalmazva: üzen az alakzatnak, hogy rajzolódjon ki). Mindegyik osztály megvalósítja a *rajzolj* metódust ugyanazzal az interfészszel, de más megvalósítással. Minden *rajzolj* metódus *virtual,* így a kései kötésnek köszönhetően majd a futás során dől el, hogy pontosan melyik megvalósítás fog lefutni attól függően, hogy milyen dinamikus típusú objektum (azaz milyen valódi típusú objektum) érkezik a *csináld* metódus paramétereként.
Hozzunk létre egy kört, egy háromszöget és egy négyzetet, majd rajzoljuk ki őket a *csinald* metódus segítségével a következő *main* függvény megvalósítással:

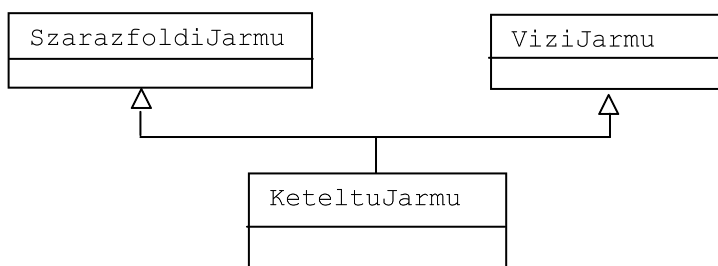
```
int main() {
	Kor k;
	Haromszog h;
	Negyzet n;
	csinald(k);
	csinald(h);
	csinald(n);
	return 0;
}
```

Mivel a kör is egy alakzat, ezért a *csinald* operáció paramétereként megfeleltethető felfele történő implicit típuskonverzió által. Az *upcast* ősre konvertálást jelent, így „elveszítjük” a konkrét típust. Ez egy biztonságos konverzió. (A *downcast* a típuskonverzió másik fajtája, leszármazottra konvertálást jelent, ami visszaállítja az eredeti típust. Ez a konverzió nem biztonságos, nem megfelelő gyerekosztályra való *downcast*-olás esetén nagy valószínűséggel hibás működés lép fel.) A *csinald* metódus így a paraméterben érkező *Kor* típusú objektumot már csak *Alakzat*-nak látja az implicit *upcast* miatt. Hagyományos korai kötés esetében az „*a.rajzolj();*” kifejezés egyszerűen meghívná az *Alakzat* osztály *rajzolj* metódusát, azonban mivel az virtuális, a fordítóprogram egy speciális utasítássorozatot generál a hagyományos függvényhívás helyett, amely az objektumhoz tartozó virtuális táblából kikeresi a *Kor rajzolj* metódusának címét és oda adja a vezérlést. *Haromszog* és *Negyzet* esetében is a *csinald* függvény megfelelően működik, és nem függ a speciális típusoktól. Ez a mechanizmus biztosítja a polimorfizmust, vagyis az objektumok felcserélhetőségét.

Többszörös öröklődés

C++-ban lehetőség van többszörös öröklődésre is, ami annyit takar, hogy egy osztálynak több őse is lehet az öröklődési hierarchia azonos szintjén, így több interfész újrafelhasználása történhet egyszerre. Névütközés esetén az elérés a „::” scope operátor segítségével történik, hogy meg lehessen különböztetni az azonos nevű osztályokat.
Nézzünk egy példát! Legyen az ősosztályunk a *Jarmu*. Származtassunk belőle két új osztályt, a *SzarazfoldiJarmu* és a *ViziJarmu* osztályt. Ekkor a *Jarmu* összes tulajdonságát megörökli a

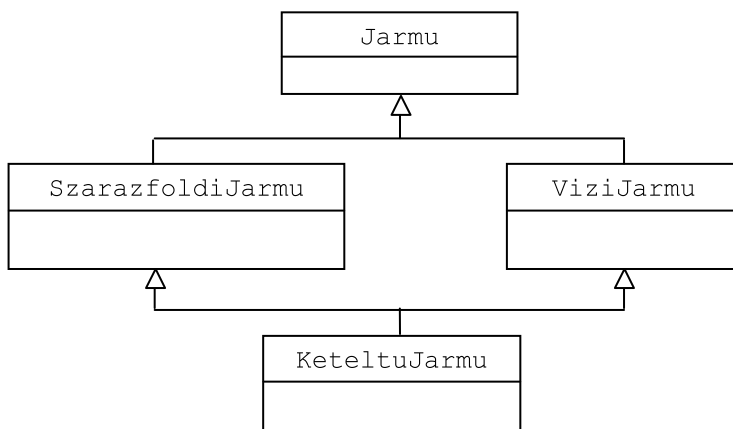
két leszármazott osztály. Ez a valóságban is megállja a helyét, mivel amit egy jármű tud, azt tudja a szárazföldi és a vízi jármű is, például elindul, megáll, stb. De hol helyeznénk el a hierarchiában a kétéltű járművet? Az is tud mindent, amit egy jármű, sőt, azt is tudja, amit a szárazföldi és a vízi jármű is tud. Tehát a *SzarazfoldiJarmu* és a *ViziJarmu* osztályból kell származtatni.
A többszörös öröklődésre mutat példát a *SzarazfoldiJarmu*, a *ViziJarmu és a KeteltuJarmu* osztály. Ezek osztályhierarchiáját mutatja be a következő ábra:



A UML diagram alapján a megvalósítás a következőképpen néz ki:

```
class SzarazfoldiJarmu {/*...*/};
class ViziJarmu {/*...*/};
class KeteltuJarmu : public SzarazfoldiJarmu, public ViziJarmu {
   /*...*/
};
```

A kétéltű jármű megörökli a mind a szárazföldi jármű, mind a vízi jármű tulajdonságait. De vonjuk be a hierarchiába a *Jarmu* osztályt is. Ekkor az öröklődési hierarchia a következőképpen néz ki:



Ezt nevezzük *gyémánt öröklődés*nek. Ez a fajta öröklődési hierarchiát körültekintően kell használni, mert a közös ős többszörösen is bekerülhet a gyerek objektumba. A *SzarazfoldiJarmu* és a *ViziJarmu* osztály tartalmazza a *Jarmu* osztály minden tulajdonságát és funkcióját, és a *KeteltuJarmu osztály* megörökli a *SzarazfoldiJarmu* és a *ViziJarmu* osztály minden tulajdonságát és funkcióját. Felmerülhet, a *KeteltuJarmu* kétszeresen örökli meg a *Jarmu* attribútumait és operációit? Azért, hogy ez ne történjen meg, az öröklődést *virtual* kulcsszóval kell ellátni. A helyes megvalósítás a következő példában látható:

```
class Jarmu {/*...*/};
class SzarazfoldiJarmu : virtual public Jarmu
{/*...*/};
class ViziJarmu : virtual public Jarmu
{/*...*/};
class KeteltuJarmu : public SzarazfoldiJarmu, public ViziJarmu
{/*...*/};
```

Absztrakt osztályok

Az *Alakzat* osztály egy tipikus absztrakt osztály, mivel nincs értelme belőle konkrét objektumot létrehozni, annyira általános. Egy ilyen meghatározatlan alakzatot meg lehet adni (a nyelv megengedi), de nem sok értelme van létrehozni belőle egy objektum példányt. Pl. nem tudnánk, hogyan is néz ki. Mivel azonban rendelkezik olyan tulajdonságokkal és operációkkal, amelyek az alakzatokat jellemzik, ezért az osztály interfésze hasznos lehet. Az *Alakzat* osztály virtuális függvényeit tisztán virtuális (*pure virtual*) függvényként deklaráljuk, ahol a virtuális függvények deklarációjában a törzse helyett az „*=0”* kifejezés szerepel. A virtuális függvényt csak akkor kell definiálni, ha pontosan ezt akarjuk meghívni.
Ha egy osztály legalább egy tisztán virtuális függvénnyel rendelkezik, akkor *absztrakt osztály*nak (elvont osztály, *abstract class*) hívjuk, ilyen osztályba tartozó objektum pedig nem hozható létre.

```
class Alakzat {
public:
   virtual void rajzolj() = 0;
};
int main() {
   Alakzat a;  // fordítási hiba
   return 0;
}
```

Az absztrakt osztály nagyon hasznos, mert különválasztja az interfészt az implementációtól: csak egy formát ad, implementáció nélkül. Egy protokollt valósít meg az osztályok között.

Névterek

A névtér (*namespace*) egyfajta hatókörként (*scope*) fogható fel. Minél nagyobb egy program, annál hasznosabbak a névterek, hogy kifejezzék a program részeinek logikai elkülönítését. Az alapértelmezett névtér a *global namespace*. Névegyezés esetén fontos a névtér használata, hogy meg lehessen különböztetni az azonos nevű osztályokat, függvényeket. Névtér definiálása a *namespace* kulcsszóval lehetséges, névtér használata közvetlenül a „::” scope operátorral történhet, vagy a *using namespace* utasítással.
Nézzük meg, mi történik, ha az *Alakzat* osztályt és leszármazottait egy *rajz* névtérbe helyezzük!

```
namespace rajz {
   class Alakzat {
   public:
      virtual void rajzolj() = 0;
   };
   class Haromszog : public Alakzat {
   public:
      virtual void rajzolj() {/*...*/}
```

 © Ferenc Rudolf, SzTE

```
   };
   class Negyzet : public Alakzat {
   public:
      virtual void rajzolj() {/*...*/}
   };
   class Kor : public Alakzat {
   public:
      virtual void rajzolj() {/*...*/}
   };
}  // rajz
```

Ekkor a *csinald* és *main* függvényekben vagy a teljes névvel hivatkozhatunk, ahogyan az alábbi példa mutatja,

```
void csinald(rajz::Alakzat& a) {
   // ...
   a.rajzolj();
}
int main() {
   rajz::Kor k;
   csinald(k);
   return 0;
}
```

vagy a *using namespace* utasítás segítségével „megnyitjuk" a névteret az alábbi példa szerint:

```
using namespace rajz;
void csinald(Alakzat& a) {
   // ...
   a.rajzolj();
}
int main() {
   Kor k;
   csinald(k);
   return 0;
}
```

Kivételkezelés

A kivételkezelés (*exception handling*) segítségével a futási időben történő hibákat lehet hatékonyabban kezelni. A kivétel egy olyan helyzet, amikor a programban egy olyan váratlan esemény következik be, ami alapesetben nincs explicit módon lekezelve. Egy ilyen állapot megszakítja a program rendes futását, azonban a kivételkezelés módszerével megoldható, hogy ahhoz a programrészhez kerüljön a vezérlés, amely képes az adott kivételt megfelelő módon kezelni.
Bár a kivételkezelés nem objektum-orientált sajátosság, a C++ programozási nyelvben rendkívül hasznos, mivel könnyebbé teszi a tényleges feladat végrehajtásáért felelős programkód és a hibakezelést megvalósító kódrészletek elválasztását, átláthatóbbá téve ily módon a teljes kódot. C++ környezetben a kivétel mindig egy objektum, ami a kivétel bekövetkeztekor jön létre, a kivételkezelés pedig egyszerűen az alábbi három elem segítségével valósítható meg:

- *try*: Védett régió, amelyben a programkód „érdemi" része található, és amelyben felléphetnek hibák, de azokkal nem helyben foglalkozunk
- *throw*: A hibát reprezentáló kivétel objektum „eldobása",

- *catch*: A kivételek elkapása és kezelése.

A *catch* blokk gyakorlatilag egy párhuzamos végrehajtási ág rendkívüli esetekre. Mivel a rendkívüli esetek ez által külön vannak kezelve, tisztább marad a kód, és nem lehet ignorálni a hibát (míg a hibakóddal visszatérő függvényt igen). Nem várt események esetén is megbízhatóan helyreállítható így a program futása.

Az objektumok élete

A C++ objektumok tárolási helyei a következők lehetnek:

- *stack*: automatikus és gyors, de nem mindig megfelelő, a felszabadítás automatikus.
- *static:* statikus, nem flexibilis, de gyors.
- *heap:* dinamikus, futás közbeni, lassúbb, felszabadítás kézzel történik.

Jellemző, hogy olyan objektumokat hozunk létre, amelyeket akkor is fel szeretnénk használni, miután visszatértünk abból a függvényből, ahol létrehoztuk azokat. Az ilyen objektumokat a *new* operátor hozza létre és a *delete* operátort használhatjuk azok törlésére. A *new* által létrehozott objektumok *heap*-en tárolt objektumok, a dinamikus memóriában vannak tárolva. Régebbi nyelvek esetében sok problémát okozott az inicializálás és eltakarítás hiánya. C++-ban ezt a problémát oldja meg a konstruktor és a destruktor.
A *konstruktor* az objektum létrehozásakor hívódik meg. A konstruktor egy metódus, melynek neve megegyezik az osztály nevével, és garantálja a létrejött objektum inicializálását. Helyette lehetne hívni pl. egy *initialize* függvényt is, de ezt mindig kézzel kellene meghívni, szemben a konstruktorral, amit a *new* operátor automatikusan meghív.
Hozzuk létre az Alakzat osztály konstruktorát!

```
class Alakzat {
   Alakzat() {
         /* inicializáló kód */
   }
};
```

A konstruktor egy speciális metódus. Lehet paraméter nélküli (*alapértelmezett/default constructor*), de lehet paramétert is megadni neki, tipikusan az osztály attribútumainak kezdőértékeit lehet vele beállítani. Paraméterek hiányában az attribútumok alapértelmezett kezdőértéket vesznek fel. Ha nem definiálunk egy konstruktort sem, akkor a fordító készít egy alapértelmezett konstruktort, azonban ha már van valamilyen (akár alapértelmezett akár nem), akkor nem készít.
A konstruktornak nincs visszatérési értéke, más függvényekkel szemben (még *void* sem). Az objektumra való hivatkozást/mutatót kapunk a *new* operátortól. Nézzünk egy példát paraméterekkel rendelkező konstruktorra és annak meghívására!

```
class Alakzat {
public:
   Alakzat(int x, int y) {
       /* inicializáló kód */
   }
};

int main() {
    Alakzat a(10,15);
    Alakzat *pa = new Alakzat(10,15);
```

```
    return 0;
}
```

Az *Alakzat* konstruktora az *x* és *y* egész típusú paraméter felhasználásával inicializálja az attribútumait. Ezután létrejön egy *a* nevű alakzat a *stack*-en, majd egy *pa Alakzat*-ra mutató pointer, mely a *new* kifejezés segítségével a *heap*-en létrehozott alakzat objektumra mutat.
C++-ban nincs automatikus szemétgyűjtés (*garbage collection*), a programozónak magának kell gondoskodnia az objektumok eltakarításáról. C++-ban a *destruktor* hívódik meg minden objektum törlésekor. A destruktor neve megegyezik az osztály nevével, csak kap egy ~ prefixet elé. Ahogy a konstruktornak, a destruktornak sincs visszatérési értéke, azonban nem lehetnek paraméterei sem.
Nézzük meg az *Alakzat* osztály destruktorát!

```
class Alakzat {
public:
   Alakzat(int x, int y) {
       /* inicializáló kód */
   }
   ~Alakzat() {
       /* takarító kód */
   }
};

int main() {
      Alakzat a(10,15);
      Alakzat *pa = new Alakzat(10,15);
      delete pa;
      return 0;
}
```

A destruktor meghívása a *delete* operátor segítségével történik, ezáltal az adott objektum törlésre kerül.

Operáció-kiterjesztés

Magasabb szintű nyelvekben neveket használunk. Természetes nyelvben is lehet több értelme a szavaknak, ilyenkor a szövegkörnyezetből derül ki az értelme. Programozásban ezt nevezzük *overloading*-nak vagy *kiterjesztés*nek (egyes szakkönyvek túlterhelésnek is nevezik), ami nem keverendő az *overriding* fogalmával, ami felüldefiniálást jelent öröklődés esetén. Régebbi nyelvekben, például C-ben, minden név egyedi volt (nincs *printf int*-re és *float*-ra külön-külön). C++-ban szükségessé vált ennek használata. Például, ha a konstruktornak csak egy neve lehet, mégis különböző inicializálást szeretnénk megadni.
A megoldás a metódusok kiterjesztése (nem csak konstruktorra). Több metódusnak is ugyanaz lesz a neve, de más a paraméterlistája. Hasonló funkció végrehajtásához miért is kellene különböző nevű függvényeket definiálni?
A következő kódrészlet arra mutat példát, hogy egy osztály rendelkezhet több konstruktorral is.

```
class Alakzat {
public:
  Alakzat() {/*...*/}
  Alakzat(int x, int y) {/*...*/}
  ~Alakzat() {
```

```
    /* takarító kód */
  }
};
```

Hogyan különböztetjük meg, hogy melyik kiterjesztett metódust hívtuk? A paraméterlistáknak egyedieknek kell lenniük. A hívás helyén az aktuális argumentumok száma és típusai határozzák meg. Konvertálható primitív típusú argumentumok esetében, ha nincs pontos egyezés, akkor az adat automatikusan konvertálódik. A metódus visszatérési értéke nem használható megkülönböztetésre.

This

Egy függvény kódja mindig csak egy példányban van a memóriában. De honnan tudja a *rajzolj* függvény, hogy melyik objektumhoz lett hívva? Egy „titkos” implicit első paramétert (*this*) generál a fordító.
A *this* explicite is felhasználható, például ha a metódus formális paraméterneve megegyezik valamelyik mező nevével:

```
this->x = x; // az első az attribútum
```

A *rajzolj* nevű metódus paraméter nélkül hívható meg, maga a hívó objektum fog kirajzolódni:

```
class Alakzat {
public:
      /*...*/
      void rajzolj() {/*...*/}
};

Alakzat a1;
Alakzat a2;
a1.rajzolj();
a2.rajzolj();
```

A helyzetet a legkönnyebb úgy elképzelni, mintha a fordítás során az osztály le lenne butítva C struktúrára, a metódusai pedig globális függvényekké lennének alakítva és egy új első paraméter generálódna hozzájuk:

```
struct Alakzat {/*...*/};
void rajzolj(Alakzat *this) {/*osztályon kívül van!*/}

Alakzat a1;
Alakzat a2;
rajzolj(&a1);
rajzolj(&a2);
```

A hívás helyén pedig az objektum címe kerülne átadásra, ami *this* néven érkezik a függvényekhez.

Operátor-kiterjesztés

A C++ programozási nyelv lehetőséget biztosít arra, hogy kiterjesszük a nyelvben definiált bináris és unáris operátorokat. Az operátor kiterjesztés növeli az absztrakciót, egyszerűbbé és könnyebben olvashatóbbá teszi az absztrakt adattípusokkal való munkát. A következőkben nézzük meg, hogyan lehet az *Alakzat* osztályunkra az == operátort megvalósítani:

```
#include <iostream>

class Alakzat {
public:
      Alakzat(int sz) : szin(sz) {}
      bool operator==(const Alakzat& a) const {return szin == a.szin;}
private:
      int szin;
};

int main() {
  Alakzat a1(1);
  Alakzat a2(2);
  if (a1 == a2)
      std::cout << "egyformak" << std::endl;
  else
      std::cout << "kulonboznek" << std::endl;
  return 0;
}
```

Azt mondjuk, hogy két alakzat megegyezik, ha azonos a színük. Az == operátor megvalósítása után pl. az *if* feltételben alkalmazható az *a1 == a2* kifejezés.

GENERIKUS PROGRAMOZÁS

A *generikus programozás* egy általános programozási modellt jelent. Maga a technika olyan programkód írását foglalja magába, amely nem függ a program egyes típusaitól. Ez az elv növeli az újrafelhasználás mértékét, hiszen típusoktól független tárolókat és algoritmusokat lehet a segítségével írni. Például egy absztrakt adatszerkezetet (mondjuk egy láncolt listát) logikus úgy tervezni és megvalósítani, hogy bármi tárolható lehessen benne. A funkcionális paradigmát használó nyelvekben (ML, Haskell, Scala) a *parametrikus polimorfizmus* fogalmat használják erre a modellre. Az első alkalmazása az Ada nyelvben jelent meg.

Sablonok

A C++-ban a generikus programozásra használt fogalom a sablon (*template*). Objektum-orientált programozási nyelv lévén generikus osztályokat és függvényeket lehet létrehozni. Ezek az osztálysablonok és függvénysablonok. Tervezési szempontból nagy hasonlóságot mutat az öröklődéssel, mivel a kód polimorfizmusát, többalakúságát teszi lehetővé. Ezért szokták *fordítási idejű polimorfizmus*nak is nevezni. Ez az elnevezés onnan ered, hogy a sablonok fordítás közben példányosodnak szokásos osztályokká, illetve függvényekké.
A sablonok létrehozására szolgáló kulcsszó a *template*. Ezután < és > jelek között adható neki egy vagy több paraméter, amelyek nem csak osztályok (típusnevek) lehetnek, hanem konstansok és sablonok is. Fontos megjegyezni, hogy az osztálysablon deklarálása során a típus paramétert jelző *class* kulcsszó helyett a *typename* szó is használható, mivel C++-ban minden típus egyben osztály is, tehát a két fogalom ebben a kontextusban ekvivalens.

Osztálysablonok

A következőkben egy példa bemutatatásával részletezésre kerül az osztálysablon létrehozásának, példányosításának módja.
A következő példában egy generikus tömb megvalósítása látható:

```
#include <iostream>
#include <stdexcept>

template<class T, int size>
class Array {
	T a[size];
public:
	T& operator[](int i) {
		if (i < 0 || i >= size)
			throw std::out_of_range("rossz index");
		return a[i];
	}
};
```

A generikus tömb két paramétere az elemeinek a típusa (*T*) és a tömb mérete (*size*). Valójában ez az osztálysablon egy hagyományos tömb reprezentációt egy osztály reprezentációba csomagol, elrejtve a háttérben zajló műveleteket.
Sablonok esetében nagyon fontos, hogy a definíció azonos fordítási egységben kell, hogy szerepeljenek a deklarációval, különben fordítási hiba lép fel. Az előző példa nem különítette

© Ferenc Rudolf, SzTE

el a deklarációt a megvalósítástól, nézzük meg, hogyan lehetne úgy elkülöníteni, hogy ne kapjunk fordítási hibát:

```
#include <iostream>

template<class T, int size>
class Array {
	T a[size];
public:
	T& operator[](int i);
};
template<class T, int size>
T& Array<T,size>::operator[](int i) {
	if (i < 0 || i >= size)
		/*...*/; // hiba
	return a[i];
}
```

Hozzunk létre két generikus tömböt, töltsük fel és írassuk ki a tartalmukat a következő *main* függvény megvalósítással:

```
int main() {
	const int s = 20;
	Array<int,s> ia;
	Array<double,s> fa;
	for(int i = 0; i < s; i++) {
		ia[i] = i * i;
		fa[i] = i * 1.414;
	}
	for(int j = 0; j < s; j++)
		std::cout << j << ": " << ia[j] << ", " << fa[j] << std::endl;
	return 0;
}
```

Az osztálysablon példányosítása során a sablonparaméterek konkrét típusokat, értékeket kapnak. Jelen példában az első sablonparaméter *int* típust kap, ami azt jelenti, hogy egészeket fog tárolni a tömb, a második sablonparaméter értéke pedig 20, azaz a tömb mérete 20 lesz.

Egy sablon példányosítása során a fordító a következőképpen viselkedik. Amikor a fordítóprogram egy sablon-definícióhoz ér, megvizsgálja azokat a szintaktikus szabályokat, amelyek a paraméterek ismerete nélkül is eldönthetőek, majd félreteszi egy gyűjteménybe, és csak a példányosításnál veszi elő újra. Példányosításkor a sablonból egy osztály keletkezik a legközelebbi *namespace scope*-ban. Tehát ha van egy *Array<int,20>* és *Array<double,20>* sablon, akkor a fordító két osztályt hoz létre, majd ezeket az osztályokat példányosítja objektumokká. Viszont ha azonos argumentumokkal van többször példányosítva a sablon, akkor a fordítóprogram csak egy osztályt generál. Tehát pl. két *Array<unsigned int, 20>* típusú objektum ugyanabból a sablonból generált osztály típusú lesz. Ugyanez történik *typedef* esetében is, hiszen az nem jelent mást, csak egy típus átnevezést. Ennek értelmében tehát pl. az *Array<unsigned int,20>* és az *Array<size_t,20>* ugyanazt a generált osztályt jelentik (*typedef unsigned int size_t*). Továbbá a fordítóprogramok felismerik a fordítási időben kiértékelhető konstans kifejezéseket, így például az *Array<int, 30-10>* egy 20 darab egész számot tároló tömböt fog jelenteni.
A példányosítás egy kicsit leegyszerűsítve úgy zajlik, hogy a fordítóprogram a sablon alapján minden különböző argumentumlista esetében egy-egy igazi osztályt készít, melynek generál

egy nevet és a sablonparaméterek minden előfordulását behelyettesíti a paraméter értékével. Például a fenti *Array<int,s> ia* példányosítás képzeletben forráskódként ábrázolva így nézne ki (az osztály neve persze ennél bonyolultabb lesz a valóságban):

```
class Array_int_20 {
	int a[20];
public:
	int& operator[](int i) {
		if (i < 0 || i >= 20)
			/*...*/; // hiba
		return a[i];
	}
};

int main() {
	const int s = 20;
	Array_int_20 ia;
	/*...*/
	return 0;
}
```

A program futtatása után kiírásra kerül 0 és 20 között a számok négyzete, illetve 1,414-szerese.

Osztálysablonok paramétereinél lehetőség van alapértelmezés megadására is:

```
template<class T, int size=20>
class Array {
	T a[size];
public:
	T& operator[](int i) {
		if (i < 0 || i >= size)
			/*...*/; // hiba
		return a[i];
	}
};

int main() {
	Array<int> ia;  // a mérete alapértelmezetten 20 lesz
	/*...*/
	return 0;
}
```

Ha egy sablonnál bizonyos paraméterhalmazok esetében valamit hatékonyabban meg lehet oldani, mint általános esetben, akkor külön implementálhatóak a sablon specializált változatai, amelyek majd akkor kerülnek használatba, ha azokkal a bizonyos típusokkal vagy értékekkel kerülnek példányosításra. Ezt *sablon specializálás*nak nevezzük. Ebben az esetben létre kell hozni egy új template osztályt, amelynek sablonparaméterei lényegében megegyeznek az eredetivel, kivéve a specializált paramétereket, ugyanis ezeket ott nem kell kiírni, az osztály neve után szereplő argumentumlistában viszont az összes paramétert meg kell adni. A következő példa ezt szemlélteti:

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

 © Ferenc Rudolf, SzTE

```
template<class T1, class T2>
class C {/*...*/};

template<class T2>
class C<int, T2> {/*...*/};

template<class T1, class T2>
ostream& operator<<(ostream& os, const C<T1,T2>& s)
{
      return os << "általános";
}

template<class T >
ostream& operator<<(ostream& os, const C<int,T>& s)
{
      return os << "specializált";
}

int main() {
      C<char,float> cc;
      C<int,float> ci;
      cout << cc << endl << ci << endl;
      return 0;
}
```

Akár még azt is megtehetjük, hogy az eredeti sablon felületét (interfészét) megváltoztatjuk azzal, hogy függvényeket törlünk és/vagy adunk hozzá a specializált változathoz, de ez tervezési és használati szempontokból félrevezető lehet a fejlesztő számára.
Egy további lehetősége ennek a programozási technikának, hogy a korábban megadott paramétereket felhasználhatjuk a későbbi paraméterekben, hasonló módon, mint ahogy a következő példa mutatja:

```
template<class T, T value>
class A {
      // …
};

int main() {
      A<int,10> a;
      return 0;
}
```

Az osztálysablonok paraméterei között sablon is szerepelhet. Nézzük az alábbi *Array1* és *Array2* osztálysablonokat:

```
class Container {
      // …
};

template <class T>
class TContainer {
      // …
};

template <class T, class Cont>
class Array1 {
      // …
```

```
};

template <class T, template <class> class Cont>
class Array2 {
      // …
};
```

Az első tömb osztálysablon egy tároló osztályt használ az egyes elemek tárolására. A második sablon definíció azonban egy megszorítást is ad a tároló osztályra, ugyanis az csak olyan sablon lehet, amelynek egy sablonparamétere van, és az egy típus. Lássuk a sablonok példányosítási módját is:

```
int main() {
      Array1<int,Container> array1;
      Array2<int,TContainer> array2;
}
```

Az első esetben egy konkrét típust kell átadni, a második példában pedig egy sablont kell átadni.
A sablonok öröklésre is képesek, sőt, egyszerű osztály(ok)ból is származhatnak. Az öröklődési mechanizmus ugyanúgy működik, mint az egyszerű osztályok esetében, mivel példányosításnál úgyis osztályok keletkeznek a sablonokból.
Létre lehet hozni az egyes sablonokban statikus tagokat is, viszont ezek viselkedése kicsit eltérő a hagyományos osztályokétól. Mivel minden sablonból a példányosítás során egy osztály keletkezik, így a generált osztályra fog a szokásos módon viselkedni a statikus adattag. Így ha példányosításra kerül egy *Array<int>* és egy *Array<double>* sablon, azokhoz külön statikus tagok tartoznak. Viszont, ha később létrehozunk még egy *Array<int>* objektumot, akkor a statikus adattagja az elsővel osztozik majd. Azt megoldani, hogy mindhárman egy tagot használjanak, egy közös nem sablon ősosztály bevezetésével lehet megoldani. Ezt mutatja be a következő példa, ahol egy közös ősosztályban deklaráltunk egy statikus adattagot, amely számolja, hogy mennyi példány lett összesen létrehozva.

```
class ArrayBase {
public:
      static int numOfArrays;
};

int ArrayBase::numOfArrays = 0;

template<class T, long size>
class Array: public ArrayBase {
      /*...*/
};
```

Függvénysablonok

A függvénysablonokat hasonlóan kell definiálni, mint az osztálysablonokat, csak itt az osztályok helyett a függvények elé írjuk a *template* kulcsszót és paraméterlistát. A következő kódrészletben erre látunk példát:

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

template<class T>
```

```
void myswap(T &a, T &b) {
    T t = a;
    a = b;
    b = t;
}

template<class T, int size>
void sort(T arr[]) {
      for(int i = 1; i < size; i++) {
            int j = i;
      while(0 < j && arr[j] < arr[j-1]) {
                  myswap<T>(arr[j], arr[j-1]);
                  --j;
            }
      }
}
```

A *sort* függvénysablon egy *T* típusú és *size* méretű tömb elemeit rendezi növekvő sorrendbe. A függvény a tömbön belül mozgatja az elemeket, az eredmény is ugyanabban a tömbben áll elő.

Hozzunk létre egy egészeket tartalmazó 5 hosszúságú tömböt és rendezzük az elemeit növekvő sorrendbe a következő *main* függvény megvalósítással:

```
int main() {
      int tomb[5] = { 3, 5, 4, 1, 2 };
      sort<int,5>(tomb);
      for (int i = 0; i < 5; i++)
            cout << tomb[i] << " ";
      cout << endl;
      return 0;
}
```

A program futtatása után a következő eredményt kapjuk:

```
1 2 3 4 5
```

Fontos, hogy a paraméterben megadott *T* típus mindig megvalósítsa a < operátort, hogy a T típusú objektumok összehasonlíthatók legyenek a rendezés során, egyébként fordítási hibát kapunk. A példában szereplő *int* típusra természetesen alapból értelmezve van a < operátor, de saját osztályok esetében erről magunknak kell gondoskodnunk.
Egy további tulajdonsága a függvénysablonoknak, hogy míg az osztályok esetében megengedettek a *default* paraméterek, a függvényeknél ez fordítási hibát okoz. Ezentúl a sablonfüggvényeknél is lehetőség van a kiterjesztésre, ha a sablonparaméterek egyértelműek, levezethetőek maradnak. Ez tehát a *függvény overloading* egy általánosítása.
A függvény- és osztálysablonok használhatók együtt is, tehát lehetőség van sablonosztályban sablonfüggvények létrehozására.

STANDARD TEMPLATE LIBRARY (STL)

A Standard Template Library egy C++ sablonosztály-könyvtár, amelyet 1994-ben mutattak be a C++ szabvány bizottságnak. Számos gyakran használt generikus osztályt és algoritmust tartalmaz, magában foglalja a számítástudomány fontosabb algoritmusait és adatszerkezeteit, így segítségével rengeteg programozási probléma megoldható anélkül, hogy bármilyen saját általánosabb osztályt kellene írni, amely általában szükségeltetik egy nagyobb fejlesztési projekt során (mint a láncolt lista és egyéb tárolók, vagy az azokon végzett műveletek). Tervezéskor a hatékonyságot tartották szem előtt, így kellően gyorsak a legtöbb alkalmazási területen, továbbá az itt megvalósított algoritmusok és adatszerkezetek függetlenek egymástól, de képesek együttműködni. Nagyon fontos, hogy a könyvtár algoritmusai, adatszerkezetei bővíthetőek, tehát ha a szabályoknak megfelelő osztályokat hozunk létre, akkor az STL algoritmusai azokon is működni fognak. Az STL készítésénél a tervezők többféle szempontot is figyelembe vettek[1]:

- Segítséget jelentsen mind a kezdő, mind a profi felhasználóknak.
- Elég hatékony ahhoz, hogy vetélytársa legyen az általunk előállított függvényeknek, osztályoknak, sablonoknak is.
- Legyen - matematikai értelemben - primitív. Egy olyan összetevő, amely két, gyengén összefüggő feladatkört tölt be, kevésbé hatékony, mint két önálló komponens, amelyet kimondottan arra a szerepre fejlesztettek ki.
- Nyújtson teljes körű szolgáltatást ahhoz, amit vállal.
- Legyen összhangban a beépített típusokkal és műveletekkel és bátorítsa a használatukat.
- Legyen típusbiztos, és bővíthető úgy, hogy a felhasználó a saját típusait az STL típusaihoz hasonló módon kezelhesse.

A standard könyvtár szerkezete

A standard könyvtár szolgáltatásait az *std* névtérben definiálták, és header fájlokban érhetjük el azok deklarációit (illetve sablon esetén a megvalósítást is). Ha egy fejállomány neve *c* betűvel kezdődik, akkor az egy C-beli könyvtár megfelelője. Minden *<X.h>* header fájlhoz, amely a standard C könyvtár részét képezi, megvan a C++-beli megfelelője is az *std namespace*-ben *<cX>* néven (*.h* kiterjesztés nélkül). Az eredeti C-beli könyvtárak továbbra is elérhetők a globális névtérben. A fontosabb könyvtárak a következők[2]:

- Tárolók: *<vector>, <list>, <deque>, <queue>, <stack>, <map>, <set>, <bitset>*. Ezek a fájlok az azonos nevű sablonokat tárolják, amelyek a nevükben megadott adatszerkezeteket reprezentálják.
- Iterátorok: *<iterator>*. A fenti tárolók bejárását segítik az iterátorok.
- Algoritmusok: *<algorithm>, <cstdlib>*. Az első fájl általános algoritmusokat tárol, a második fájl pedig az *<stdlib.h>* C-beli könyvtár megfelelője.

[1] A teljes listát lásd: Bjarne Stroustrup: A C++ programozási nyelv, 565.oldal
[2] Egy teljesebb leírás: Bjarne Stroustrup: A C++ programozási nyelv, 566-571.oldal

© Ferenc Rudolf, SzTE

- Általános eszközök: *<utility>, <functional>, <memory>, <ctime>*. Ezek a fájlok a memóriakezeléssel foglalkoznak, függvényobjektumokat biztosítanak, illetve a C-szerű dátum- és időkezelést teszik lehetővé.
- Ellenőrzések, diagnosztika: *<exception>, <stdexcept>, <cassert>, <cerrno>*. Ezek a modulok a szabványos kivételeket, a hibaellenőrző makrót, valamint a C-szerű hibakezelést biztosítják.
- Karakterláncok: *<string>, <cctype>, <cwtype>, <cstring>, <cwchar>, <cstdlib>*. Az első állomány egy új sztring osztály, a többi pedig C-ből öröklődött.
- Ki- és bemenet: *<iosfwd>, <iostream>, <ios>, <streambuf>, <istream>, <ostream>, <iomanip>, <sstream>, <cstdlib>, <fstream>, <cstdio>, <cwchar>*. Ezek a header fájlok a *stream* kezelését biztosítják, illetve a visszafele kompatibilitást őrzik meg a C-vel.
- Nemzetközi szolgáltatások: *<locale>, <clocale>*. Segítségükkel könnyebben megvalósíthatók a többnyelvű szoftverek. Kulturális eltérések meghatározására szolgál. Például az eltérő dátumformátumokat, karakterrendezési szabályokat könnyebben kezelhetjük ezekkel a programkódokkal.
- A programnyelvi elemek támogatása: *<limits>, <climits>, <cfloat>, <new>, <typeinfo>, <exception>, <cstddef>, <cstdarg>, <csetjmp>, <cstdlib>, <ctime>, <csignal>*. Ezek az állományok főként a típusinformációkhoz való hozzáférést, régebbi C-s könyvtárak elérését, kivételkezelését biztosítják.
- Numerikus értékek: *<complex>, <valarray>, <numeric>, <cmath>, <cstdlib>*. Ezek az állományok többnyire matematikai műveletekhez adnak hozzáférést.

String osztály

A *string* az egyik legtöbbet használt STL-beli osztály, amely egységbe zárja a C-beli karakterláncot. Ez az új osztály azért előnyös, mert a régi változatával ellentétben kezeli a túlindexelést, elrejti a fizikai ábrázolást, valamint sokkal egyszerűbb és intuitívabb a használata. Valójában a *string* a *basic_string* osztálysablon egy *char*-ra példányosított változata:

```
typedef basic_string<char> string;
```

A *basic_string* egy olyan általános sablonosztály, amelynek nem csak karakterlánc lehet az eleme, hanem más objektumok is. Ennek segítségével például nagyon könnyen meg lehet valósítani a lokalizációt, tehát az egyes nyelvekre, karakterkészletekre specializálást.
Nézzük meg a következő példakódot, amely a sztringek létrehozására mutat néhány módszert:

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

int main() {
      string s1; // üres string
      string s2("valami"); // konstruktorban megadott kezdőérték
      string s3 = "valami mas"; // copy constructor
      string s4(s3); // copy constructor
      cout << s1 << endl << s2 << endl << s3 << endl << s4 << endl;
      return 0;
}
```

A futás eredménye:

```
valami
valami mas
valami mas
```

Lehetőség van továbbá arra is, hogy egy sztring részsztringjét adjuk értékül, vagy azzal inicializáljunk:

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

int main() {
	string s1("valamilyen szoveg");
	string s2(s1, 0, 6); // első 6 karakter
	cout << s2 << endl;
	string s3(s1, 4, 6); // 6 karakter a 4. pozíciótól (5. karaktertől)
kezdve
	cout << s3 << endl;
	string s4 = s1.substr(3, 7); // 7 karakter a 3.-tól
	cout << s4 << endl;
	return 0;
}
```

A futás eredménye:

```
valami
milyen
amilyen
```

Iterátorokat is létre lehet hozni, amelyek segítségével egyszerűen be lehet járni a sztringet. Az iterátor ebben az esetben olyan osztály, amely a karaktereket tároló tömb bejárását biztosítja. Az STL-ben a tároló kezdetét és végét a *begin* és az *end* metódusokkal lehet lekérni a bejáró számára.

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

int main() {
	string s1("valami");
	string::const_iterator it1 = s1.begin(); // iterátor s1 első betűjén
	string:: const_iterator it2 = s1.end();   // iterátor s1 „utolsó
utáni” betűjén
	++it1;       // növeljük az iterátort, azaz átlépünk a következő
betűre
	--it2;       // eggyel visszaléptetjük az iterátort
	string s2(it1,it2); // új sztring aminek tartalma az it1-től it2-ig
	tart
	cout << s2 << endl;
	for (it1 = s1.begin(); it1 != s1.end(); ++it1)
		cout << *it1; // kiírjuk az aktuális karaktert
	cout << endl;
	return 0;
}
```

A futás eredménye:

```
alam
valami
```

A *string* fontos tulajdonsága, hogy képes önmagát átméretezni, vagyis beállítani a kapacitását. A *kapacitás* azt jelenti, hogy mennyi helyet foglalt le a program az objektumnak. Például sztring konkatenációk sorozata esetén hasznos ez az információ, amikor is általában jó gyakorlat előre lefoglalni egy nagyobb szelet memóriát, ami által rengeteg átméretezés és ez által memóriamásolási művelet spórolható meg. A következő példa a méret és kapacitás közötti különbségre mutat rá:

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

int main() {
      string s1("valami");
      cout << s1 << endl;
      cout << "meret = " << s1.size() << endl;
      cout << "kapacitas = " << s1.capacity() << endl;
      s1.insert(0, "meg "); // 0. pozícióra beillesztünk
      cout << s1 << endl;
      cout << "meret = " << s1.size() << endl;
      cout << "kapacitas = " << s1.capacity() << endl;
      s1.reserve(500); // 500 karaktert kér lefoglalni
      s1.append(" es valami"); // hozzáfűzés
      cout << s1 << endl;
      cout << "meret = " << s1.size() << endl;
      cout << "kapacitas = " << s1.capacity() << endl;
      return 0;
}
```

A futás eredménye:

```
valami
meret = 6
kapacitas = 15
meg valami
meret = 10
kapacitas = 15
meg valami es valami
meret: 20
kapacitas = 511
```

Az előbb bemutatott műveleteken kívül a *string* képes megkeresni (*find*) és kicserélni (*replace*) egy szövegrészletet a karakterláncban. Ez a két tagfüggvény használatára mutat példát az alábbi *csereMind* függvény, amely az *s* sztringben található összes *mit* részsztringet lecseréli a *mire* sztringre:

```
string& csereMind(string& s, const string& mit, const string& mire) {
      size_t indul = 0;
      size_t talalt;
      while ((talalt = s.find(mit, indul)) != string::npos) {
```

```
            s.replace(talalt, mit.size(), mire);
            indul = talalt + mire.size();
      }
      return s;
}
```

A *size_t* egy előjel nélküli egész típusnak feleltethető meg, az *npos* pedig a *size_t* értékét túllépő értéknek tekinthető, ami a következőképpen van definiálva:

```
static const size_t npos = -1;
```

Ezeken túl az operátorok is meg vannak valósítva a string műveleteinek megfelelően, tehát például a konkatenáció elvégezhető a + és += operátorral, a lexikografikus összehasonlítás pedig az ==, !=, <, >, stb. operátorokkal, sőt a [] operátorral hivatkozni lehet a sztring egyes karaktereire is. A következő program sztring konkatenációra (+ operátor használatára) mutat példát:

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

int main() {
      string s1("egy ");
      string s2("meg ");
      string s3("az ketto");
      s1 = s1 + s2 + s1;
      s1 += s3;
      cout << s1 << endl;
      return 0;
}
```

A futás eredménye:

```
egy meg egy az ketto
```

A string osztály támogatja a különböző karakterkészleteket is. Sőt, akár bármilyen, megfelelő jellemzőkkel felruházott objektum is lehet karakter. Egy karaktertípus jellemzőit a hozzá tartozó *char_traits* osztály írja le, amely a

```
template<class charT> struct char_traits {/*...*/};
```

sablon specializációja. Minden *char_traits* az *std* névtérben szerepel, és a szabványos változatok a <*string*> header fájlból érhetők el. Az általános *char_traits* osztály egyetlen jellemzőt sem tartalmaz, jellemzőkkel csak az egyes karaktertípusokhoz készített változatok rendelkeznek. A *basic_string*-hez használt karaktertípusnak rendelkeznie kell egy *char_traits* specializációval.
A *basic_string* sablon az *std* névtérből, a <*string*> header fájlon keresztül érhető el. A *basic_string* sablon deklarációja a következőképpen néz ki:

```
template<class charT,
   class Traits = char_traits<charT>,
   class Allocator = allocator<charT> >
class basic_string {
   /*...*/
};
```

 © Ferenc Rudolf, SzTE

Itt az első paraméter a karakter típusát definiálja, a második a jellemzőket, a harmadik pedig egy memóriafoglalást definiáló osztályt. Az utóbbit ritkán szokás megadni.
A leggyakoribb karakterlánc típusok pedig *typedef* segítségével hozhatók létre:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

Saját sztring osztály

A következőkben egy saját string osztály készítését mutatjuk be, amely nem érzékeny a kis- és nagybetűkre, valamint kihasználja a *basic_string* és *char_traits* lehetőségeit. Evégett az osztályunkat a *char_traits* sablonosztályból származtatjuk, így az ő általa deklarált érintett függvényeknek is elkészítjük a saját felüldefiniált megvalósítását, amit majd a *basic_string* tud használni a karakterműveletekhez. Ezek a függvények az *eq, lt* és compare.
A következő kódrészletben tehát a saját *traits* osztályunk, az *ichar_traits* kódja látható, majd utána az *istring* definiálása.

```
#ifndef ICHAR_TRAITS_H
#define ICHAR_TRAITS_H

#include <iostream>
#include <string>

using std::char_traits;
using std::basic_string;
using std::ostream;

struct ichar_traits : char_traits<char> {

	// Két karakter egyenlőségét vizsgálja.
	static bool eq(char c1st, char c2nd) {
		// nagybetűsítve hasonlítjuk össze
		return toupper(c1st) == toupper(c2nd);
	}

	// Két karakter sorrendbeli összehasonlítása.
	static bool lt(char c1st, char c2nd) {
		// nagybetűsítve hasonlítjuk össze
		return toupper(c1st) < toupper(c2nd);
	}

	// Két karaktersorozat összehasonlítása.
	/* 0-val tér vissza, ha mind a kettő nulla vagy a két karaktersorozat
	legfeljebb első n karaktere megegyezik, -1-gyel ha az első paraméter
	kisebb mint a második, vagy nulla, de a második nem nulla, illetve 1-
	gyel ha a második kisebb mint az első, vagy nulla, de az első nem
	nulla.*/
	static int compare(const char* str1, const char* str2, size_t n) {
		for (size_t i = 0; i < n; i++) {
			if (*str1 == 0 && *str2 == 0) return 0;
			else if (*str1 == 0) return -1;
			else if (*str2 == 0) return 1;


			// a példa kedvéért most kisbetűsítve hasonlítjuk össze
			else if (tolower(*str1) < tolower(*str2)) return -1;
```

```
			else if (tolower(*str1) > tolower(*str2)) return 1;
			str1++;
			str2++;
		}
		return 0;
	}
};

// definiáljuk a saját string típusunkat
/* az istring char-t fog használni karakterként, és a sablonnak megadjuk
paraméterként a fent definiált ichar_traits osztályunkat, amit a
basic_string használni fog a műveletekhez. */
typedef basic_string<char, ichar_traits> istring;

// megvalósítjuk az adatfolyamba beillesztő (insertion) operátort
inline ostream& operator<<(ostream& os, const istring& s) {
	return os << std::string(s.c_str(), s.length());
}
#endif
```

A különböző traits-eket tartalmazó template-ek, pl. a string és istring nem keverhetőek műveletekben.
Hozzunk létre néhány saját string objektumot a következő *main* függvény megvalósítással:

```
#include "ichar_traits.h"
using namespace std;

int main() {
	istring first = "tHis";
	istring second = "ThIS";
	cout << first << endl;
	cout << second << endl;
	cout << first.compare(second) << endl; // összehasonlítjuk a két
istringet
	cout << (first != second) << endl; // összehasonlítjuk a két
istringet
	return 0;
}
```

A futtatás kimenete:

```
tHis
ThIS
0
0
```

 © Ferenc Rudolf, SzTE

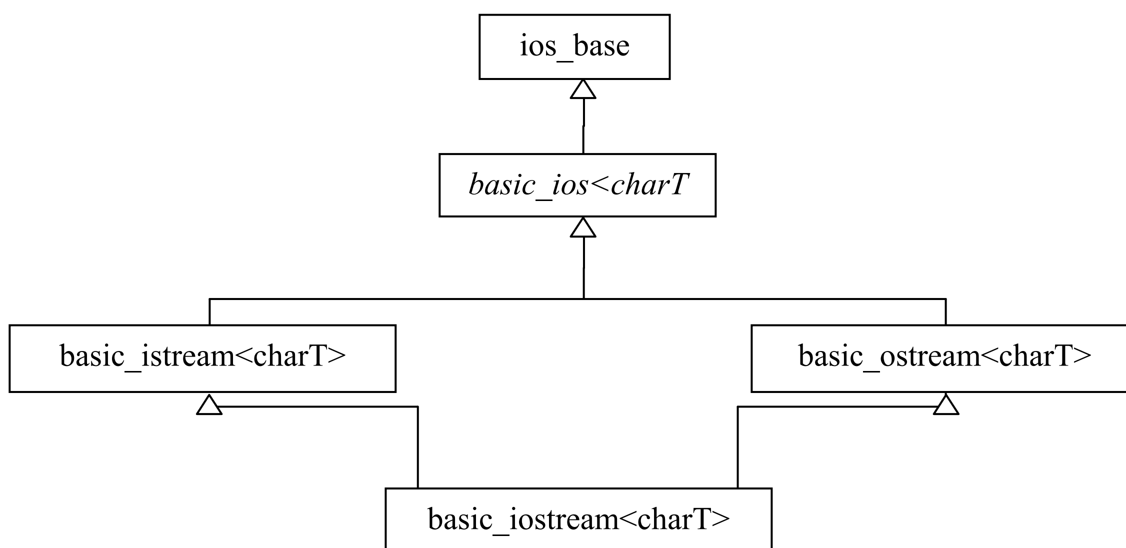
FOLYAMOK

A *folyam* szó alatt olyan információs csatornát értünk, amely képes az adatok közvetítésére két végpont között. A folyam működési elve lényegében egy valós folyam analógiáján alapul. A forrás szolgáltatja a folyamnak az adatokat, amelyek egymás után folyamatosan haladnak, egészen addig, amíg a forrás ki nem apad, azaz a folyam le nem zárul, véget nem ér. A nyelő oldalán az adatok folyamatosan érkeznek, és csak a folyam vége az a tény, amelyet a vevő érzékelni képes. Az alap analógián kívül érdemes megjegyezni, hogy léteznek olyan folyamok is, amelyek pozícionálhatóak, és ezzel kissé megváltoztatják a folyamokról megalkotott képet. Ezen kívül érdemes kiemelni még azt a tényt, hogy ami bekerül a folyamba, az nem azonnal jelenik meg a folyam másik végén, mivel a folyam általában elraktároz valamennyi adatot, amely hasznos lehet speciális műveletek elvégzésekor, illetve ezzel megnöveljük a folyamunk áteresztőképességét, és így a tömbösített műveletvégzés gyorsíthatja az adatok kezelését.

Adatfolyamok

Az *adatfolyam*ok *(stream)* olyan információs csatornák, amelyek sohasem telnek meg (tárkorlátosan) és az alapkonvenciójuk alapján akkor érnek véget, ha elfogy az adatforrás. Olyan adat közvetítő objektumot értünk alatta, amely karaktereket szállít és formatál. Minden I/O művelet elvégzése egységes interfészen keresztül valósul meg, ezáltal ugyanúgy kezelhetők az adatok, attól függetlenül, hogy a folyam célja a konzol, egy fájl, vagy a memória, mind olvasás, mind írás terén. A folyamok előnye abban rejlik, hogy nemcsak egyszerűbben és nagyobb biztonsággal kezelhetőek, mint a standard C könyvtár, hanem egyes mérések szerint hatékonyabbak is lehetnek. Az alábbiakban felsorolásra kerülnek az adatfolyam hierarchia szerkezetében megtalálható osztályok, azok öröklődési hierarchiáját pedig a következő ábra mutatja:

- *ios_base:* karakter típustól független műveletek,
- *basic_ios<charT>:* karakter típustól függő általános műveletek,
- *basic_istream<charT>:* bemenő adatok kezelése,
- *basic_ostream<charT>:* kimenő adatok kezelése,
- *basic_iostream<charT>:* ki/bemenő adatok kezelése.



A különböző *stream* típusok (*istream, ostream, iostream)* valójában sablon példányok. Például az istream típusdefiníciója a következő:

```
typedef basic_istream<char> istream;
```

Az STL implementációban ezen sablonoknak van egy második paramétere is, a *class traits*, amely a folyamban szállított adatok jellemvonásait írja le. Így a fenti istream típusdefiníció is a valóságban ennek megfelelően *typedef basic_istream<char, char_traits<char> > istream;*. De mivel a traits technika részletesen csak egy későbbi fejezet témája, és a folyamok hétköznapi használatához nincs is rá szükség, így most eltekintünk a *stream* sablonok második paraméterétől.

Két operátor minden beépített típusra ki van terjesztve: a << beszúró (*inserter*) operátor és >> kinyerő (*extract*) operátor. Ha saját osztályra szeretnénk használni ezeket az operátorokat, akkor meg kell valósítani az osztályhoz a << és >> operátorokat.
Háromféle standard I/O *stream* létezik: a *cin*, amely billentyűzetről olvas, a *cout,* ami a képernyőre ír, és a *cerr***,** ami szintén a képernyőre, mint hibakimenetre ír. Ezek a *stream*-ek természetesen csak alapértelmezésben vannak a megjelölt helyekre irányítva, és ezeket az alapértelmezéseket a program futtatója képes módosítani a program megfelelő paraméterezésével.
Az alábbi példakód a standard bemenetről kér be három különböző típusú értéket (egy egész számot, egy lebegőpontos számot és egy sztringet), majd ezeket kiírja a konzolra:

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

using namespace std;

int main() {
      int i;
      cin >> i;
      float f;
      cin >> f;
      string s;
      cin >> s;
```

 © Ferenc Rudolf, SzTE

```
	cout << "i: " << i << endl;
	cout << "f: " << f << endl;
	cout << "s: " << s << endl;
	return 0;
}
```

A program a „10 20.5 valami” input esetén a következő output-ot adja:

```
i: 10
f: 20.5
s: valami
```

A példakódban található *endl* egy speciális előre definiált módosító a folyam számára, amely képes a folyam manipulálására, konkrétan egy sortörést vált ki a folyamban. Ezekkel a módosítókkal a manipulátorok részben fogunk foglalkozni bővebben.

Saját adatfolyam operátorok

A << és >> operátoroknak minden beépített típusra található kiterjesztése, így a *stream*-ek képesek ezek használatára. Amennyiben saját típust kívánunk kiíratni egy *stream*-re, kiterjeszthetjük a fent említett operátorokat a saját típusra, és így egyszerűen lehet az adatainkat *stream*-eken kezelni. Ha saját osztályhoz szeretnénk megvalósítani a << és >> operátorokat, érdemes tudni, hogy ezek az operátorok szigorúan két paraméterrel rendelkeznek: az első paraméter egy nem konstans referencia a folyamra (bemenő folyam esetén *istream,* kimenő folyam esetén *ostream*), a második paraméter pedig bemenő folyam esetén referencia a saját típusunkra, kimenő folyam esetén konstans referencia a saját típusra. Így képesek vagyunk megvalósítani bármely típusra egyszerű és szabványos formában a kiíratási és beolvasási műveleteket. Mivel a visszatérési érték maga a folyam referencia, ezért megvalósítható segítségével a láncolás, mint azt már korábban is láthattuk (az első példánkban a << operátor után ismét használható az operátor más adatok kiíratására, ez lehetővé teszi akár különböző típusú elemek egymás után láncolt egyszerű kiíratását). Ahhoz, hogy a már bemutatott *stream*-eken végzett műveletek a szabványos módon működjenek, nem képezhetik a saját osztályunk részét az operátorok: az osztályunkon kívül kell a *stream* operátorait megvalósítani. Egy saját dátum osztály létrehozásával mutatjuk be a << és >> operátorok implementációját és használatát.

```
#include <iostream>
#include <iomanip>

class Date {
public:
	Date(int d, int m, int y) : day(d), month(m), year(y) {}
	int getYear() const {return year;}
	int getMonth() const {return month;}
	int getDay() const {return day;}
	friend std::ostream& operator<<(std::ostream&, const Date&);
	friend std::istream& operator>>(std::istream&, Date&);
private:
	int year, month, day;
};

std::ostream& operator<<(std::ostream& os, const Date& d) {
	os.fill('0');
	os << std::setw(2) << d.getDay() << '-'
```

```
            << std::setw(2) << d.getMonth() << '-'
            << std::setw(4) << d.getYear();
      return os;
}

std::istream& operator>>(std::istream& is, Date& d) {
      is >> d.day;
      char dash;
      is >> dash;
      if (dash != '-') /*...*/; // a hibakezelés csak jelképes
      is >> d.month;
      is >> dash;
      if (dash != '-') /*...*/; // a hibakezelés csak jelképes
      is >> d.year;
      return is;
}

int main() {
      Date d(1,5,2010);
      std::cout << d << std::endl;
      std::cin >> d;
      std::cout << d << std::endl;
      return 0;
}
```

A *Date* osztályban látható, hogy az operátorok deklarációja meg van jelölve a *friend* kulcsszóval, ezzel biztosítható, hogy az osztály privát részét is elérhessék az osztályon kívül szereplő, viszont ahhoz szorosan kapcsolódó saját operátoraink. Az *output stream operátor* (<<) implementációja a kitöltő karaktert a 0 karakterre állítja. Az egyes dátum tagokat fix szélességében írja ki, ehhez használjuk a *std::setw(int n)* függvényt. Így, amennyiben az egyes tagok rövidebbek (pl. első nap), az üres területek 0-val lesznek kitöltve. *Az input stream operátor* (>>) megvalósításában adunk példát egy összetett osztály adatainak bekérésére. Az elválasztó karakter a '–', azaz a *dash* karakter. A *hiba* kommenttel jelzett részek jelölik az input ellenőrzésért felelős, hibakezeléssel ellátandó kódok helyét. Látható, hogy egy ilyen *stream* implementáció használatával a *Date* objektumok kezelése sokkal kényelmesebbé válik, és ez megvalósítható tetszőleges saját típusra. A fenti példa kód futtatása a következő eredményt szolgáltatja az elkészített operátorok használatára:

Kiírt adat:
`01-05-2010`
Bekért adat:
`05-05-2010`
Kiírt adat:
`05-05-2010`

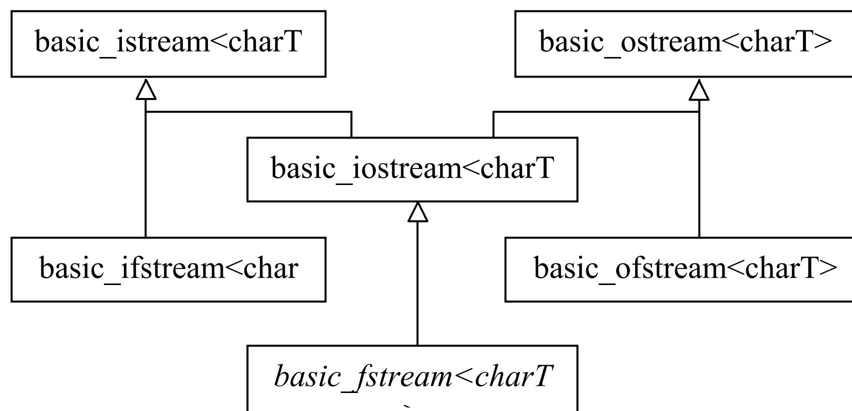
Megjegyezzük, hogy az egyes operátorok használhatóak függvényhívás formájában is függvény mivoltuk végett. Ebben az esetben például egy << operátor hívás a *cout*-ra a fent definiáltak alapján a következőképpen nézne ki a fenti *main* függvény utolsó soraként (sortörés nélkül):

```
operator<<(std::cout, d);
```

Fájlfolyamok

Tekintsük először a fájlfolyamok öröklődési hierarchiájának felépítését az alábbi ábrán, ahol a különböző fájlfolyamok a következő funkciókért felelősek:

- *basic_ifstream<charT>:* fájlból olvasás,
- *basic_ofstream<charT>:* fájlba írás,
- *basic_fstream<charT>:* fájlból olvasás/írás.



A különböző *file stream* osztályok (*ifstream, ofstream, fstream)* valójában sablon példányok. Például az *ifstream* típusdefiníciója a következő:

```
typedef basic_ifstream<char> ifstream;
```

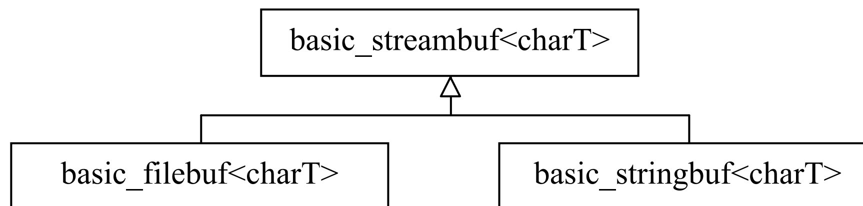
Az alábbi rövid példában szemléltetjük a fájlfolyamok egyszerű használatát az *ifstream* használatán keresztül:

```
#include <fstream>
#include <iostream>

using namespace std;

int main() {
      const int size = 100;
      char buff[size];
      ifstream in("main.cpp");
      while (in.getline(buff, size))
            cout << buff << endl;
      return 0;
}
```

A fenti kód önmagát írja ki eredményül (a main.cpp tartalmát) a jelenleg beállított *cout stream*-re (alapesetben a képernyőre), úgy, hogy soronként beolvassa egy *ifstream* segítségével a main.cpp fájl tartalmát.

Adatfolyam pufferezés

Miután az adatok bekerülnek a folyamba, nem azonnal kerülnek ki onnan, hanem a folyam megfelelő események bekövetkeztéig tárolja őket. Minden *stream* objektum rendelkezik egy mutatóval valamilyen *streambuf*-ra. Ezen keresztül a nyers adatfolyam elérhető formázás nélkül. Lekérhető a *rdbuf* segítségével, akár egyszerűen hozzá lehet kapcsolni egy másik *iostream* osztályhoz a << operátor segítségével. Ezek a pufferek végzik a folyamok adatainak köztes tárolását. A *streambuf* osztályok öröklődési hierarchiája látható a következő ábrán:




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

int main() {
	ifstream in("main.cpp");
	cout << in.rdbuf();
	return 0;
}
```

A fenti program ismét önmagát írja ki, nyers, formázatlan alakjában (a futtatás végeredménye maga a példa kód lesz). Itt nem soronként olvassuk be a fájl tartalmát, hanem egyszerre az *rdbuf* függvénnyel, amit aztán egyben ki is íratunk.

Keresés az adatfolyamban

Szükség esetén pozícionálhatunk is a folyam feldolgozása során az abszolút, illetve a relatív pozíció (fájl eleje: *ios::beg*, aktuális pozíció: *ios::cur* vagy vége: *ios::end*) megadásával:

```
istream& seekg(streampos pos);
istream& seekg(streamoff off, ios::seek_dir dir);
```

Nézzünk egy példát az aktuális pozíció megváltoztatására!

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

int main() {
	const int NUM = 5, LEN = 10;  // 5 rekord 10 hosszal
	char data[NUM][LEN] = {
		"elso",
		"masodik",
		"harmadik",
		"negyedik",
		"otodik"
	};
	//beallito cimke, binaris kiirast allit be
	ofstream out("proba.bin", ios::out|ios::binary);
```

 © Ferenc Rudolf, SzTE

```
	for (int i = 0; i < NUM; i++)
		out.write(data[i], LEN);
	out.close();
```

A példa binárisan kiírja a *data* tömbben eltárolt elemeket a proba.bin fájlba *LEN* méretű rekordok formájában.
Nézzük meg, miként lehet visszaolvasni a már kiírt adatokat binárisan, a fentebb meghatározott fájlból LEN rekord mérettel. A program a beolvasott adatokat eredeti helyükre írja vissza a tömbben.

```
	// visszaolvaso beallitasa, es letrehozasa
	ifstream in("proba.bin", ios::in|ios::binary);
	in.read(data[0], LEN);  //  az  elso  elem  beolvasasa  data[0]-ba,  LEN
	hosszal
	cout << data[0] << endl;  // elso
	in.seekg(-LEN,  ios::end);  //  a  vegetol  10  karakterrel  pozicional
	vissza
	in.read(data[1], LEN);
	cout << data[1] << endl;      // otodik
	in.seekg(3 * LEN);            // abszolut cimzes
	in.read(data[2], LEN);
	cout << data[2] << endl;      // negyedik
	in.seekg(-LEN  *  2,  ios::cur);    //  relativ  cimzes  az  aktualis
	poziciotol
	in.read(data[3], LEN);
	cout << data[3] << endl;      // harmadik
	in.seekg(LEN, ios::beg);      // relativ cimzes a stream kezdetetol
	in.read(data[4], LEN);
	cout << data[4] << endl;      // masodik
	return 0;
}
```

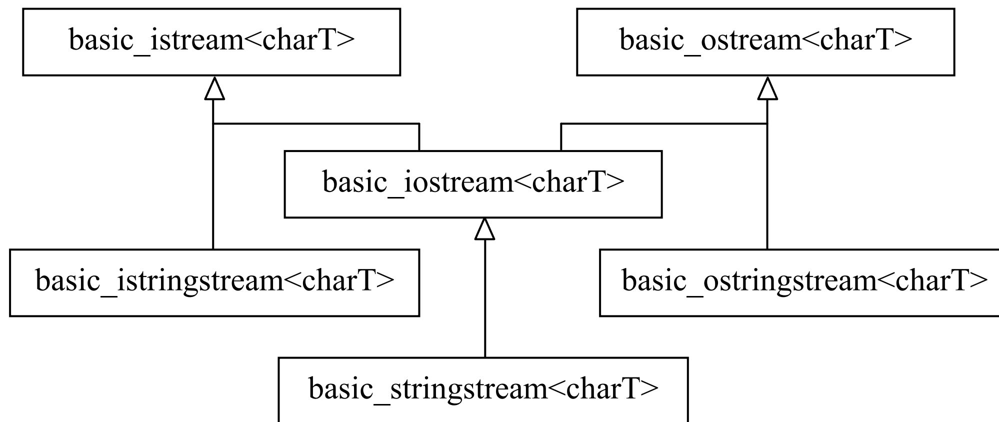
A *seekg* függvény a fájl aktuális pozícióját áthelyezi az első paraméterben (*offset*) megadott bájttal, a második paraméterben (*seeking direction*) megadott irányban. A *seekg* egyparaméteres változata felel meg az abszolút pozíció megadásának, ahol csak a pozíciót kell megadni paraméterben.
A program futtatása után a következő eredményt kapjuk:

```
elso
otodik
negyedik
harmadik
masodik
```

Sztring folyamok

A sztring folyam öröklődési hierarchiája megegyezik a fájlfolyam öröklődési hierarchiájának felépítésével. A különböző sztring folyamok a következő funkciókért felelősek:

- *basic_istringstream<charT>:* memóriából olvasás,
- *basic_ostringstream<charT>*: memóriába írás,
- *basic_stringstream<charT>*: memória írás/olvasás.



A különböző stream osztályok (*istringstream, ostringstream, stringstream)* valójában sablon példányok, mint amint azt már láthattuk a többi *stream* esetén is. Például az *istringstream* típusdefiníciója a következő:

```
typedef basic_istringstream<char> istringstream;
```

A következő példában nézzük meg egy egyszerű, bemenő string folyam használatát, és annak alapvető kezelési formáit:

```
#include <iostream>
#include <sstream>

using namespace std;

int main() {
	istringstream iss("47 1.414 Ez egy teszt");
	int i;
	double f;
	iss >> i >> f;
	cout << i << endl << f << endl;
	string s;
	iss >> s;
	cout << s << endl;
	cout << iss.rdbuf() << endl; //kiolvassa a stream teljes, megmaradt
	tartalmat
	return 0;
}
```

A példa az *istringstream* használatát mutatja be. A fenti példa az alábbi kimenetet adja:

```
47
1.414
Ez
 egy teszt
```

Most tekintsünk egy másik példát a kimenő string folyamok használatára:

```
#include <iostream>
#include <sstream>

using namespace std;
```

```
int main() {
      cout << "Kerek egy int-et, egy float-ot es egy string-et: " << endl;
      int i;
      float f;
      cin >> i >> f;  // i és f feltoltese
      cin >> ws;  // elválasztó (whitespace) karakterek kihagyása
      string maradek;
      // Az aktualis sor aktualis poziciojatol bekeri annak a vegeig levo
tartalmat
      getline(cin, maradek);
      ostringstream os;  // letrehozza es a kovetkezo sorokkal feltolti az
os-t
      os << "integer = " << i << endl;
      os << "float = " << f << endl;
      os << "string = " << maradek << endl;
      // a stringbuf masolatat adja string-kent a str() metódus
      string result = os.str();
      cout << result << endl;
      return 0;
}
```

A program beolvassa a kapott egészet, lebegőpontos számot és sztringet, majd az *os* kimenő sztring folyamra küldi ezeket. Végül az *str* függvény segítségével *string* típusúvá konvertálja *os* tartalmát és kiírja a standard kimenetre.

A fenti példa a következő kimenetet produkálja:

```
Kerek egy int-et, egy float-ot es egy string-et:
19 3.14    vege van
integer = 19
float = 3.14
string = vege van
```

Kimenő folyam formázása

Az *ios* (a *basic_ios char*–ra példányosított változata) különböző jelzőket (*flags*) tartalmaz, amik befolyásolják a folyam formázását.
Ilyen jelzők az alábbiak:

- *ios::skipws*: elválasztó karakterek kihagyása,
- *ios::showbase*: számrendszer megjelenítése,
- *ios::showpoint*: tizedespont lebegőpontos számoknál,
- *ios::uppercase*: nagybetűs A-F és E,
- *ios::showpos*: írja ki a „+” jelet pozitív számoknál.

Az alábbi függvények segítségével kérhetők le, illetve állíthatók be ezek a jelzők:

- *fmtflags ios::flags();*
- *fmtflags ios::flags(fmtflags newflags);*
- *fmtflags ios::setf(fmtflags ored_flag);*
- *fmtflags ios::unsetf(fmtflags clear_flag);*

A *formátum mezők* arra szolgálnak, hogy a kimenő adat külalakját megváltoztassák. Ez egy nagyon előnyös funkció, mivel sokkal egyszerűbbé teszi a folyamokban szereplő számok

kiírását, ezzel sok terhet levéve a programozóválláról. Az egyes mező értékek logikus módon kizárják egymást, azaz egyszerre csak egy lehet beállítva:

- *ios::basefield* (ios::dec / ios::hex / ios::oct): melyik számrendszerben ábrázolja,
- *ios::floatfield* (ios::scientific / ios::fixed): számformátum,
- *ios::adjustfield* (ios::left / ios::right / ios::internal): igazítás.

A fenti mezők a *fmtflags ios::setf(fmtflags bits, fmtflags field)* függvénnyel kérhetőek le, illetve állíthatóak be.

A fent bemutattakon kívül beállíthatjuk még például a szélességet, a kitöltő karaktert, illetve a pontosságot. Ezek függvényekkel kérhetőek le, illetve állíthatóak be, a következő formában:

- *int ios::width():* aktuális szélesség lekérése,
- *int ios::width(int n):* szélesség beállítása,
- *int ios::fill():* aktuális kitöltő karakter lekérése,
- *int ios::fill(int n):* kitöltő karakter beállítása,
- *int ios::precision():* aktuális lebegőpontos pontosság lekérése,
- *int ios::precision(int n):* lebegőpontos pontosság beállítása.

A fenti jelzők és függvények használatára mutat példát az alábbi kód:

```
#include <iostream>
#include <sstream>

using namespace std;

int main() {
	ostringstream os;
	os << (float)10 << endl;
	// tizedes pont, es a pozitiv elojel megjelenitese
	os.setf(ios::showpos|ios::showpoint);
	os << (float)10 << endl;
	// beallitja az abrazolasi pontossagot
	os.precision(5);
	os << (float)10 << endl;
	os << 0x10 << endl;
	// modositja az abrazolas alapjaul szolgalo szamrendszert.
	os.setf(ios::hex,ios::basefield);
	os << 0x10 << endl;
	string result = os.str();
	cout << result << endl;
	return 0;
}
```

A program először kiíratja a 10-es számot, mint lebegőpontos számot egy *ostringstream*-be. Ez után beállítja, hogy pozitív szám esetén a '+' jel is kiírásra kerüljön, továbbá a tizedespont is megjelenjen. Majd a kiírás pontosságát 5-re állítja. Kiíratja a 10-es hexadecimális számot decimális számrendszerben, majd átállítja a kiírást hexadecimális számrendszerré és úgy is kiíratja a 10-es hexadecimális számot. Végül az egészet sztringgé konvertálja és kiküldi a standard outputra.
A futtatás végeredménye:

```
10
+10.0000
```

```
+10.000
+16
10
```

Manipulátorok

Függvényhívások helyett manipulátornak nevezett speciális függvények is alkalmazhatók a folyam formázására. Ezek előnye, hogy használhatók a << és a >> operátorokkal közös kifejezésben, ezen kívül javítják a kód olvashatóságát, a kód külalakját szebbé és letisztultabbá teszik. A manipulátorok két típusra bonthatók: a paraméterrel rendelkezőkre és a paraméter nélküliekre. A paraméterekkel rendelkezőkre a továbbiakban *effektor*ként fogunk hivatkozni. Először tekintsük át a paraméter nélküli manipulátorokat:

- *showbase, noshowbase*: számrendszer megjelenítése,
- *showpos, noshowpos*: írja/ne írja ki a „+” jelet pozitív számoknál,
- *uppercase, nouppercase*: nagybetűs/kisbetűs A-F és E,
- *showpoint, noshowpoint*: tizedespont lebegőpontos számoknál,
- *skipws, noskipws*: elválasztó karakterek kihagyása/figyelembe vétele,
- *left, right, internal*: igazítás,
- *scientific, fixed:* számformátum.

Paraméterrel rendelkező manipulátorok (*effektorok*):

- *setioflags(fmtflags n)*: n-ben levő jelzőket állítja be,
- *resetioflags(fmtflags n)*: töröl minden jelzőt, majd az *n*-ben levő jelzőket állítja be,
- *setbase(base n)*: melyik számrendszerben ábrázolja,
- *setfill(char n)*: kitöltő karakter beállítása,
- *setprecision(int n)*: pontosság beállítása,
- *setw(int n)*: szélesség beállítása.

Az alábbiakban nézzünk egy példát a manipulátorok használatára:

```
#include <iostream>
#include <iomanip>
#include <sstream>

using namespace std;

int main() {
      ostringstream os;
      os << (float)10 << endl;
      os << showpos << showpoint << (float)10 << endl;
      os << setprecision(5) << (float)10 << endl;
      os << 0x10 << endl;
      os << hex << 0x10 << endl;
      string result = os.str();
      cout << result << endl;
      return 0;
}
```

Amennyiben a fenti példát összevetjük a jelzőket beállító függvényeket használó példával, látható, hogy a forráskód letisztultabb, egyszerűbb képet mutat. A háttérben ennek ellenére a műveletek a mezőkkel való manipuláció megvalósulását eredményezik. Látható hogy ezzel

lényegében az írásmód leegyszerűsödik. A forráskód azonos eredményt szolgáltat a mező-beállításokat használó kóddal is:

```
10
+10.0000
+10.000
+16
10
```

Saját manipulátorok

Tekintsük a következő egyszerű saját paraméter nélküli példa manipulátort:

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

ostream& nl(ostream& os) {
	return os << '\n';
}

int main() {
	cout << "sortores" << nl << "minden" << nl
	<< "szo"           << nl << "kozott" << endl;
	return 0;
}
```

A saját készítésű *nl* nevű manipulátor egy sortörést fűz a kimenet folyamhoz. A kód futtatásának végeredménye:

```
sortores
minden
szo
kozott
```

A példa alapján látható, hogy a *stream*ek viselkedésének módosítására készíthetünk saját manipulátorokat, amelyek elvégezhetnek tetszőleges műveletet a bemeneten, és beállíthatnak tetszőleges formázást a *stream*-en.

A ... << *cout* << *nl* << ... kifejezésben az nl függvény nevének leírása tulajdonképpen a függvény címét jelenti. Így ahhoz, hogy ez a kifejezés leforduljon, lennie kell az STL implementációban egy olyan *operator*<< megvalósításnak, mely képes függvénypointert fogadni. (A *cout* << *nl* hívás írható lenne úgy is, hogy *cout.operator*<<*(nl)*.) Ez az operator az *ostream* fájlban található a *basic_ostream* tagfüggvényeként. A forráskódja leegyszerűsítve a következő:

```
template<class _Elem, class _Traits>
class basic_ostream     : virtual public basic_ios<_Elem, _Traits> {
public:
	typedef basic_ostream<_Elem, _Traits> _Myt;
	/*...*/
	_Myt& operator<<(_Myt& (*_Pfn)(_Myt&)) {
		return (*_Pfn)(*this);  // callback a manipulátorunkra
	}
	/*...*/
};
```

Paraméterekkel rendelkező manipulátorokat, vagyis effektorokat másképp kell készíteni. Példaként nézzük meg az alábbi forráskódot:

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

class prefix {
	string str;
public:
	prefix(const string& s, int width) : str(s,0,width) {}
	friend ostream& operator<<(ostream& os, const prefix& fw) {
		return os << fw.str;
	}
};

int main() {
	string s = "Teszt";
	for (int i = s.size(); i >= 0; --i)
		cout << prefix(s,i) << endl;
	return 0;
}
```

A példa definiál egy *prefix* nevű effektort, ami két paramétert vár, az első paraméterben kapott sztring első x karakterét küldi a kimenetre, ahol x a második paraméter.
A feni kód futtatásának végeredménye a következő:

```
Teszt
Tesz
Tes
Te
T
```

A *cout << prefix(s,i)* kifejezés úgy is írható, hogy *operator<<(cout, prefix(s,i))*. Így egyértelműbben látszik, hogy a *prefix* osztályunkhoz megvalósított *operator<<* függvény hívódik meg. A függvényhívás előtt a paraméterek kiértékelődnek, ami esetünkben azt jelenti, hogy a *prefix* osztályból létrejön egy ideiglenes objektum, lefut a konstruktora, mely beállítja az *str* adattagot úgy, hogy az az átadott *s* sztring *i* hosszúságú prefixszét tartalmazza. Ezek után az operator hívásra így tekinthetünk: *operator<<(cout, tmp)*, ahol *tmp* az ideiglenes *prefix* objektum. Az *operator<<* lefutása során kiírja az *str* adattag tartalmát a *cout*-ba.
Végezetül tekintsük át még egyszer a bemutatott technikákat. A *stream*-ek bemutatása során látható volt, hogy a *stream* osztályok egyszerű, egységesített módszert kínálnak adatok mozgatására különböző adatforrások és adatnyelők között. Az egységes kezelési mód lehetővé teszi, hogy egy *stream* rendszer használatának elsajátításával lényegében az összes többit is megismerjük. A hatékonyság növelésének érdekében a kimeneti *stream*-eken automatikus formázási lehetőséget biztosítanak a jelzők, amelyekkel könnyedén alakíthatjuk ki az kimenetünk kinézetét az általunk kívánt formára, a *stream* tulajdonságainak beállításával. Ezen kívül a fejezet végén megismertük a manipulátorokat és a paraméterezhető effektorokat, amelyek a kimenet formázását teszik még egyszerűbbé azáltal, hogy lehetőséget biztosítanak speciális operátor kiterjesztések segítségével függvénypointerek és osztályok használatára a kimeneti adatok megadása mellett. Ezáltal a formázás közvetlenül az adat mellett jelenik meg, tömörebb és átláthatóbb írásmódot biztosítva a programozó számára.

GENERIKUS PROGRAMOZÁSI IDIÓMÁK

Ebben a fejezetben a generikus programozási idiómákról lesz szó. A fejezetben három idióma fog tárgyalásra kerülni, ezek sorban a *Traits* technika, a *Policy* és a *Curiously recurring template pattern*. Mindhárom idióma megértése segítő példákon keresztül lesz bemutatva.

Traits (jellemvonások)

Először is a *Traits* technika kerül bemutatásra, melyet Nathan Myers dolgozott ki. Ezzel a módszerrel a típusfüggő deklarációkat tudjuk egybecsomagolni, típusokat és értékeket lehet egymáshoz rendelni különböző összefüggésekben. Így a kód átláthatóbb, karbantarthatóbb lesz, a későbbiekben könnyebb lesz a kód módosítása.
Nézzünk egy példát a *Traits* technikára. A példában egy rajzfilmet fogunk illusztrálni, melyben szereplők szerepelhetnek. Az adott szereplő pedig kétféle enni- vagy innivalót fogyaszthat. Először is definiálunk két italt (víz és tej) és két ennivalót (méz és süti).

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

struct Viz {
      friend ostream& operator<<(ostream& os, const Viz&) {return os <<
"viz";}
};

struct Tej {
      friend ostream& operator<<(ostream& os, const Tej&) {return os <<
"tej";}
};

struct Mez {
      friend ostream& operator<<(ostream& os, const Mez&) {return os <<
"mez";}
};

struct Suti {
      friend ostream& operator<<(ostream& os, const Suti&) {return os <<
"suti";}
};
```

Ezek lesznek az elem osztályok, a szereplők *Traits*-ei (jellemvonásai), melyekkel leírható, hogy az adott szereplő mit szeret fogyasztani. Látható, hogy a struktúrák által definiált működés hasonló. Mindegyik struktúrában felüldefiniálásra került a << operátor, ami egy közös interfészt biztosít hozzájuk. Az interfész természetesen más is lehet. Fontos észrevenni, hogy az elem osztályok teljesen függetlenek egymástól, nem hivatkoznak egymásra.
Most definiáljuk a szereplőket:

```
struct Micimacko {
      friend ostream& operator<<(ostream& os, const Micimacko&)
      {return os << "Micimacko";}
};

struct RobertGida {
```


© Ferenc Rudolf, SzTE

```
        friend ostream& operator<<(ostream& os, const RobertGida&)
        {return os << "Robert Gida";}
};
```

Látható, hogy ezek a struktúrák is hasonlítanak egymásra, szintén definiálnak egy közös interfészt (*operator*<<), és függetlenek egymástól és az elem osztályoktól is. Most pedig megadjuk az elsődleges *Traits* sablont:

```
template<class Szereplo> class SzereploTraits;
```

Az elsődleges *Traits* sablont csak a sablon általános esetének definiálására használjuk, mivel ezt fogjuk tovább specializálni további sablonokká. A jelen példában lévő *SzereploTraits* sablonban fogjuk megadni a jellemvonásokat, hogy az egyes szereplők mit ehetnek és ihatnak. Ezt úgy tehetjük meg, hogy az egyes szereplőkre specializáljuk a *SzereploTraits* sablont. Ilyenkor a megfelelő sablonparamétert elhagyjuk a definícióból és az osztály neve után jelezzük az adott szereplőre való specializációt. Specializáljuk a *SzereploTraits* sablont *Micimacko*-ra és *RobertGida*-ra:

```
template<>
class SzereploTraits<Micimacko> {
public:
        typedef Viz ital_tipus;
        typedef Mez uzsonna_tipus;
};

template<>
class SzereploTraits<RobertGida> {
public:
        typedef Tej ital_tipus;
        typedef Suti uzsonna_tipus;
};
```

A specializált osztályokban definiáltunk két típust, az *ital_tipust* és az *uzsonna_tipust*, mely típusok eltérnek az egyes specializált osztályokban. Így megadható, hogy Micimackó vizet és mézet, Róbert Gida pedig tejet és süteményt szeret fogyasztani. Ezek a *Traits* osztályok jelentik az egyetlen kapcsolatot az elem és szereplő osztályok között. Most pedig megadjuk a *Rajzfilm* osztálysablont:

```
template <class Szereplo, class Traits = SzereploTraits<Szereplo> >
class Rajzfilm {
        typedef typename Traits::ital_tipus ital_tipus;
        typedef typename Traits::uzsonna_tipus uzsonna_tipus;
        ital_tipus ital;
        uzsonna_tipus uzsonna;
        Szereplo szereplo;
public:
        void szerepel() {
                cout << "Amit " << szereplo << " eszik az: "
                << ital << " es " << uzsonna << endl;
        }
};
```

A *Rajzfilm* osztálysablon két paraméterrel rendelkezik, az elsőben megadható, hogy melyik szereplő fog szerepelni a mesében, a másodikban pedig magát a *Traits*-et adhatjuk meg,

amelyben definiáltuk, hogy az adott szereplő mit eszik és iszik. A második paramétert kezdő (*default*) értékkel láttuk el, mivel ha *Micimackóval* vagy *Róbert Gidával* paraméterezzük fel a sablont, akkor ezeknek az osztályoknak a *SzereploTraits* specializációja megfelelő lesz *Traits* paraméternek. Az osztályon belül definiálunk *ital_tipus* és *uzsonna_tipus* típusokat. Ezeknek a típusoknak a típusát a paraméterként átadott *Traits* osztályból fogjuk átvenni, így a típusok adottak lesznek lokálisan is. Ezért az adott *Traits* osztálynak rendelkeznie kell *ital_tipus* és *uzsonna_tipus* típus definíciókkal, különben a program nem fordulna le. A *typedef* kulcsszó után ki kell írni a *typename* kulcsszót, mivel ezzel jelezzük a fordítónak, hogy amit a *Traits* osztályból használunk, azok típusok. Ha ezt nem tennénk meg, akkor a fordító nem tudná, hogy most típust vagy valami mást (adattagot, metódust) szeretnénk elérni a *Traits* osztályból. Létrehozunk egy-egy *Szereplo*, *ital_tipus* és *uzsonna_tipus* adattagot, amiket a *szerepel* függvényben fogunk használni. A *szereplo* objektum típusát a paraméterként átadott típus fogja meghatározni, tehát ha *Micimacko*-t adtunk át, akkor a *szereplo* objektum valójában *Micimacko* típusú lesz. Ugyanígy, ha az első sablonparaméter *Micimacko* volt, akkor a második paraméter annak *Traits* osztálya lesz, a *SzereploTraits<Micimacko>*, mely az *ital_tipus*-t *Viz*-ként, az *uszonna_tipus*-t pedig *Mez*-ként definiálja.
Mivel a *Traits*-ekre nincs típus megkötés, ezért megtehetjük azt is, hogy a *Rajzfilm* sablonnak nem az adott szereplőre specializált *Traits* osztályát használjuk. Ebben az esetben egy olyan osztályt kell megadnunk, amiben szerepelnek azok a típusnevek, amelyekre a *Rajzfilm* osztály hivatkozik. Tehát egy olyan osztályt kell átadnunk, melyben szerepel *ital_tipus* és *uzsonna_tipus* típus definíció, sőt ennek az osztálynak nem is kell sablonnak lennie. Így el lehet érni például, hogy *Micimackó* víz helyett is mézet fogyasszon. Ehhez viszont az kell, hogy a *Rajzfilm* második paraméterét is megadjuk példányosításkor.

```
class EgyebTraits {
public:
	typedef Mez ital_tipus;
	typedef Mez uzsonna_tipus;
};
```

Hozzuk létre a következő *main* függvény megvalósítással Róbert Gida és Micimackó főszereplésével Rajzfilm objektumokat, és nézzük meg, mit esznek és isznak:

```
int main() {
	Rajzfilm<RobertGida> rf1;
	rf1.szerepel();
	Rajzfilm<Micimacko> rf2;
	rf2.szerepel();
	Rajzfilm<Micimacko, EgyebTraits> rf3;
	rf3.szerepel();
	return 0;
}
```

A *main* függvény első sorában a *Rajzfilm* sablon példányosítása történik *RobertGida* típusra. Ilyenkor a fordító készít egy teljesen új osztályt, oly módon, hogy a *Rajzfilm* osztálysablont „lemásolja”, a *Szereplo* sablonparaméter helyére pedig a *RobertGida* típust helyettesíti. Így a *Rajzfilm* osztályba már *RobertGida* objektum fog létrejönni a *Szereplo* helyén. A fordító látja, hogy a *Rajzfilm* második paramétere megint csak egy sablon, ezért azt is példányosítja: *RobertGida*-ra specializált *SzereploTraits* osztály jön létre. A lefordított *Rajzfilm* osztályon belül a *Traits* osztály már konkrét osztályt fog jelenteni, ami rendelkezik a megfelelő típus definíciókkal. Ha létrejött a *Rajzfilm* osztályból a *RobertGida*-val példányosított objektum,

akkor annak már meg lehet hívni a *szerepel* metódusát. Mivel az osztály *RobertGida*-val lett példányosítva, a *szerepel* metódusban már a konkrét *RobertGida* objektum << operátora fog meghívódni. A *Micimackóval* való példányosítás hasonlóan működik. A harmadik példányosításnál paraméterezzük a *Rajzfilm* sablon osztály második paraméterét is, így megadható Micimackótól teljesen független *Traits* is. A *main* függvénynek a kimenete pedig az alábbi néhány sor:

```
Amit Robert Gida eszik az: tej es suti
Amit Micimacko eszik az: viz es mez
Amit Micimacko eszik az: mez es mez
```

A *Traits* technika előnye, hogy könnyű új elem és szereplő osztályokat úgy felvenni a már létezőkhöz, hogy a kész kódon nem kell változtatni semmit, sőt akár úgy is lehetne, hogy a már meglévő kódnak nem ismerjük a tartalmát, csak a hozzájuk tartozó interfészeket (típus definíciók, metódusok). Mivel új elemek hozzáadása során a meglévő kódhoz nem kell hozzányúlni, ezért azt nem is kell újratesztelni, kivéve az újonnan megírt részt. Tehát megtehetjük azt, hogy a kód változtatása nélkül létrehozunk pl. egy *Füles* nevű szereplőt, aki tejet iszik és mézet eszik, de ugyanígy felvehetnénk új elem osztályokat is:

```
struct Fules {
      friend ostream& operator<<(ostream& os, const Fules&)
      {return os << "Fules";}
};

template<>
class SzereploTraits<Fules> {
public:
      typedef Tej ital_tipus;
      typedef Mez uzsonna_tipus;
};
```

Használata pedig az előzőekhez hasonló:

```
Rajzfilm<Fules> rf4;
rf4.szerepel();
```

A kiment eredménye pedig:

```
Amit Fules eszik az: tej es mez
```

Policy (eljárásmód)

A *Policy* technikával a sablonunk viselkedését szabályozhatjuk, oly módon, hogy bizonyos funkcionalitást leválasztunk, külső osztályban valósítunk meg és sablon paraméterként adunk át. Ezzel a technikával más programozók személyre szabhatják a sablon osztályunkat, akár úgy is, hogy nem ismerik pontosan a sablon kódját. A *Policy* technikának a lényege, hogy egy osztálysablonhoz felveszünk egy új típus sablon paramétert, és az érkező osztály metódusait fogja használni az osztálysablon. Lássunk erre egy konkrét példát az előző rajzfilmes kód kiegészítéseképpen:

```
class Eves {
```

```
public:
      static const char* teendo() {return "eszik";}
};

class Kajalas {
public:
      static const char* teendo() {return "kajal";}
};

template <class Szereplo, class Teendo, class Traits =
SzereploTraits<Szereplo> >
class Rajzfilm {
      typedef typename Traits::ital_tipus ital_tipus;
      typedef typename Traits::uzsonna_tipus uzsonna_tipus;
      ital_tipus ital;
      uzsonna_tipus uzsonna;
      Szereplo szereplo;
      public:
      void szerepel() {
            cout << "Amit " << szereplo << " " << Teendo::teendo()
            << ": " << ital << " es " << uzsonna << endl;
      }
};
```

Mint látható, a *Rajzfilm* sablon paramétereinek száma háromra módosult, a második paraméteren keresztül lehet a sablonhoz funkcionalitást rendelni. Jelen példában a *Teendo* sablonparaméter azért került a második helyre, mert a *Traits* paraméternek alapértelmezett értéket adtunk és a *default* értékkel rendelkező paramétereknek az utolsó paramétereknek kell lenniük. Az adott funkcionalitás kihasználására pedig a *szerepel* metódusban van példa: a *Teendo* sablonparaméteren keresztül hivatkozunk a statikus függvényre. A sablon példányosításakor olyan osztályt kell megadni, melyben szerepel egy *teendo* nevű statikus metódus, ennek teljesülése fordítási időben kerül ellenőrzésre. Nézzünk egy példát a fenti kód futtatására:

```
int main() {
      Rajzfilm<RobertGida, Eves> rf1;
      rf1.szerepel();
      Rajzfilm<Micimacko, Kajalas> rf2;
      rf2.szerepel();
      Rajzfilm<Micimacko, Kajalas, EgyebTraits> rf3;
      rf3.szerepel();
      return 0;
}
```

A program kimenete pedig a következő:

```
Amit Robert Gida eszik: tej es suti
Amit Micimacko kajal: viz es mez
Amit Micimacko kajal: tej es mez
```

 © Ferenc Rudolf, SzTE

A kimenetből látható, hogy a főprogramban megadott eljárásmód szerint történt az „eszik” illetve „kajál” kiírása. Ez a kis példa bemutatta a *policy* technika alapötletét. Az STL implementációban a konténerek memória allokátorai ezen elv szerint vannak implementálva.

Curiously recurring template pattern („szokatlan módon ismétlődő” saját ősosztály)

A *Curiously recurring template pattern* független az eddig megismert idiómáktól. Maga a technika Jim Coplien nevéhez fűződik. Az idióma lényege, hogy bizonyos esetekben közös ősosztály helyett „szokatlan módon ismétlődő” saját ősosztály használható.
Vegyük azt az alap problémát, hogy nyilván szeretnénk tartani, hogy hány darab adott típusú objektum él a memóriában. Egy nagyon egyszerű megoldási módját a következő programkód tartalmazza:

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

class CountedClass {
      static int cnt;
public:
      CountedClass() {++cnt;}
      CountedClass(const CountedClass&) {++cnt;}
      ~CountedClass() {--cnt;}
      static int getCount() {return cnt;}
};

int CountedClass::cnt = 0;
```

Tehát az osztályba felveszünk egy statikus változót és ennek az értékét megnöveljük, ha új objektum keletkezik (konstruktor híváskor) és csökkentjük egyel, ha egy objektumot törlünk (destruktor híváskor). Nézzünk egy példát a fenti kód futtatására:

```
int main() {
      CountedClass a;
      cout << CountedClass::getCount() << endl; // 1
      CountedClass b;
      cout << CountedClass::getCount() << endl; // 2
      {
            CountedClass c(b);
            cout << CountedClass::getCount() << endl; // 3
            a = c;
            cout << CountedClass::getCount() << endl; // 3
      }
      cout << CountedClass::getCount() << endl; // 2
      return 0;
}
```

Ez a megoldás jól működik, de elég primitív megoldás, mivel minden osztályban kell lennie egy ilyen statikus változónak, melyben nyilvántartjuk az élő objektumok számát, valamint minden konstruktornak és destruktornak naprakészen kell tartania a számlálót.
Egy másik megközelítés, hogy veszünk egy ősosztályt, és abban deklaráljuk a statikus változót.

```
#include <iostream>
```

```
using namespace std;

class Counted {
	static int cnt;
public:
	Counted() {++cnt;}
	Counted(const Counted&) {++cnt;}
	~Counted() {--cnt;}
	static int getCount() {return cnt;}
};

int Counted::cnt = 0;

class CountedClass : public Counted {};
class CountedClass2 : public Counted {};

int main() {
	CountedClass a;
	cout << CountedClass::getCount() << endl; // 1
	CountedClass b;
	cout << CountedClass::getCount() << endl; // 2
	CountedClass2 c;
	cout << CountedClass2::getCount() << endl; // 3 (hiba)
	return 0;
}
```

Látható, hogy ez a megoldás rosszul működik, mivel nem lehet számon tartani, hogy egy adott osztályból mennyi objektum él, hanem csak azt, hogy a teljes öröklődési hierarchiában szereplő osztályokból hány objektum van életben. Ennek az az oka, hogy ugyanazt a statikus adattagot örökli minden gyerek osztály.

Erre a problémára a megoldás a *saját ismétlődő ősosztály* használata. Vegyük a következő osztálysablont:

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

template<class T>
class Counted {
	static int cnt;
public:
	Counted() {++cnt;}
	Counted(const Counted<T>&) {++cnt;}
	virtual ~Counted() {--cnt;}
	static int getCount() {return cnt;}
};


template<class T>
int Counted<T>::cnt = 0;
```

Ha példányosítjuk a *Counted* sablont valamilyen típussal, akkor a fordító egy új osztályt fog létrehozni, ahol a *T* paraméter minden előfordulási helyére a paraméterként átadott típus lesz beírva. Így tehát egy új független osztály jön létre, amit használhatunk ősosztálynak. Készítsünk is két osztályt, melyeket a *Counted* osztályból származtatunk:

 © Ferenc Rudolf, SzTE

```
class CountedClass : public Counted<CountedClass> {
      /*...*/
};

class CountedClass2 : public Counted<CountedClass2> {
      /*...*/
};
```

Mindkét osztálynál a *Counted* ősosztályt más típussal paramétereztük fel, így két egymástól független különböző ősosztály fog példányosulni, és mindkét ősosztály rendelkezik saját statikus változóval. A példában látható, hogy a *Counted* sablonosztályt a gyerekosztállyal paramétereztük fel. Ez megtehető, ha a *T* paramétert nem használjuk fel a sablonosztályban. Viszont ezzel a megoldással garantáljuk, hogy mindig új osztály keletkezzen. A következő main megvalósítással már helyes eredményeket kapunk:

```
int main() {
      CountedClass a;
      cout << CountedClass::getCount() << endl; // 1
      CountedClass b;
      cout << CountedClass::getCount() << endl; // 2
      CountedClass2 c;
      cout << CountedClass2::getCount() << endl; // 1 (!)
      return 0;
}
```

TEMPLATE METAPROGRAMOZÁS

Metaprogramozás segítségével bizonyos műveletek eredményei már fordítási időben kiszámolhatók, gyorsítva ezzel a program futását úgy, hogy a fordító által az eredmények már be lesznek fordítva a programba és így a futás során kevesebbet kell számolni. A metaprogramozás Turing-teljes, tehát támogatja a szelekciót (például specializációval) és az ismétlést (rekurzióval). Metaprogramozással bármilyen számítási feladat megoldható.
Tekintsük a következő egyszerű példát, ahol metaprogramozással mondjuk meg, hogy két szám közül melyik a nagyobb:

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

template<int n1, int n2>
struct Max {
	static const int val = n1 > n2 ? n1 : n2;
};

int main() {
	cout << Max<10,20>::val << endl;
	return 0;
}
```

Fordításkor a 10-es és a 20-as érték behelyettesítődik az *n1* és *n2* paraméterek helyére. A modern fordítóprogramok többek között ún. *konstans propagáció*s optimalizáló algoritmust is alkalmaznak, melynek az a lényege, hogy a fordítóprogram minden olyan konstans kifejezésnek kiszámolja az értékét, amelynek értéke kiszámolható fordítási időben. Ha a fordító ki tud számolni egy értéket, akkor azt be is helyettesíti a megfelelő formula helyére. Jelen példában a behelyettesítés után ez a sor áll:

```
	static const int val = 10 > 20 ? 10 : 20;
```

De ez a kifejezés fordítási időben kiszámítható, mivel minden érték adott és a rajtuk végzett művelet egyszerűen elvégezhető, ezért a fordított állományba nem a kifejezés kerül, hanem a kifejezés eredménye.

```
	static const int val = 20;
```

Nézzünk egy másik példát metaprogramozásra, a faktoriális számolását. Egy szám faktoriálisát megkapjuk, ha 1-től az adott számig összeszorozzuk a számokat. Vizsgáljuk meg az alábbi kódot:

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

template<int n>
struct Factorial {
	static const int val = Factorial<n-1>::val * n;
};
```

 © Ferenc Rudolf, SzTE

Ezzel lényegében leképeztük C++ nyelvre a faktoriális-számítás rekurzív képletét. Ehhez adni kell még egy megállási feltételt, mivel ha ezt elmulasztanánk, akkor nem lehetne lefordítani a programot, mivel a fordítóprogram végtelen rekurzióba esne, amint próbálná lepéldányosítani az egyre kisebb értékkel felparaméterezett sablont. A megállási feltételt sablon specializációval oldjuk meg. Ha az *n* értéke 0, akkor a konstans *val* változónk értéke legyen 1 (mivel 0! = 1).

```
template<>
struct Factorial<0> {
      static const int val = 1;
};
```

Példa a sablon használatára:

```
int main() {
      cout << Factorial<3>::val << endl;
      return 0;
}
```

Fordítás során a fordító elkezdi példányosítani a *Factorial* sablont és elkészíteni a *Factorial_3* nevű osztályt, melynek minden *n* paraméter előfordulási helyére a 3 kerül. A *val* értéke a következőképpen néz ki:

```
      static const int val = Factorial<2>::val * 3;
```

Mivel a *Factorial_2* osztály még nem létezik, ezért a fordító nem tudja kiszámolni ezt a kifejezést, így félbehagyja a sablon példányosítását és nekilát a *Factorial* sablon példányosításának a 2-s értékre. A fordító elkezdi elkészíteni a *Factorial_2* osztályt. A *Factorial_2* osztálynak a *val* erteke a következő:

```
      static const int val = Factorial<1>::val * 2;
```

Az előzőekhez hasonlóan ezt az értéket se lehet még meghatározni. Így a fordító elkezdi elkészíteni a *Factorial_1* osztályt, melynek a *val* értéke a következő:

```
      static const int val = Factorial<0>::val * 1;
```

A fordító ezután elkészíti a *Factorial_0* osztályt, mely osztályt specializációként adtunk meg. A *Factorial_0* osztálynak a *val* értéke már konkrét szám (1), így a fordító eggyel vissza tud lépni a *Factorial_1* osztály készítéséhez. Itt a kifejezés úgy néz ki, hogy 1 * 1, amit könnyen ki tud számolni a fordító, így ide már csak az eredmény kerül. Ezután nincs szükség többet a *Factorial_0* osztályra. Ha kiszámolta a *Factorial_1* osztály *val* értékét, akkor visszalép a fordító a *Factorial_2* osztály készítéséhez. Itt a kifejezés az 1 * 2, aminek az értékét megint csak meg tudja határozni, így ide a 2 érték íródik. A Factorial_1 osztályra sincs többé szükségünk. Végül a *Factorial_3* osztály *val* értékét is már meg tudja határozni. A *Factorial_2* osztályra sincs szükség a továbbiakban. Így egyetlen egy szám marad, a 6, és egyetlen egy osztály a *Factorial_3*. A 6-os értéket a fordítóprogram beírja a *main*-be, és így a *Factorial_3*-ra sincs már szükség. Futási időben egyetlen dolga marad csak a főprogramnak, hogy kiírja a végeredményt.

KIFEJEZÉS SABLONOK

A kifejezés sablonok (*expression templates*) célja legfőképp a matematikai számítások gyorsítása, kihasználva a C++ adta nyelvi lehetőségeket. Ezek a sablonok segítik a fordítási időben történő optimalizálást, segítségükkel legalább olyan gyors kód készíthető, mint Fortran-ban kézzel optimalizálva. Ez jelentős mondás, hiszen a Fortran programozási nyelv direkt arra a célra lett kifejlesztve, hogy matematikai számításokat végezzen, ezért főleg fizikusok használják. Ez a módszer a műveletek használatakor megőrzi a természetes matematikai jelölésmódot, annak köszönhetően, hogy C++-ban lehetőség van operátor kiterjesztés (*operator overloading*) segítségével felülírni a matematikai műveleti jeleket, amelyek ezután tetszőleges saját osztályra értelmezhetőek lesznek. A kifejezés sablonok segítségével kifejezéseket adunk át függvény argumentumként, ami sok C++ matematikai könyvtár alapja. A kifejezés sablonok használatával nagymértékben csökkenteni lehet a program műveleti komplexitását és az általa felhasznált memóriát is. Egy egyszerű példaprogram segítségével a következőkben bemutatásra kerül a kifejezés sablonok implementálása és használata.

A feladat

Hozzunk létre egy saját vektor osztályt, amely támogatja a vektorok összeadását! Természetesen a többi műveletet is meg lehetne valósítani, de erre most nem térünk ki. Először adunk egy egyszerű megoldást, majd megmutatjuk, hogy ezt mennyivel jobban le lehet implementálni kifejezés sablonok használatával.

Egy egyszerű megoldás

Először nézzük azt a megoldást, ami mindenkinek először eszébe jutna:

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

template<class T, long N>
class Vektor {
    T data[N];
public:
    friend Vektor<T,N> operator+(const Vektor<T,N>& left,
                                 const Vektor<T,N>& right) {
      Vektor<T,N> tmp;
      for (long i = 0; i < N; ++i)
          tmp.data[i] = left.data[i] + right.data[i];
      return tmp;
    }

    const T& operator[](long i) const {
      return data[i];
    }

    T& operator[](long i) {
      return data[i];
    }
};
```

 © Ferenc Rudolf, SzTE

A *Vektor* sablon osztály tetszőleges típusú elemet tud tárolni, összesen *N* darabot. A + műveleti jel kiterjesztésének segítségével össze lehet adni két *Vektor* típusú objektumot. Az operátor eljárás két konstans vektor hivatkozást vár, ami azt jelenti, hogy a kapott értékek nem módosíthatók, és a referencia szerinti átadás miatt nem másolódnak le feleslegesen, hanem közvetlenül lehet rájuk hivatkozni a függvény törzsén belül. Ez az operátor kiterjesztés egy globális függvény, nem része az osztálynak, csak az osztályon belül van definiálva, és mivel *friend* módosítóval rendelkezik, hozzáfér az osztály privát adattagjaihoz is. A függvény belsejében létrejön egy ideiglenes *Vektor* objektum (*tmp*) a kapott *T* és *N* sablon paraméterekkel. Ezután egy *for* ciklus bejárja a paraméterben kapott *Vektor* objektumok tartalmát, azaz a *data[]* tömbből kiolvassa az adott pozíción található értékeket, majd összeadva azokat értékül adja az ideiglenes *Vektor* objektum *data[]* tömbjének aktuális pozíciójú elemének. Végezetül visszatér a lokális *tmp* objektummal, ami azt jelenti, hogy az egész objektum tartalma lemásolódik a *stack*-re. (A *stack* egy verem típusú adatszerkezet, ezen keresztül történik a paraméterátadás és érték visszaadás a függvények hívásakor.)
A másik két metódus szintén operátor kiterjesztést valósít meg. Az első a lekérő index operátort fogalmazza meg a *Vektor* típusra. A szögletes zárójelpáron belüli értéket *long* típusúnak definiáltuk, hiszen az *N* értéke is *long*, így hivatkozni tudunk a vektor teljes hosszára. Ez a metódus egyszerűen visszaadja a *Vektor data[]* tömbjének a zárójelpáron belül hivatkozott pozíción lévő objektumot. A *const* kulcsszó használata biztosítja, hogy a visszaadott értéket ne lehessen módosítani, azaz valóban csak lekérést valósítson meg.
A másik index operátor megvalósítás azonban már a *const* kulcsszó hiánya miatt megengedi a módosítást, azaz ami megkapja az adatra mutató referenciát, az módosíthatja is annak értékét.
A továbbiakban létrehozunk még két sablon függvényt:

```
template<class T, long N> void init(Vektor<T,N>& v) {
    for (long i=0; i<N; ++i)
      v[i] = rand() % 100;
}

template<class T, long N> void print(Vektor<T,N>& v) {
    for (long i=0; i<N; ++i)
      cout << setw(4) << v[i];
    cout << endl;
}
```

Ezek rendre a *Vektor* típusú objektum adattömbjének véletlenszerű adatokkal való feltöltését végzik 0 és 100 közötti egész számokkal, valamint a *Vektor* típusú objektum elemeinek kiíratását (4 karakter szélességen kiírva minden elemet).
Nézzük meg a *Vektor* osztály viselkedését a következő *main* függvény megvalósítással:

```
int main() {
    Vektor<int,5> v1;
    init(v1);
    print(v1);
    Vektor<int,5> v2;
    init(v2);
    print(v2);
    Vektor<int,5> v3;
    v3 = v1 + v2;
    print(v3);
    Vektor<int, 5> v4;
    v4 = v1 + v2 + v3;
```

```
    print(v4);
    return 0;
}
```

A program futtatása után a következő eredményt kapjuk:

```
41     67     34     0      69
24     78     58     62     64
65     145    92     62     133
130    290    184    124    266
```

Ezek a sorok rendre a *v1*, *v2*, *v3* és *v4* vektorok elemeit mutatják. Az első két vektort az *init* metódus véletlenszerű adatokkal tölti fel, majd a harmadik vektor az első két vektor összegét, míg a negyedik vektor az első három vektor összegét reprezentálja.
Látható, hogy a megoldás helyes, jó eredményt ad. Mégis több okból lehetnek kétségeink a program minőségét illetően. Először is, az összeadás megvalósításakor létrejött egy ideiglenes *Vektor* objektum a *stack*-en, ami felesleges memóriafoglalásnak minősül és gigabájtos méretű vektorok esetén igencsak problémás lehet, hiszen a *stack* kisméretű memória. Emellett, amikor visszaadja ennek az ideiglenes lokális változónak az értékét, akkor ismét másolás történik a *stack*-en. Nyilvánvaló, hogy ez a megoldás első körben végül is egy helyes megoldás, de igen idő- és tárigényes.
Nézzük csak meg például a *v4 = v1 + v2 + v3* művelet valójában mekkora számítást és helyet igényel:

```
v4 = v1 + v2 + v3;
v4 = operator+(v1,v2) + v3;
v4 = tmp1 + v3;
v4 = operator+(tmp1,v3);
v4 = tmp2;
v4.operator=(tmp2);
```

Először is *v1 + v2* hajtódik végre, amikor is létrejön egy ideiglenes *Vektor* objektum a *stack*-en. Amikor az *operator+* metódus visszaadja az összeadás eredményét, még egyszer átmásolódik a *stack*-re. Ezután ez a temporális eredmény kerül összeadásra a *v3 Vektor* objektummal, amely még egy ideiglenes *Vektor* létrejöttét eredményezi a *stack*-en. Ez az eredmény visszaadásakor újbóli stack másolást eredményez. Csak a *v4*-ben van szükség a műveletsor eredményére (vagyis az összegre), a köztes ideiglenes eredményeket sehol sem használjuk fel a programunk során, felesleges azok eltárolása. Látszik, hogy feleslegesen sok másolás történik és sok *stack* memória kerül felhasználásra ezzel a megoldással.

Egy jobb megoldás

A kifejezés sablonok használatával mind memória, mind gépidő megtakarítható. Egy jó, de még mindig nem teljes megoldást mutat be a következő kódrészlet.

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

template<class,long> class VektorSzum;

template<class T, long N>
class Vektor {
```

```
    T data[N];
public:
    Vektor<T,N>& operator=(const VektorSzum<T,N>& right) {
        for (long i = 0; i < N; ++i)
             data[i] = right[i];
        return *this;
    }
    const T& operator[](long i) const {
      return data[i];
    }

    T& operator[](long i) {
      return data[i];
    }
};
```

Ebben a megoldásban megjelenik az összeadás fogalma, mint sablonosztály. Mivel a *VektorSzum* sablonosztálynak nincs törzse (most csak deklaráljuk), a sablonparamétereknél is elég csak a típusokat megadni, nem kell azonosítót rendelni hozzájuk.
A következő különbség az előző megoldáshoz képest az, hogy itt összeadás operátor helyett értékadás operátort használunk (=). Ez azt jelenti, hogy az értékadás oldala a *Vektor* osztály egy példánya lesz, míg a jobb oldala egy *VektorSzum* típusú objektum. Tehát a *Vektor* objektum egy vektor összeget kér értékadásnál. Az értékadó operátor megvalósítása a következőképpen történik: a *for* ciklus végigmegy a paraméterben kapott vektorösszeg elemein (figyeljük meg a [] operátor használatát a *VektorSzum* típusra), majd mindegyiket egyesével értékül adja a bal oldali operandus (azaz maga az objektum példány) adat tömbje megfelelő elemének. Végül pedig az objektum saját magára mutató referenciájával tér vissza, ahogy az a függvénydefinícióban visszatérési értékként is látható. A két index operátor ugyanúgy került megvalósításra, mint az előző példában.
Vizsgáljuk meg a megoldás következő kódrészletét:

```
template <class T, long N>
class VektorSzum {
      const Vektor<T,N>& bal;
      const Vektor<T,N>& jobb;
public:
      VektorSzum(const Vektor<T,N>& b, const Vektor<T,N>& j) : bal(b),
jobb(j) {}
      T operator[](long i) const {return bal[i] + jobb[i];}
};

template<class T, long N>
inline VektorSzum<T,N>
operator+(const Vektor<T,N>& bal, const Vektor<T,N>& jobb) {
      return VektorSzum<T,N>(bal,jobb);
}
```

Látható, hogy itt már teljes egészében ki van dolgozva a *VektorSzum* osztály (vagyis itt került definiálásra a korábbi deklaráció). A korábbi kódrészletnél azért volt szükség az osztály deklarációjára, hogy a *Vektor* osztályon belül hivatkozni lehessen erre a típusra, ne eredményezzen fordítási hibaüzenetet.
A *VektorSzum* sablon osztály hasonlóan két paramétert vár, a tartalmazandó elemek típusát és a vektor méretét. Két *Vektor* hivatkozás típusú adattagja van, hiszen ez az osztály két vektor összeadását képviseli. A két adattag az összeadás bal és jobb oldali operandusának felel meg.

Ezek az adattagok, hasonlóan az első példához, azért konstans referenciával vannak hivatkozva, hogy ne lehessen az értékeiket megváltoztatni az osztályon belül. A *VektorSzum* osztály konstruktora két *Vektor* típusú konstans referenciát vár paraméterül, majd ezeket a konstruktor inicializációs listában értékül adja a *bal* és *jobb* adattagoknak.
A példaprogramban a hangsúly a [] operátoron van. Itt jön ugyanis az ötlet lényege. Ha egy *VektorSzum* típusú objektumra használjuk a lekérő index operátort, akkor az az *összeg* hivatkozott elemét fogja visszaadni (amit így csak szükség esetén számol ki).
Nem elhanyagolható az a tény sem, hogy itt nem történik felesleges *stack* memória foglalás, valamint nem adja össze az összes tagot egyesével feleslegesen. Mindig csak az aktuálisan szükséges elemeket kéri le a hivatkozott bal- és jobboldali operandusoktól. Ezáltal a *VektorSzum* egy kevés memóriát felhasználó objektum lesz, ellentétben az első példában megadott *Vektor* objektummal, amely elég pazarlóan bánt a memóriával.
Továbbá, érdekes még az összeadás műveleti jel kiterjesztése, amit meghívva tulajdonképpen nem végez összeadást, csak létrehoz és visszaad egy, a paraméterekből összeállított *VektorSzum* típusú objektumot. Ennek megfelelő elemeire később hivatkozhat az, aki ezt a referenciát megkapja - így megkapva az összeadás eredményeit. Vegyük észre, hogy ez egy *inline* metódus, ami azt jelenti, hogy fordítási időben behelyettesítődik a függvény törzse minden hívás helyére. Mivel ez egy rövid metódus, így kódunk futási idejének optimalizálására tökéletesen megfelel az *inline* hívás használata, hisz így megszabadulunk a függvényhívás okozta plusz műveletektől.
Próbáljuk ki a hatékonyabb megoldást a következő *main* függvény megírásával:

```
int main() {
    Vektor<int,5> v1;
    init(v1);
    print(v1);
    Vektor<int,5> v2;
    init(v2);
    print(v2);
    Vektor<int,5> v3;
    v3 = v1 + v2;
    print(v3);
    return 0;
}
```

A program futtatása után a következő eredményt kapjuk:

```
41     67     34     0      69
24     78     58     62     64
65     145    92     62     133
```

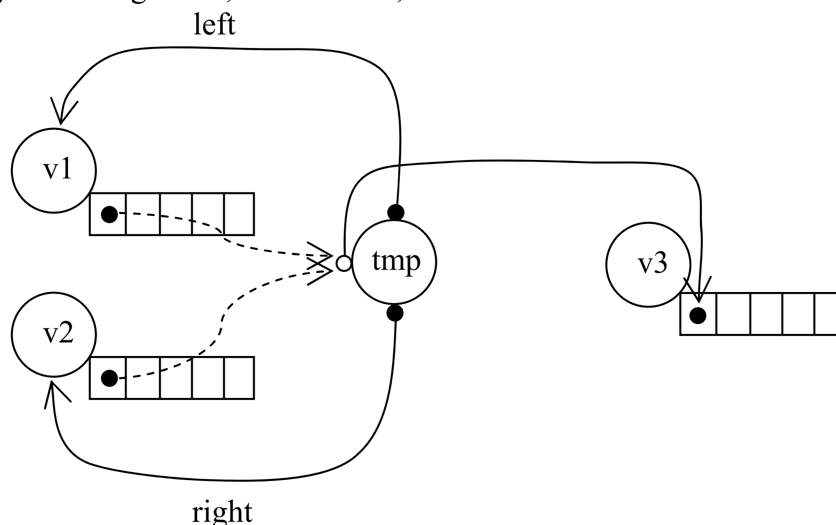
Látható, hogy ez a megoldás is helyesen működik, azonban nézzük meg, hogy valójában mi történik ennek a kódnak a hátterében. Tekintsük a *v3 = v1 + v2* műveletet:

```
v3 = v1 + v2;
v3 = operator+(v1,v2);
v3 = VektorSzum(v1,v2);
v3 = tmp1;
v3.operator=(tmp1);
```

Először is meghívódik a kiterjesztett + operátor a paraméterben átadott két *Vektor* típusú objektumra, melynek hatására egy temporális *VektorSzum* objektum jön létre a megadott két paraméterrel. Mivel a *Vektor* osztályban felülírtuk az értékadás operátort (jobb oldalon egy

 © Ferenc Rudolf, SzTE

VektorSzum objektumot vár paraméterben), ezért a *Vektor* = *VektorSzum* értékadás egy helyes művelet. Ne feledjük, hogy itt csak a *VektorSzum* objektum lesz ideiglenes, de ez kis méretű (mint ahogy azt korábban láthattuk). Ezután ez az értékadás operátor lesz meghívva a *Vektor* objektumra és az ideiglenesen létrejött *VektorSzum* típusú objektumra. Felhasználva az index operátor kiterjesztését kiolvassa a *VektorSzum* objektumból az összeadások eredményeit (tulajdonképpen az értékadáskor végzi el az összeadást).
Ahhoz, hogy jobban megértsük, mi is történt, tekintsük a következő ábrát:



A szaggatott nyilak a referencia hivatkozások, míg a folyamatos nyilak a valódi memória másolást jelentik. Vagyis a *v1* és *v2* tömbjeinek megfelelő indexen lévő elemei, mint egy-egy hivatkozás, kerülnek be az ideiglenesen létrejövő *tmp* objektumba, viszont az ideiglenes eredmény visszaadásakor tényleges másolás történik a *v3* megfelelő indexű tömbelemébe.
Ekkor a *v3* valóban a *v1* és *v2* vektorok összegét fogja tartalmazni, azonban sokkal kevesebb másolást és memóriafoglalást végzett el ezzel a megoldással, mint az az első próbálkozásunkkor történt. Nincsenek nagy ideiglenes objektumok (csak kicsik), és késleltetett számolást végeztünk (csak akkor számol, amikor arra ténylegesen szükség van).
Látszik, hogy mekkora erő rejlik a kifejezés sablonok használatában, azonban a most adott megoldás még nem tökéletes. Ugyanis csak két vektor összeadását támogatja a *VektorSzum* osztály, de mi a helyzet, ha például három vektort akarunk összeadni?
Ebben az esetben a következő példakód nem fordul le:

```
Vektor<int,5> v4;
v4 = v1 + v2 + v3;
```

Hiszen a megvalósításunk nem tartalmazza a *Vektor* és *VektorSzum* típusok közötti összeadást. Valóban, itt először a *v1* + *v2* összeadásból egy *VektorSzum* típusú objektum keletkezik, azonban *VektorSzum* + *Vektor* összeadás sehol sem került még definiálásra.

Egy teljes megoldás

Vezessük be a *VektorSzum* + *Vektor* összeadás fogalmát is a programunkban!

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

template <class T, long N, class Bal, class Jobb> class VektorSzum;

template<class T, long N>
class Vektor {
	T data[N];
public:
	template<class Bal, class Jobb>
	Vektor<T,N>& operator=(const VektorSzum<T,N,Bal,Jobb>& jobb) {
		for (long i = 0; i < N; ++i)
			data[i] = jobb[i];
		return *this;
	}

	const T& operator[](long i) const {
		return data[i];
	}

	T& operator[](long i) {
		return data[i];
	}
};
```

Figyeljük meg a *VektorSzum* sablon osztály paramétereit! Az osztály további két paramétere egy *Bal* és egy *Jobb* azonosítójú tetszőleges osztály paraméter. Az ötlet nem más, mint hogy képezzünk a sablonok segítségével különböző *VektorSzum* objektumokat kihasználva a fordító adta sablon példányosítási lehetőségeket. Nem kötjük meg, hogy mi az összeadás bal oldalán, illetve a jobb oldalán hivatkozott objektum típusa.
A *Vektor* osztály is bonyolultabbá válik. Sablonfüggvénnyé kell tenni az értékadó operátort is, azért, hogy kezelni tudja a tetszőleges bal- és jobboldalú összeadást. Az értékadó operátor egy *VektorSzum* objektumot vár továbbra is, azonban a *VektorSzum*-hoz most két plusz paraméterre is szükségünk van, amit itt meg kell adni. Ettől eltekintve a többi rész ugyanúgy van megvalósítva a *Vektor* osztályon belül, mint eddig.
Kicsit bonyolultnak tűnik, hogy sablon osztálynak készítünk sablon tagfüggvényt – tehát még az osztály sablonparaméterein kívül egyéb sablonparamétereket is kap –, de ezzel egy igen hatékony megvalósítás jön létre.
Nézzük most a *VektorSzum* sablonosztály módosított forráskódját:

```
template <class T, long N, class Bal, class Jobb>
class VektorSzum {
	const Bal& bal;
	const Jobb& jobb;
public:
	VektorSzum(const Bal& b, const Jobb& j) : bal(b), jobb(j) {}
	T operator[](long i) const {
		return bal[i] + jobb[i];
	}
};

template<class T, long N>
inline VektorSzum<T,N,Vektor<T,N>,Vektor<T,N> >
operator+(const Vektor<T,N>& bal, const Vektor<T,N>& jobb) {
```

 © Ferenc Rudolf, SzTE

```
      return VektorSzum<T,N,Vektor<T,N>,Vektor<T,N> >(bal,jobb);
}



template<class T, long N, class Bal, class Jobb>
inline VektorSzum<T, N, VektorSzum<T,N,Bal,Jobb>, Vektor<T,N> >
operator+(const VektorSzum<T,N,Bal,Jobb>& bal, const Vektor<T,N>& jobb) {
      return VektorSzum<T,N,VektorSzum<T,N,Bal,Jobb>, Vektor<T,N>
>(bal,jobb);
}
```

Itt már a sablonparaméterek segítségével a *VektorSzum* osztály „tetszőleges" két osztály összegét képes reprezentálni, pontosabban a bal és jobb oldali operandusai a sablonparaméterekben megadott típusúak lesznek. A konstruktor is ennek megfelelően módosul, a lekérő index operátor ugyanúgy az összeget adja vissza, mint korábban.
Az előző megoldásban csak egyfajta kiterjesztését adtuk meg az összeadás operátornak (emiatt kaptunk fordítási hibát *VektorSzum* + *Vektor* összeadás esetén). Jelen esetben *Vektor* + *Vektor* és *VektorSzum* + *Vektor* összeadás definíciókat fogalmazunk meg. Az előzőhöz hasonlóan most is inline módon megpróbálja majd a fordító ezeket a műveleteket behelyettesíteni a hívás helyére, amennyiben lehetséges. Természetesen ennek csak akkor van értelme, ha rövid a függvény törzse, ami jelen esetekben teljesül. Ezzel a felesleges *stack*-re másolást ki lehet küszöbölni.
A helyes megoldás kulcsa ezekben a sorokban található, itt a visszaadandó *VektorSzum* objektumot a megfelelő sablon paraméterekkel kell példányosítani. Mivel a *VektorSzum* objektumhoz a sablon paraméterek száma 4-re nőtt, mivel meg kell adnunk, hogy mi a bal és jobb oldali operandus típusa, így a kétféle összeadás esetén kétféleképpen kell példányosítani a *VektorSzum* objektumot. *Vektor* + *Vektor* összeadás esetén *Vektor<T,N>*, *Vektor<T,N>* paramétereket kell még az eddigi *T* és *N* mellé megadni, *VektorSzum* + *Vektor* esetén pedig *VektorSzum<T,N,Left,Right>*, *Vektor<T,N>* paramétereket. Ez első ránézésre kicsit zavarónak tűnhet, de vegyük észre, mennyivel hatékonyabb kódhoz jutottunk.
Érdemes még figyelmet fordítani arra, hogy ha sablon argumentumként sablont írunk, akkor a két > jel között egy szóköznek szerepelnie kell. Ennek hiányában egyes régebbi fordítók a >> *bitshift* operátornak vélnék ezt a karaktersorozatot, és emiatt fordítási hibát adna, mivel a fordító nem tudja értelmezni, mint kifejezést.
Ellenőrizzük le megoldásunk helyességét a következő *main* függvény segítségével:

```
int main() {
    Vektor<int,5> v1;
    init(v1);
    print(v1);
    Vektor<int,5> v2;
    init(v2);
    print(v2);
    Vektor<int,5> v3;
    v3 = v1 + v2;
    print(v3);
    Vektor<int,5> v4;
    v4 = v1 + v2 + v3;
    print(v4);
    return 0;
}
```

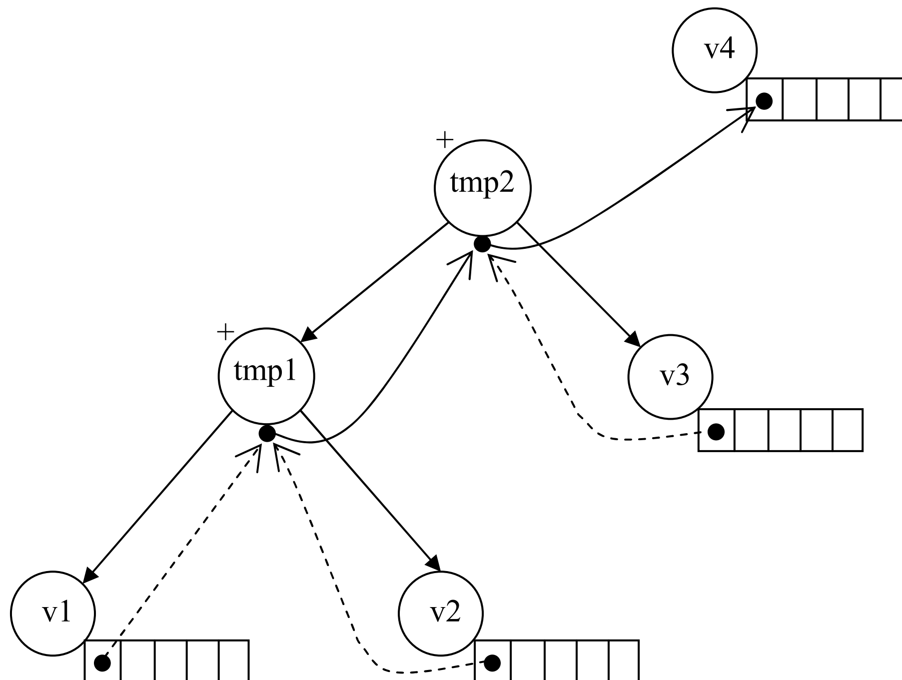
A *v3 = v1 + v2* utasítás esetén feltűnhet, hogy a + operátor nem kapott sablon paramétereket. Ez azért lehetséges, mert a fordító már tudja, hogy a *v1 int*-tel és 5-tel lett példányosítva, valamint a *v2* is ugyanilyen paraméterezésű.
A program futtatása után a következő eredményt kapjuk:

```
41     67     34     0      69
24     78     58     62     64
65     145    92     62     133
130    290    184    124    266
```

Látható, hogy a megoldás ez esetben is korrekt. Tekintsük most a *v4 = v1 + v2 + v3* összeadás mögött megbúvó metódushívásokat:

```
v4 = v1 + v2 + v3;
v4 = operator+(v1,v2) + v3;
v4 = VektorSzum(v1,v2) + v3;
v4 = tmp1 + v3;
v4 = operator+(tmp1,v3);
v4 = VektorSzum(tmp1,v3);
v4 = tmp2;
v4.operator=(tmp2);
```

Először a *v1* és *v2* objektumokra hívódik meg az összeadás operátor, aminek eredményeképp előáll egy *VektorSzum* objektum a *v1*, *v2* paraméterekkel. Ez egy ideiglenes objektum, nevezzük el *tmp1*-nek. Ezután a *tmp1*-hez hozzáadja a *v3*-at. Ez már egy *VektorSzum + Vektor* összeadást jelent. Mivel ezt az esetet is kezeltük a fenti kódban, ezért ezzel most nem lesz gond. Ismét egy *VektorSzum* típusú objektum fog előállni, ami szintén ideiglenes objektum, nevezzük *tmp2*-nek. Végül a *tmp2*-t fogjuk értékül adni a *v4* vektornak, ekkor fog megtörténni a tényleges összeadás, amikor is majd visszanyúl a megfelelő tagokért, kiolvasva azokat az egyes vektorokból.
Ezek a hívások a következőképpen néznek ki:



A szaggatott nyilak itt is a referencia hivatkozásokat, míg a folyamatos nyilak a valódi memória másolást jelentik. A *tmp1* alatti kifejezésfában képződik egy valódi érték, majd felmásolódik a *tmp2*-be és ott meg is szűnik (a *for* ciklus következő lépésében jön létre újra). A felmásolt érték összeadódik a *v3*-ból vett hivatkozás által mutatott értékkel, majd hasonló módon felmásolódik a *v4* eredmény objektum megfelelő tömbelemébe.
Az ábrán látható típusok fordító által használt képzelt elnevezései, ha a példányosítási paramétereket egymás mögé másoljuk aláhúzás karakterrel elválasztva:

```
v1:   Vektor_int_5
v2 :  Vektor_int_5
tmp1:       VektorSzum_int_5_Vektor_int_5_Vektor_int_5
v3:   Vektor_int_5
tmp2:
      VektorSzum_int_5_VektorSzum_int_5_Vektor_int_5_Vektor_int_5_Vektor_in
t_5
```

Megfigyelhető, hogy a *tmp1* és a *tmp2* *VektorSzum* típusa különböző. Az alapjuk ugyanaz a sablon, de különböző a paraméterezésük.
Sajnos ez még mindig nem a teljes megoldás, hiszen pl. a *v1 + (v2 + v3)* összeadás nem működik, mivel a *Vektor + VektorSzum* esete még nincs lekezelve, azonban ez már egy egyszerű *operator+* kiterjesztéssel megoldható a többihez hasonlóan.
Ennek a megoldásához a következő kódrészletre van még szükségünk:

```
template<class T, long N, class Bal, class Jobb>
inline VektorSzum<T, N, Vektor<T,N>, VektorSzum<T,N,Bal,Jobb> >
operator+(const Vektor<T,N>& bal, const VektorSzum<T,N,Bal,Jobb>& jobb) {
      return      VektorSzum<T,N,Vektor<T,N>,      VektorSzum<T,N,Bal,Jobb>
>(bal,jobb);
}
```

Végső tanulságképp levonható, hogy a kifejezés sablonok használata esetén, mielőtt bármilyen igazi művelet megtörténne, előbb a kifejezés fája épül fel, majd azt értékeli ki – a módszer lényegében innen kapta a nevét.

 © Ferenc Rudolf, SzTE

GENERIKUS ALGORITMUSOK ÖSSZETEVŐI

A szoftverfejlesztésben az algoritmusok képezik a számítások magját. Az STL és a hozzá hasonló generikus programozási paradigmán alapuló könyvtárak azáltal, hogy olyan algoritmusokat (is) tartalmaznak, amelyek elemek bármilyen sorozatán – típustól függetlenül – képesek működni, nagymértékben segíthetik a szoftverfejlesztést, valamint az elkészült szoftver megértését és karbantartását.
Az STL-t sokan generikus konténerek gyűjteményeként kezelik, holott készítői eredetileg generikus algoritmusok gyűjteményének szánták (és az első változat nem is C++-ra, hanem ADA nyelvre lett kidolgozva). Az volt a céljuk, hogy szinte minden feladatra készítsenek egy előre definiált, biztonságosan működő, generikus algoritmust, hogy ne kelljen minden alkalommal pl. új ciklust/ciklusokat írni, ha azonos típusú adatok valamilyen halmazán akarunk műveleteket végrehajtani.
A generikus algoritmusok az általános használhatóság érdekében olyan sablonfüggvények, amelyek nem konkrétan egy adott konténerrel dolgoznak, hanem a konténereket bejáró iterátorokkal. Ennek köszönhető, hogy nem csak egy adott konténer típus esetén használhatóak. (Az iterátorokkal a későbbiekben részletesebben is foglalkozunk.)
Ez a lehetőség forradalmasította a szoftverfejlesztést. Az előzőek alapján egyértelműen látható az általános algoritmusok hasznossága. Használatuk azonban bizonyos interfészek/működések implementálását teszi szükségessé, ami némi tanulási időt igényel.

Generikus algoritmus használata

A generikus algoritmusok használatának bemutatásához nézzük először az egyik legegyszerűbbet, a *copy* algoritmust. Ez az algoritmus sorozatok másolására használható anélkül, hogy bármilyen ciklust implementálni kellene. A *copy* algoritmus deklarációja a következőképpen néz ki:

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

A *copy* sablonfüggvény három paramétert vár: az első másolandó elemre mutató iterátort, az utolsó másolandó elem utánra mutató iterátort, valamint a fogadó sorozat kezdetére mutató iterátort. Az algoritmus a harmadik paraméterben megadott helytől kezdve folyamatosan másolja az elemeket. Követelmény, hogy az első két paraméternek ugyanolyan típusúnak kell lennie, valamint hogy legyen elég hely a cél konténerben, mert ha nincs, akkor egyéb adatokat is felülírhat.
Ahhoz, hogy használjunk bármilyen STL-beli algoritmust, először is be kell *include*-olni az <*algorithm*> header fájlt, amely tartalmazza az összes STL-beli generikus algoritmust.
A *copy* algoritmus használatához vizsgáljuk meg az alábbi példakódot:

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

```
int main() {
	int a[] = {10, 20, 30};
	const size_t S = sizeof a / sizeof a[0];  // A tömb elemszáma
	int b[S];
	copy(a, a+S, b);
	copy(a, a+S, ostream_iterator<int>(cout," "));
	cout << endl;
	copy(b, b+S, ostream_iterator<int>(cout," "));
	cout << endl;
	return 0;
}
```

A példakód egy *a* nevű tömb elemeinek egy másik, *b* nevű tömbbe másolását végzi. A példakód megértéséhez fontos információ lehet, hogy a tömb nevének leírása valójában egy a tömb első elemére mutató pointert jelent, így alkalmazható rá a pointer aritmetika, és ily módon lesz a második paraméter az utolsó elem után mutató pointer. (Az iterátorok és a pointerek kezelése nagyon hasonló, így olyan konténerek esetében, amelyek az általuk tárolt elemeket egymás után írják a memóriába, bármelyik használható.) A második és harmadik *copy* hívás az *a* illetve *b* tömbök tartalmát a képernyőre másolja. Ezt egyelőre fogadjuk el, hogy így működik. Az ostream_iterator bemutatására az iterátorokat taglaló fejezetben kerül majd sor. A program futtatása után a következő eredményt kapjuk:

```
10 20 30
10 20 30
```

Természetesen a *copy* nemcsak tömbökön működik, hanem mint ahogyan a bevezetőben említettük, bármilyen sorozatot át tud másolni. Nézzünk egy másik példát, ahol a *copy* algoritmus egy sztringeket tároló *vector*-t másol át.

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

int main() {
	string a[] = {"egy", "ketto", "harom"};
	const size_t S = sizeof a / sizeof a[0];
	vector<string> v1(a,a+S); // az a tömböt elejétől a végéig betölti a
v1-be
	vector<string> v2(S);     // legyen benne eleg hely!
	copy(v1.begin(), v1.end(), v2.begin());
	copy(v1.begin(), v1.end(), ostream_iterator<string>(cout," "));
	cout << endl;
	copy(v2.begin(), v2.end(), ostream_iterator<string>(cout," "));
	cout << endl;
	return 0;
}
```

A *vector* egy STL-beli dinamikus tömb megvalósítás, ami automatikusan átméretezi magát, ha szükséges. (A későbbi fejezetekben részletesebben is bemutatásra kerülnek az STL konténerei.) Az program hasonlóan működik az előző példához. A *vector begin* metódusa

létrehoz egy iterátort, ami az első elemre mutat, az *end* metódusa pedig az utolsó elem utánra mutató iterátorral tér vissza. A program futtatása után a következő eredményt kapjuk:

```
egy ketto harom
egy ketto harom
```

Ebben a példában azonban a *vector* konténer dinamikussága nem lett kihasználva, hiszen a *v2* vektornak is előre lefoglaltuk a szükséges helyet. Még általánosabb kódot kaphatunk a *back_inserter* metódus használatával, ami egy speciális, beszúró iterátort ad vissza, ami kihasználja az STL konténerek dinamikusságát, és új elem beszúrásakor – amennyiben szükséges – növeli a konténer kapacitását (lásd a „Beszúró iterátorok" alfejezetet). Nézzük meg, hogyan néz ki a *back_inserter* metódus használatával a javított kód.

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

int main() {
      string a[] = {"egy", "ketto", "harom"};
      const size_t S = sizeof a / sizeof a[0];
      vector<string> v1(a,a+S);
      vector<string> v2;  // v2 ures!
      copy(v1.begin(), v1.end(), back_inserter(v2));
      copy(v1.begin(), v1.end(), ostream_iterator<string>(cout," "));
      cout << endl;
      copy(v2.begin(), v2.end(), ostream_iterator<string>(cout," "));
      cout << endl;
      return 0;
}
```

A program futtatása után a következő eredményt kapjuk:

```
egy ketto harom
egy ketto harom
```

Ahhoz, hogy megértsük a *copy* algoritmus működését, ismerkedjünk meg a *copy* algoritmus forráskódjával, ami a következőképpen néz ki:

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator
result) {
      while (first != last)
            *result++ = *first++;
      return result;
}
```

Minden lépésben átmásolja a megfelelő elemet a célhalmaz megfelelő helyére (kezdetben természetesen az első elemet a célsorozat első helyére), ezután lépteti mindkét iterátort (így azok a soron következő elemre fognak mutatni). Ezt addig ismétli, amíg a másolandó elemre

mutató iterátor az utolsó másolandó elem utánra nem mutat. Ekkor az algoritmus visszaad egy olyan iterátort, ami a cél konténer utolsó eleme utánra mutat, és ezzel vége az eljárásnak.

A megvalósításból látható, hogy a *copy* (és az STL többi generikus algoritmusa) működésének elengedhetetlen feltétele, hogy az átadott iterátoroknak implementálniuk kell a dereferencia/indirekció (*) és poszt-inkrementálás (++) operátorokat. Abban az esetben, ha saját iterátort használunk, az algoritmusok használata előtt biztosítani kell a fenti operátorok meglétét (egyébként le se fordul a kód).

Predikátumok

Mint láttuk, a *copy* algoritmus kiválóan megoldja a különböző típusú sorozatok átmásolását. Azonban sokszor szükség lehet arra, hogy az adott sorozatnak csak azon elemeit másoljuk át, amelyek megfelelnek bizonyos követelményeknek. Számos STL-beli algoritmusnál van lehetőség arra, hogy átadjunk egy olyan függvényt, ami megvizsgálja, hogy az adott elem kielégíti-e az általunk állított követelményeket, és egy ennek megfelelő logikai értékkel tér vissza. Az ilyen függvényeket hívjuk *predikátum*oknak. Nézzünk néhány olyan STL-beli algoritmust, amelyek predikátumok felhasználásával működnek.

Feltételes csere (*replace_if*)

```
template <class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last, Predicate
pred,
              const T& new_value) {
      while (first != last) {
            if (pred(*first)) *first = new_value;
            ++first;
      }
}
```

A *replace_if* algoritmus egy adott konténer bizonyos elemeit egy másik elemre cseréli. Az algoritmusnak a következő paramétereket kell megadni: az első elemre mutató iterátor, az utolsó elem utánra mutató iterátor, a használni kívánt predikátum és a helyettesítő érték, amire le akarjuk cserélni a feltételt kielégítő elemeket.
A következő példában egy egész értékeket tároló tömbnek a 15-nél nagyobb elemeit 3-as értékekkel helyettesítjük.

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

// A predikatum. Igaz ha x > 15.
bool gt15(int x) {
      return x > 15;
}

int main() {
      int a[] = {10, 20, 30};
      const size_t S = sizeof a / sizeof a[0];
      replace_if(a, a+S, gt15, 3);
      copy(a, a+S, ostream_iterator<int>(cout," "));
```

 © Ferenc Rudolf, SzTE

```
	cout << endl;
	return 0;
}
```

A program futtatása után a következő eredményt kapjuk:

```
10 3 3
```

Ha megvizsgáljuk a használt *replace_if* algoritmust, akkor kiderül, hogy a helyes működéséhez az átadott predikátumnak a következő követelményeket kell kielégítenie: *bool* típusra konvertálható típussal kell visszatérnie, függvényként hívhatónak kell lennie (mert használva van a függvényhívás operátor), valamint csak egy paramétere lehet.

Feltételes másolás cserével (*replace_copy_if*)

```
template<class InputIterator, class OutputIterator, class Predicate, class
T>
OutputIterator replace_copy_if(InputIterator first, InputIterator last,
                               OutputIterator result, Predicate pred,
                               const T& new_value) {
	while (first != last) {
		*result++ = pred(*first) ? new_value : *first;
		++first;
	}
	return result;
}
```

A gyakorlatban sokszor szükség lehet arra, hogy az eredeti tömb is a rendelkezésünkre álljon, és egy olyan is, ahol bizonyos elemeket más elemekkel helyettesítettünk. Ennek a második tömbnek az elkészítésében segít a *replace_copy_if* algoritmus, miközben az eredetit változatlanul hagyja. A paraméterezése annyiban tér el a *replace_if* algoritmusétól, hogy a másolandó konténer vége után mutató iterátort követően meg kell adni a célkonténer elejére mutató iterátort is.
Nézzünk egy példát a *replace_copy_if* algoritmus használatára!

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

// predikatum
bool gt15(int x) {
	return x > 15;
}

int main() {
	int a[] = {10, 20, 30};
	const size_t S = sizeof a / sizeof a[0];
	int b[S];
	int* endb = replace_copy_if(a, a+S, b, gt15, 3);
	copy(b, endb, ostream_iterator<int>(cout," "));
	cout << endl;
	return 0;
}
```

A program futtatása után a következő eredményt kapjuk:

```
10 3 3
```

Feltételes másolás (*remove_copy_if*)

```
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result, UnaryPredicate pred) {
	while (first != last) {
		if (!pred(*first))
			*result++ = *first;
		++first;
	}
	return result;
}
```

Az előzőleg használt *replace_copy_if* algoritmus a feltételt kielégítő elemeket egy másikkal helyettesítette a cél sorozatban. Ha azonban a feltételt kielégítő elemeket egyáltalán nem kívánjuk átmásolni az új sorozatba, akkor a *remove_copy_if* algoritmust kell használni. A paraméterezése nagyon hasonló a *replace_copy_if* paraméterezéséhez, csak a helyettesítő értéket természetesen nem kell megadni.

Nézzünk egy példát a *remove_copy_if* algoritmus használatára!

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

// predikatum
bool gt15(int x) {
	return x > 15;
}

int main() {
	int a[] = {10, 20, 30};
	const size_t S = sizeof a / sizeof a[0];
	int b[S];
	int* endb = remove_copy_if(a, a+S, b, gt15);
	int* beginb = b;
	copy(beginb, endb, ostream_iterator<int>(cout," "));
	cout << endl;
	return 0;
}
```

A program futtatása után a következő eredményt kapjuk:

```
10
```


© Ferenc Rudolf, SzTE

Függvény objektumok

Az előző példák közös hibája a mágikus konstans használata. Ez a mágikus konstans esetükben a 15, ami még a függvény nevében is szerepel. Azonban korántsem biztos, hogy csak erre az értékre lesz szükségünk. Előfordulhat például, hogy gt20, gt25, stb. függvényeket is le kell majd implementálni. Ekkor több probléma is felmerül. Egyrészt ha sok hasonló függvényt készítünk, akkor az implementálás sok időt vesz igénybe, másrészt, ha kiderül, hogy valahol valamit elrontottunk benne, akkor a hibát az összes helyen ki kell javítani. Ezen felül a függvényekhez szükséges összes értéket fordítási időben ismerni kell. Mivel a predikátumoknak csak egy paraméterük lehet, az sem működik, hogy a küszöbértéket egy második paraméterben adjuk át a predikátumnak.
A megoldást a *függvény objektum*ok (*function object*s) jelentik. A függvény objektumok olyan osztályok példányai, amik kiterjesztik a függvényhívás operátort – () –, így használhatóak függvényekként. Ezeket az objektumokat – mint minden más objektumot – a konstruktoruk segítségével egyszerűen inicializálhatjuk.
A következő példakód az előzőekben megvizsgált csere nélküli feltételes másolást valósítja meg függvény objektum használatával.

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

// fuggveny objektum
class gt_n {
	int ertek;
public:
	gt_n(int i) : ertek(i) {}	//konstruktor a küszöbérték
inicializáláshoz
	bool operator()(int x) {	//a függvényhívás operátor kiterjesztése
		return x > ertek;
	}
};

int main() {
	int a[] = {10, 20, 30};
	const size_t S = sizeof a / sizeof a[0];
	int b[S];
	int* endb = remove_copy_if(a, a+S, b, gt_n(15));
	copy(b, endb, ostream_iterator<int>(cout," "));
	cout << endl;
	return 0;
}
```

A *remove_copy_if* algoritmusnak átadott *gt_n(15)* argumentum kiértékelése során létrejön egy temporális objektum veremben a *gt_n* osztályból, majd meghívódik a konstruktora, ahol az adattag értéke 15-re inicializálódik. Nevezzük ezt az objektumot *x*-nek. Valójában nem kap semmilyen nevet, de így tudunk rá a továbbiakban hivatkozni. Ezután a *remove_copy_if* algoritmuson belül (lásd a megvalósítását az előző fejezetben), minden iterációban ennek az *x* objektumnak a függvényhívás operátora fog meghívódni. A *pred(*first)* híváskor tulajdonképpen az *x.operator()(*first)* függvényhívás fog lefutni, az összehasonlítandó érték helyébe mindig az aktuálisan vizsgált elemet helyettesítve. Tehát az aktuálisan vizsgált elemet fogja összehasonlítani az általa eltárolt értékkel, jelen esetben a 15-tel.

A program futtatása után a következő eredményt kapjuk:

```
10
```

Függvény objektum adapterek

Az STL több hasznos függvény objektumot is tartalmaz. Ezek meglehetősen egyszerűek, de kombinálásukkal összetett predikátumok is készíthetők. Ezen függvények kombinálása a *függvény objektum adapter*ek (*function object adapter*s) használatával lehetséges. Ahhoz, hogy ezeket a beépített függvény objektumokat és függvény objektum adaptereket használni tudjuk, be kell *include*-olni a <*functional*> header fájlt.
Az STL-ben nem csak olyan függvény objektumok léteznek, amelyek egy paramétert várnak és *bool* értékkel térnek vissza. Az STL a bemenő paraméterek száma, és a visszaadott érték alapján több csoportba rendezi a függvény objektumokat. Ezek alapján a következő csoportok különíthetőek el:

- A *generátor*oknak nincs semmilyen bejövő paraméterük, és egy tetszőleges típusú értékkel térnek vissza.
- Az *unáris függvény*ek egy paramétert várnak, visszatérési értékük pedig bármi lehet (akár *void* is). Ha az unáris függvény *bool* értékkel tér vissza, akkor *unáris predikátum*nak nevezzük.
- A *bináris függvény*ek már két paramétert várnak, és az unáris függvényekhez hasonlóan bármilyen visszatérési értékük lehet (akár *void* is). Ha egy bináris függvény *bool* értékkel tér vissza, akkor *bináris predikátum*nak nevezzük.

Nézzünk egy példa programot a függvény objektum adapterek használatára! A következőkben továbbra is maradunk a csere nélküli feltételes másolásnál, azonban a predikátum, ami megvizsgálja, hogy az aktuális érték nagyobb-e, mint 15, beépített függvény objektumok és függvény adapterek használatával kerül implementálásra.

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

int main() {
	int a[] = {10, 20, 30};
	const int S = sizeof a / sizeof a[0];
	remove_copy_if(a, a+S, ostream_iterator<int>(cout," "),
	              bind2nd(greater<int>(),15));
	cout << endl;
	return 0;
}
```

A nekünk megfelelő STL-beli függvényobjektum a *greater*. Ez egy bináris predikátum, amely akkor ad vissza igaz értéket, ha az első argumentuma nagyobb, mint a második. Ezt azonban így nem tudjuk használni, hiszen a *remove_copy_if* algoritmus unáris predikátumot vár. Ehhez le kellene rögzítenünk a *greater* egyik paraméterét, jelen esetben a másodikat. Erre való a *bind2nd* függvény objektum adapter. Ez két paramétert vár, elsőként egy bináris függvény objektumot vagy függvényt kell neki megadni (amit lehet két paraméterrel hívni), második paraméterként pedig a rögzíteni kívánt értéket kéri. A *bind2nd* a megadott paraméterekből

 © Ferenc Rudolf, SzTE

készít egy *binder2nd* függvény objektumot. A *binder2nd* eltárolja a függvényt vagy függvény objektumot és a második paraméter rögzített értékét. A *binder2nd* függvényhívás operátora pedig egy unáris operátor, ami alkalmazza az eltárolt függvényt/függvény objektumot, átadva neki a kapott paramétert és a tárolt értéket.
A példaprogram futása a következőképpen történik. (A továbbiakban jelölje a létrejövő *binder2nd* objektumot *b*, az egész értékekre példányosított *greater<int>* objektumot *g*, az aktuálisan vizsgált elemet pedig *e*.) A *remove_copy_if* algoritmus használatakor a *bind2nd(g,15)* létrehozza a *b* objektumot, a megfelelő mezőit *g*-re és 15-re állítva, azáltal, hogy meghívja a konstruktorát a *g* és 15 paraméterekkel. A *remove_copy_if* metóduson belül ezután minden iterációban a *b(e)* függvényhívás hajtódik végre (tulajdonképpen a *b.operator()(e)*). Ez pedig meghívja a tárolt függvényt, tehát *g*-t, az *e*-re és az eltárolt értékre, vagyis 15-re. Így végső soron minden iterációs lépésben a *g(e,15)* bináris predikátum fog kiértékelődni. Ez pedig pontosan az, amire nekünk szükségünk van.
A program futtatása után a következő eredményt kapjuk:

```
10
```

Az STL nem csak a második argumentum lekötését teszi lehetővé, hanem az elsőét is. Ehhez a *bind1st* függvényt használhatjuk, ami egy *binder1st* függvény objektumot készít. Ezek működése megegyezik a *bind2nd* függvény és *binder2nd* függvény objektum működésével, azzal a különbséggel, hogy az első értéket fixálják le.

A következő táblázat mutatja az STL beépített függvény objektumait.

Név	**Típus**	**Eredmény**
plus	BinaryFunction	arg1 + arg2
minus	BinaryFunction	arg1 - arg2
multiplies	BinaryFunction	arg1 * arg2
divides	BinaryFunction	arg1 / arg2
modulus	BinaryFunction	arg1 % arg2
negate	UnaryFunction	- arg1
equal_to	BinaryPredicate	arg1 == arg2
not_equal_to	BinaryPredicate	arg1 != arg2
greater	BinaryPredicate	arg1 > arg2
less	BinaryPredicate	arg1 < arg2
greater_equal	BinaryPredicate	arg1 >= arg2
less_equal	BinaryPredicate	arg1 <= arg2
logical_and	BinaryPredicate	arg1 && arg2
logical_or	BinaryPredicate	arg1 || arg2
logical_not	UnaryPredicate	!arg1
unary_negate	UnaryPredicate	!(UnaryPredicate(arg1))
binary_negate	BinaryPredicate	!(BinaryPredicate(arg1, arg2))

Adaptálható függvény objektumok

Ahhoz, hogy megértsük a *binder2nd* algoritmus pontos működését, ismerkedjünk meg a *binder2nd* algoritmus forráskódjával, ami a következőképpen néz ki:

```
template <class Operation>
class binder2nd : public unary_function<Operation::first_argument_type,
                                        Operation::result_type> {
protected:
      Operation op;
      Operation::second_argument_type value;
public:
      binder2nd(const Operation& x, const Operation::second_argument_type&
y)
            : op(x), value(y) {}
      result_type operator()(const argument_type& x) const {
            return op(x, value);
      }
};
```

Amennyiben olyan saját függvény objektumot akarunk készíteni, amellyel a függvény objektum adapterek dolgozni tudnak, definiálnunk kell bizonyos típusokat. Ezek a típusok unáris függvény objektumok esetén az *argument_type* és a *result_type*. Az első a bemenő paraméter, a második pedig a visszaadott érték típusát határozza meg. Bináris függvényobjektum esetén a *first_argument_type*, a *second_argument_type* és a *result_type* típusokat kell definiálni. Ezek az első bemenő paraméter, a második bemenő paraméter és a visszaadott érték típusait határozzák meg.
Mint látható a kódban, a *binder2nd* használja a kapott bináris függvény objektum első és második paraméterének, valamint a visszatérési értékének a típusát is, valamint a függvényhívás operátor kiterjesztésekor az unáris függvény paraméterének és visszatérési értékének a típusával dolgozik. Mivel ezeket az adatokat használja, világos, hogy amennyiben ezeket egy általunk írt függvény objektum esetén nem definiáljuk, akkor a *bind2nd* nem lesz képes vele dolgozni, nem fordul le a kódunk. (A *binder1st* teljesen hasonlóan működik.)
A kódban észrevehetjük, hogy a *binder2nd* egy *unary_function* sablonosztályból származik, amely sablon paraméterként a bemenő paraméter és a visszatérési érték típusát kéri. Ez az osztály tulajdonképpen csupán annyit tud, hogy definiálja a szükséges típusneveket, ezáltal megkönnyítve az unáris függvény objektumok létrehozását.
Az *unary_function* kódja:

```
template <class Arg, class Result>
struct unary_function {
      typedef Arg argument_type;
      typedef Result result_type;
};
```

Természetesen a bináris függvény objektumokhoz is létezik ilyen „segítő” osztály, ez a *binary_function* osztály, aminek a megvalósítása a következő:

```
template <class Arg1, class Arg2, class Result>
struct binary_function {
      typedef Arg1 first_argument_type;
      typedef Arg2 second_argument_type;
      typedef Result result_type;
};
```

Ezek segítségével könnyen elkészíthetjük saját függvény objektumainkat, valamint az STL beépített függvény objektumai is ezekből származnak. Például, a korábban használt *greater* megvalósítása a következőképpen néz ki:

 © Ferenc Rudolf, SzTE

```
template <class T>
struct greater : binary_function<T, T, bool> {
      bool operator()(const T& x, const T& y) const { return x > y; }
};
```

Ha a *greater*-t *int*-tel példányosítjuk (mint azt a korábbi példában tettük), akkor a *binary_function<int, int, bool>* osztályból fog származni, ami definiálja a *first_argument_type*-ot és a *second_argument_type*-ot *int*-re, a *result_type*-ot pedig *bool*-ra. Innen pedig a *binder2nd*-nek egyértelmű, hogy egy olyan bináris predikátumot kapott, ami két darab *int* értéket vár paraméterként, a kiterjesztett függvényhívás operátoránál pedig egy *int* értéket kell várnia, és egy *bool* értéket fog visszaadni. Innentől kezdve pedig a korábban bemutatott módon zajlik a példányosítás és a függvényhívás.

Függvény pointer adapterek

Az STL algoritmusok, amikor egy függvényt/predikátumot várnak paraméterül, mind függvény objektumot, mind függvény pointert elfogadnak. A függvény pointerek esetében azonban nem használhatóak a függvény objektum adapterek, mert ezek feltételezik a megfelelő típusdefiníciók meglétét. Azonban, ha saját függvényünkön akarjuk használni a függvény objektum adaptereket, nem kell magunknak kézzel átalakítani azt függvény objektummá, ugyanis a *ptr_fun* adapterek pontosan erre lettek kitalálva.
A *ptr_fun* adapterek egy függvény pointert várnak paraméterül, amit adaptálható függvény objektummá alakítanak. Fontos tudni azonban, hogy a *ptr_fun* adapterek csak olyan függvényeket képesek függvény objektummá alakítani, amelyek egy vagy két bemenő paramétert várnak, tehát generátor típusú függvény objektumokat nem készíthetünk ezek segítségével. A *ptr_fun* először egy unáris vagy egy bináris függvényre mutató pointer objektumot készít (*pointer_to_unary_function* vagy *pointer_to_binary_function*), amely már az *unary_function*, illetve a *binary_function* osztályból származik, így használható a függvény objektum adapterek számára.
Nézzünk egy példa programot a *ptr_fun* adapterek használatára!

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



// predikatum
bool paros(int x) {
       return x % 2 == 0;
}

int main() {
      int d[] = {123, 94, 10, 314, 315};
      const size_t S = sizeof d / sizeof d[0];
      vector<bool> vb;
      transform(d, d+S, back_inserter(vb), not1(ptr_fun(paros)));
      copy(vb.begin(), vb.end(), ostream_iterator<bool>(cout, " "));
      cout << endl;
```

```
	return 0;
}
```

A példakód egy egész értékeket tároló tömböt alakít át egy *bool* értékeket tároló *vector*-rá, az alapján, hogy az adott pozíción álló szám páratlan-e. Erre a feladatra a *transform* algoritmus megfelelő. A paritást eldöntő predikátum pedig egyszerű függvény, nem függvény objektum. A program futtatása után a következő eredményt kapjuk:

```
1 0 0 0 1
```

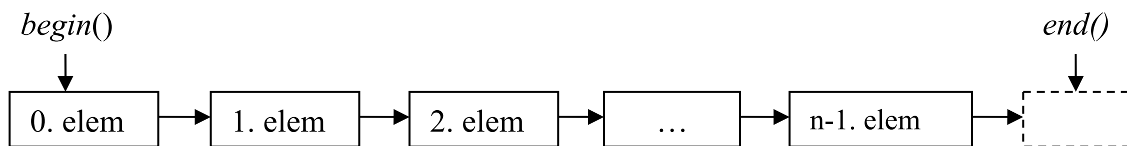
 © Ferenc Rudolf, SzTE

GENERIKUS ALGORITMUSOK

A standard könyvtár beépített tárolók mellett beépített algoritmusokat is tartalmaz, amik segítségével a tárolók igen hasznossá válnak, mert a segítségükkel könnyen végrehajthatók a legáltalánosabb műveletek is. Ezen algoritmusok segítségével bejárhatjuk a tárolót, lekérdezhetjük a méretét, másolhatunk, törölhetünk elemeket, rendezhetjük, összehasonlíthatjuk, vagy elemeket kereshetünk benne. Minden algoritmus egy sablonfüggvény, ezzel téve lehetővé, hogy többféle elemsorozatra is alkalmazható legyen. A szabvány algoritmusok kivétel nélkül az *std* névtérben találhatók és az *<algorithm>* fejállomány tartalmazza őket. Az algoritmusokat két nagy csoportra lehet bontani: a sorozatot módosító és nem módosító algoritmusokra. A legtöbb algoritmus lehetőséget ad a fejlesztőnek, hogy maga határozza meg azt a feladatot, amelyet minden elemen, illetve elempáron végre szeretne hajtani (pl. a programozó döntheti el, hogy két elemet mikor tekint egyenlőnek). Ezáltal az algoritmusok nagyon általánosak és hasznosak, mivel sok felesleges munkától kímélhetik meg a fejlesztőt és nem mellesleg javítják a kód olvashatóságát is.

Iterátorok

A standard könyvtárban található algoritmusok elemek sorozatán hajtanak végre valamilyen műveletet. Az ilyen sorozatok legegyszerűbben iterátor párokkal (*begin* és *end*) ábrázolhatók, melyek az első és az utolsó utáni elemet adják vissza. A következő ábrán látható a sorozat fogalmának grafikus ábrázolása:



A legtöbb algoritmus a tárolók bejárása céljából iterátorokkal dolgozik. Iterátornak tekinthető minden olyan objektum, ami iterátorként képes viselkedni. Az iterátor tekinthető egy sorozat elemeire mutató pointernek is. Három fő műveletet kell teljesítenie: az éppen kijelölt elemet megadni (dereferencia/indirekció: *, ->), a következő elemre lépni (léptetés: ++), és az egyenlőséget (==) megállapítani. Az iterátor osztályok az *std* névtérhez tartoznak és az *<iterator>* fejállományban találhatók. Nem minden iterator engedi meg pontosan ugyanannak a művelethalmaznak a használatát. Az olvasáshoz például másfajta műveletekre van szükség, mint az íráshoz. A vektorok lehetővé teszik, hogy bármikor bármelyik elemet kényelmesen és hatékonyan elérhessük, míg a listáknál és adatfolyamoknál ez a művelet rendkívül költséges. Ezért az iterátorokat öt osztályba soroljuk, aszerint, hogy milyen műveleteket képesek hatékonyan (azaz konstans idő alatt) megvalósítani:

- InputIterator: elemek olvasása egy irányban,
- OutputIterator: elemek írása egy irányban,
- ForwardIterator: elemek olvasása és írása egy irányban,
- BidirectionalIterator: elemek olvasása és írása mindkét irányban,
- RandomAccessIterator: elemek olvasása és írása direkt eléréssel, teljes pointer aritmetika megvalósítása.

A következő táblázat megmutatja, mely iterátor kategória milyen műveleteket képes végrehajtani:

Kategóra:	író (kimeneti)	olvasó (bemeneti)	előre haladó	kétirányú	közvetlen elérésű
Olvasás:		=*p	=*p	=*p	=*p
Hozzáférés		->	->	->	-> [ ]
Írás:	*p=		*p=	*p=	*p=
Haladás:	++	++	++	++ --	+ - += -=
Összehasonlítás:		== !=	== !=	== !=	== != < > <= >=

Egy iterátornak kategóriájától függetlenül lehetővé kell tennie mind a konstans, mind a nem konstans elérést arra az objektumra, amelyre mutat. Egy *const_iterator*-on keresztül az elemet nem változtathatjuk meg, legyen az bármilyen kategóriájú.
A jegyzetben nem adunk teljes áttekintést az STL-beli algoritmusokról, csak a leggyakrabban előforduló algoritmusokat mutatjuk be. Az algoritmusok mind sablonfüggvények, amint arra a korábbiakban már láttunk is példákat, azonban a következőkben az egyszerűség kedvéért eltekintünk a sablon paraméterlisták kiírásától.

Feltöltés és generálás

A *fill* és a *generate* algoritmusok lehetővé teszik egy adott tartomány rendszerezett feltöltését. Mindkét algoritmus egyszerű értékadást valósít meg.

```
fill(ForwardIterator first, ForwardIterator last, const T& value);
fill_n(OutputIterator first, Size n, const T& value);
```

Megkülönböztethetünk *fill* és *fill_n* algoritmusokat, ahol mindkettő egy adott sorozat minden elemét a harmadik paraméterben megadott értékre állítja, azzal a különbséggel, hogy a *fill* függvény a sorozat elejét és végét várja paraméterül, míg a *fill_n* a sorozat elejét és a feltöltésre váró elemek számát (*n*).

```
generate(ForwardIterator first, ForwardIterator last, Generator gen);
generate_n(OutputIterator first, Size n, Generator gen);
```

Hasonlóképpen megkülönböztethetünk *generate* és *generate_n* algoritmusokat, ahol mindkettő egy adott sorozat minden eleménél meghívja a harmadik paraméterben átadott függvényt és a visszaadott értéket állítja be a soron következő elemnek, azzal a különbséggel, hogy a *generate* függvény a sorozat elejét és végét várja paraméterül, míg a *generate_n* a sorozat elejét és a feltöltésre váró elemek számát (*n*).

Az imént ismertetett algoritmusok használatára nézzünk egy példát! A példa vektorokat tölt fel konkrét elemekkel, valamint függvénnyel generált elemekkel. Először is valósítsuk meg a vektor elemeinek kiíratását egy *print* függvénnyel:

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

 © Ferenc Rudolf, SzTE

```
template<typename Iterator>
void print(Iterator first, Iterator last, string cont) {
   cout << cont << ": ";
   while (first != last)
      cout << *first++ << " ";
   cout << endl;
}
```

Ezután implementáljunk egy függvényobjektumot, amely egy számtani sorozat következő elemét adja vissza!

```
class UgroGen {
   int i;
   int ugras;
public:
   UgroGen(int ugras) : i(0), ugras(ugras) {}
   int operator()() {
      int j = i;
      i += ugras;
      return j;
   }
};
```

Végül futtassuk az algoritmusokat egy *main* függvény megvalósítással:

```
int main() {
   vector<string> v1(5);
   fill(v1.begin(), v1.end(), "ha");
   print(v1.begin(), v1.end(), "v1");

   vector<string> v2;
   fill_n(back_inserter(v2), 7, "hi");
   print(v2.begin(), v2.end(), "v2");

   vector<int> v3(10);
   generate(v3.begin(), v3.end(), UgroGen(2));
   print(v3.begin(), v3.end(), "v3");

   vector<int> v4;
   generate_n(back_inserter(v4), 12, UgroGen(3));
   print(v4.begin(), v4.end(), "v4");

   return 0;
}
```

A program futtatása után a következő eredményt kaptuk:

```
v1: ha ha ha ha ha
v2: hi hi hi hi hi hi hi
v3: 0 2 4 6 8 10 12 14 16 18
v4: 0 3 6 9 12 15 18 21 24 27 30 33
```

Számlálás

Minden konténernek van saját *size* nevű függvénye, amely megmondja, éppen hány elemet tárol. Ha valamilyen feltételnek eleget tevő elemek számára vagyunk kíváncsiak, akkor a *count*, illetve a *count_if* STL-beli algoritmusokat használhatjuk.

```
size_t count(InputIterator first, InputIterator last, const T& value);
size_t count_if(InputIterator first, InputIterator last, Predicate pred);
```

A *count* megszámolja, hány olyan elem van a sorozatban, amely megegyezik a harmadik paraméterben szereplő *value* értékkel. A *count_if* harmadik paraméterként nem egy konstans értéket, hanem egy predikátumot vár, így azon elemek számát adja az algoritmus, melyekre a predikátum teljesül.

Az imént ismertetett algoritmusok használatára nézzünk egy példát! Írjunk programot, mely tetszőleges karakterláncban megszámolja az előforduló karaktereket, hogy melyikből hány darab van, megszámolja a kisbetűket és kiírja sorba rendezve a karaktereket! Ehhez írjunk saját függvényobjektumot, amely egy véletlen karaktert ad visszatérési értékként!

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <vector>
#include <set>
#include <string>
#include <ctime>
using namespace std;

class CharGen {
   static const char* source;
   static const int len;
public:
   CharGen() { srand(time(0)); }
   char operator()() { return source[rand() % len]; }
};
const char* CharGen::source = "ABCDEFGHIJabcdefghij";
const int CharGen::len = strlen(source);

int main() {
   vector<char> v;
   generate_n(back_inserter(v), 16, CharGen());
   print(v.begin(), v.end(), "v");

   set<char> cs(v.begin(), v.end());
   for(set<char>::iterator it = cs.begin(); it != cs.end(); it++)
      cout << *it << ":" << count(v.begin(), v.end(), *it) << ", ";

   int k = count_if(v.begin(), v.end(), bind2nd(greater_equal<char>(),
   'a'));
   cout << endl << "Kisbetuk szama: " << k << endl;

   sort(v.begin(), v.end());
   print(v.begin(), v.end(), "sorban");

   return 0;
}
```

 © Ferenc Rudolf, SzTE

A back_inserter(v) hívás egy speciális beszúró iterátort készít, ami minden értékadás alkalmával felülírás helyett egy új elemet szúr be a sorozat végére és abba írja be az értéket (erről a későbbiekben még lesz szó). Azt, hogy kisbetűs-e, egyszerűen úgy döntjük el, hogy nagyobb-e vagy egyenlő a karakter ASCII kódja az *a* karakterénél.
A program futtatása után pl. a következő eredményt kapjuk:

```
v: d C a j a C i j G h d b A C I d
A:1, C:3, G:1, I:1, a:2, b:1, d:3, h:1, i:1, j:2,
Kisbetuk szama: 10
sorban: A C C C G I a a b d d d h i j j
```

Sorozatok manipulálása

Ebben a fejezetben egy konténer elemeinek manipulálásáról lesz szó: hogyan lehet egy konténer elemeit másolni, megfordítani, rotálni, cserélni más konténer elemeivel.

```
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator
dest);
```

A *copy* algoritmus átmásolja az első sorozat összes elemét a harmadik paraméter által mutatott helytől kezdődően. A kimenetnek nem feltétlenül kell tárolónak lennie, bármi megadható, amihez kimeneti bejáró megadható (pl. *ostream_iterator*).

```
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
                                     BidirectionalIterator1 last,
                                     BidirectionalIterator2 destEnd);
```

A *copy* függvény egy másik variációja a *copy_backward*, ahol a cél végét megadó iterátort kell átadni az eljárásnak és visszafelé írja be az elemeket (ám az elemek sorrendje változatlan marad).

```
OutputIterator reverse_copy(BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator dest);
```

Ha az elemek sorrendjét fel szeretnénk cserélni, akkor a *reverse_copy* algoritmust kell használni. Ez a függvény a *copy*-hoz hasonlóan másolja át az elemeket, azonban fordított sorrendben.

```
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                           ForwardIterator last, OutputIterator dest);
```

A *rotate_copy* algoritmus úgy másolja át az elemeket a negyedik paraméter által mutatott helytől kezdődően, hogy a második paraméterben megadott középső elemtől (*middle*) jobbra, illetve balra eső részsorozatokat felcseréli.

A következő algoritmusok a sorozatokat helyben manipulálják, nem marad meg az eredeti sorozat, az új sorozat felülírja a régit.

```
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

A *reverse* algoritmus helyben megfordítja az elemek sorrendjét.

```
void rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator
last);
```

A *rotate* algoritmus helyben felcseréli a második paraméterben megadott középső elemtől jobbra, illetve balra eső részsorozatokat.

```
ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1
last1,
                             ForwardIterator2 first2);
```

Végül a *swap_ranges* algoritmus két sorozat elemeinek felcseréléséért felelős.

További sorozat manipuláló algoritmusok:

- *next_permutation, prev_permutation*
- *random_shuffle*
- *partition, stable_partition*

Az imént ismertetett algoritmusok használatára nézzünk egy példát!

```
int main() {
   vector<int> v1(8);
   generate(v1.begin(), v1.end(), UgroGen(2));
   print(v1.begin(), v1.end(), "v1");

   vector<int> v2(v1.size());
   copy_backward(v1.begin(), v1.end(), v2.end());
   print(v2.begin(), v2.end(), "v2 (copy_backward)");

   reverse_copy(v1.begin(), v1.end(), v2.begin());
   print(v2.begin(), v2.end(), "v2 (reverse_copy)");

   reverse(v1.begin(), v1.end());
   print(v1.begin(), v1.end(), "v1 (reverse)");

   int half = v1.size() / 2;
   swap_ranges(v1.begin(), v1.begin() + half, v1.begin() + half);
   print(v1.begin(), v1.end(), "v1 (swap_ranges)");

   generate(v1.begin(), v1.end(), UgroGen(2));
   print(v1.begin(), v1.end(), "v1");

   int third = v1.size() / 3;
   rotate(v1.begin(), v1.begin() + third, v1.end());
   print(v1.begin(), v1.end(), "v1 (rotate)");

   return 0;
}
```

A program futtatása után a következő eredményt kaptuk:

```
v1: 0 2 4 6 8 10 12 14
v2 (copy_backward): 0 2 4 6 8 10 12 14
v2 (reverse_copy): 14 12 10 8 6 4 2 0
v1 (reverse): 14 12 10 8 6 4 2 0
v1 (swap_ranges): 6 4 2 0 14 12 10 8
v1: 0 2 4 6 8 10 12 14
v1 (rotate): 4 6 8 10 12 14 0 2
```

 © Ferenc Rudolf, SzTE

Keresés és csere

Ha sorozatokkal dolgozunk, szükség lehet elemek keresésére és cseréjére. A keresésekben és cserékben a következő STL szabvány algoritmusok segítenek.

```
InputIterator find(InputIterator first, InputIterator last,
                   const EqualityComparable& value);
```

A *find* algoritmus megkeresi az első *value* értékű elemet a [*first,last*) intervallumban.

```
InputIterator find_if(InputIterator first, InputIterator last, Predicate
pred);
```

A *find_if* algoritmus megkeresi azt az első elemet a [*first,last*) intervallumban, amire a *pred* predikátum teljesül.

```
ForwardIterator1 find_first_of(ForwardIterator1 first1, ForwardIterator1
last1,
                               ForwardIterator2 first2, ForwardIterator2
last2);
```

A *find_first_of* algoritmus megkeresi a [*first1,last1*) intervallumban a [*first2,last2*) intervallumban szereplő valamelyik elem első előfordulását.

```
ForwardIterator1 find_first_of(ForwardIterator1 first1, ForwardIterator1
last1,
                               ForwardIterator2 first2, ForwardIterator2
last2,
                               BinaryPredicate binary_pred);
```

A *find_first_of* algoritmus egy másik variációja hasonló az előzőhöz, csak egyenlőség helyett az ötödik paraméterben megadott predikátum alapján dönt.

```
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2);
```

A *search* algoritmus megkeresi a [*first1,last1*) intervallumban a [*first2,last2*) részsorozat első előfordulását.

```
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2,
                        BinaryPredicate binary_pred);
```

A *search* algoritmus másik variációja megkeresi a [*first1,last1*) intervallumban a [*first2,last2*) részsorozat első olyan előfordulását, amire a megadott predikátum teljesül.

```
ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2);
```

A *find_end* algoritmus megkeresi a [*first1,last1*) intervallumban a [*first2,last2*) részsorozat utolsó előfordulását.

```
ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2,
```

```
                                    BinaryPredicate binary_pred);
```

A *find_end* algoritmus másik változata megkeresi a [*first1,last1*) intervallumban a [*first2,last2*) részsorozat utolsó olyan előfordulását, amire a megadott predikátum teljesül.

További kereső és cserélő algoritmusok:

- *adjacent_find, search_n*
- *min_element, max_element*
- *replace_if, replace_copy, replace_copy_if*

Az imént ismertetett algoritmusok használatára nézzünk egy példát!

```
int main() {
   int a[] = {1,2,3,4,5,6,6,7,7,7,8,8,8,8,11,11,11};
   vector<int> v(a,a+(sizeof a/sizeof *a));
   print(v.begin(), v.end(), "v");

   vector<int>::iterator it = find(v.begin(), v.end(), 4);
   cout << "find: " << *it << endl;

   it = find_if(v.begin(), v.end(), bind2nd(greater<int>(), 8));
   cout << "find_if: " << *it << endl;

   int b[] = {8,11};
   const int bs = (sizeof b/sizeof *b);
   print(b, b+bs, "b");

   it = find_first_of(v.begin(), v.end(), b, b+bs);
   cout << "find_first_of: " << *it << endl;

   it = search(v.begin(), v.end(), b, b+bs);
   print(it, it+bs, "search");

   int c[] = {11,11};
   const int cs = sizeof c / sizeof *c;
   print(c, c+cs, "c");

   it = find_end(v.begin(), v.end(), c, c+cs);
   print(it, v.end(),"find_end");

   return 0;
}
```

A program futtatása után a következő eredményt kapjuk:

```
v: 1 2 3 4 5 6 6 7 7 8 8 8 11 11 11
find: 4
find_if: 11
b: 8 11
find_first_of: 8
search: 8 11
c: 11 11
find_end: 11 11
```

Összehasonlítás

Sorozatok összehasonlítására is léteznek szabványos algoritmusok mind az ==, mind pedig a <, > relációk eldöntésére. Mindhárom esetben saját predikátum osztály segítségével meghatározható, hogyan legyenek értelmezve az összehasonlítás műveletek a sorozat elemein.

```
bool equal(InputIterator first1, InputIterator last1, InputIterator
first2);
```

Az *equal* algoritmus megmondja, hogy a [*first1,last1*) és [*first2,...*) sorozatok megegyeznek-e.

```
bool equal(InputIterator first1, InputIterator last1,InputIterator first2,
           BinaryPredicate binary_pred);
```

Az *equal* algoritmus egy másik változata összehasonlítja a [*first1,last1*) és [*first2,...*) sorozatokat, de a == helyett a megadott *binary_pred* predikátum alapján dolgozik.

```
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2);
```

A *lexicographical_compare* algoritmus igazzal tér vissza, ha az első sorozat lexikografikusan kisebb, mint a második sorozat (lexikografikus sorrend például sztringek esetében az ABC sorrend).

```
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, BinaryPredicate binary_pred);
```

A *lexicographical_compare* algoritmus másik változata, amely a < reláció helyett a *binary_pred* predikátum alapján dönt.

További összehasonlító algoritmusok:

- *mismatch*

Az imént ismertetett algoritmusok használatára nézzünk egy olyan példát, ami összehasonlít két sztringet!

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

int main() {
   string s1("Ez egy teszt");
   string s2("Ez egy Teszt");
   cout << "s1: " << s1 << endl << "s2: " << s2 << endl;
   cout << "equal s1 & s1: " << equal(s1.begin(), s1.end(), s1.begin()) <<
   endl;
   cout << "equal s1 & s2: " << equal(s1.begin(), s1.end(), s2.begin()) <<
   endl;
   cout << "lexicographical_compare s1 & s1: " <<
      lexicographical_compare(s1.begin(), s1.end(), s1.begin(), s1.end())
   << endl;
   cout << "lexicographical_compare s1 & s2: " <<
      lexicographical_compare(s1.begin(), s1.end(), s2.begin(), s2.end())
   << endl;
```

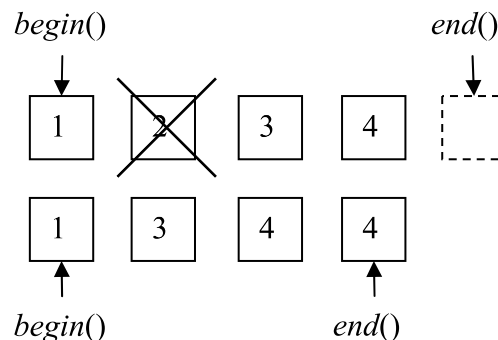
```
   cout << "lexicographical_compare s2 & s1: " <<
      lexicographical_compare(s2.begin(), s2.end(), s1.begin(), s1.end())
   << endl;
   return 0;
}
```

A program futtatása után a következő eredményt kaptuk:

```
s1: Ez egy teszt
s2: Ez egy Teszt
equal s1 & s1: 1
equal s1 & s2: 0
lexicographical_compare s1 & s1: 0
lexicographical_compare s1 & s2: 0
lexicographical_compare s2 & s1: 1
```

Elemek törlése

Az STL-beli általános algoritmusok iterátorokat használnak, így nincs is tudomásuk arról, hogy a sorozatok amiken dolgoznak, milyen fizikai adatszerkezetben léteznek (pl. láncolt lista, bináris fa), így a törlő algoritmusok igazából nem törölnek fizikailag sorozat elemeket , mert nem is tudhatják, hogy hogyan kellene azt végrehajtani. Ehelyett az történik, hogy a törlendő elemek helyére csúsztatja a sorozat további részét és visszaadja az új *end* iterátort, ami így a régi sorozat egy elemére fog mutatni, de az új, rövidebb sorozatban ez már az érvényes elemek közül az utolsó utáni elemre mutat. Ezt szemlélteti a következő ábra:



Tekintsük át a törléssel foglalkozó algoritmusokat!

```
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                       const T& value);
```

A *remove* algoritmus törli a sorozatból a *value* értékű elemeket.

```
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                          Predicate pred);
```

A *remove_if* algoritmus törli a sorozatból azon elemeket, amelyekre a predikátum teljesül.

```
OutputIterator remove_copy(InputIterator first, InputIterator last,
                           OutputIterator result, const T& value);
```

A *remove_copy* algoritmus hasonló a *remove* algoritmushoz, csak az eredeti sorozat változatlan marad, az eredmény egy új sorozatban kerül tárolásra.

```
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result, Predicate pred);
```

A *remove_copy_if* algoritmus hasonló a *remove_if* algoritmushoz, csak az eredeti sorozat változatlan marad, az eredmény egy új sorozatban kerül tárolásra.

```
ForwardIterator unique(ForwardIterator first, ForwardIterator last);
```

A *unique* algoritmus törli a sorozatból az ismétlődő szomszédos elemeket. Ahhoz, hogy ez az algoritmus a sorozatból minden ismétlődő elemet töröljön, szükséges, hogy az ismétlődő elemek egymás mellett legyenek, amit például rendezéssel biztosíthatunk.

```
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                       BinaryPredicate binary_pred);
```

A *unique* algoritmus egy másik változata törli a sorozatból azon ismétlődő szomszédos elemeket, melyekre a harmadik paraméterben megadott predikátum igaz.

```
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result);
```

A *unique _copy* algoritmus hasonló a *unique* algoritmushoz, csak az eredeti sorozat változatlan marad, az eredmény egy új sorozatban kerül tárolásra.

```
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result, BinaryPredicate
binary_pred);
```

A *unique_ copy* algoritmus predikátumos változata hasonlít a *unique* predikátumos változatához, csak az eredeti sorozat változatlan marad, az eredmény egy új sorozatban keletkezik.

Az imént ismertetett algoritmusok használatára nézzünk egy olyan példát, mely törli az ismétlődő elemeket egy karakterláncból, majd egyesével törli a karakterlánc elemeit!

```
int main() {
   string s1;
   s1.resize(15);
   generate(s1.begin(), s1.end(), CharGen());
   print(s1.begin(), s1.end(), "s1 original");

   string s2(s1.begin(), s1.end());
   sort(s2.begin(), s2.end());
   print(s2.begin(), s2.end(), "s2");

   string::iterator s2end = unique(s2.begin(), s2.end());
   print(s2.begin(), s2end, "s2");

   print(s2.begin(), s2.end(), "s2");

   string::iterator it = s2.begin();
   string::iterator it2 = s1.end();
   while(it != s2end) {
      cout << "Torlendo:" << *it << " ";
      it2 = remove(s1.begin(), it2, *it);
```

```
      print(s1.begin(), it2, "");
      it++;
   }

   return 0;
}
```

A program futtatása után a következő eredményt kaptuk:

```
s1 original: JchGBAdJBBHifFF
s2: ABBBFFGHJJcdfhi
s2: ABFGHJcdfhi
s2: ABFGHJcdfhidfhi
Torlendo:A : JchGBdJBBHifFF
Torlendo:B : JchGdJHifFF
Torlendo:F : JchGdJHif
Torlendo:G : JchdJHif
Torlendo:H : JchdJif
Torlendo:J : chdif
Torlendo:c : hdif
Torlendo:d : hif
Torlendo:f : hi
Torlendo:h : i
Torlendo:i :
```

Rendezés

Több algoritmus is épít arra a funkcionalitásra, hogy a sorozatokat rendezni lehet. Ilyen például a *unique* algoritmus, amely csak a szomszédos ismétlődő elemeket törli és ha rendezetlen sorozatra alkalmazzuk, akkor maradhatnak a sorozatban ismétlődő elemek.

```
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

A *sort* algoritmus a paraméterben kapott sorozat elemeit helyben növekvő sorrendbe rendezi.

```
void sort(RandomAccessIterator first, RandomAccessIterator last,
          StrictWeakOrdering binary_pred);
```

A *sort* algoritmus másik változata a harmadik paraméterben megadott predikátum alapján rendezi a sorozat elemeit.

```
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
```

A *stable_sort* algoritmus hasonló a *sort* algoritmushoz, azonban a *stable_sort* megőrzi az azonos elemek sorrendjét.

```
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 StrictWeakOrdering binary_pred);
```

A *stable_sort* algoritmus predikátumos változata hasonló a *sort* algoritmus predikátumos változatához, azonban a *stable_sort* megőrzi az azonos elemek sorrendjét.

További rendező algoritmusok:

- *partial_sort,*

 © Ferenc Rudolf, SzTE

- *partial_sort_copy,*
- *nth_element.*

Keresés rendezett sorozatokban

Az STL algoritmusai között vannak olyanok, melyek hatékony keresést biztosítanak rendezett sorozatokban.

```
bool binary_search(ForwardIterator first, ForwardIterator last, const T&
value);
```

A *binary_search* algoritmus bináris keresést valósít meg, megállapítja, hogy a *value* érték szerepel-e a sorozatban.

```
bool binary_search(ForwardIterator first, ForwardIterator last, const T&
value,
                    StrictWeakOrdering binary_pred);
```

A *binary_search* algoritmus másik változata hasonló a korábban említetthez, azonban a < reláció helyett a *binary_pred* predikátum alapján rendezi az elemeket.

```
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
const T& value);
```

A *lower_bound* algoritmus megadja a *value* érték első előfordulását a sorozatban. Ha nem szerepel a sorozatban, akkor azt adja meg, hol kellene lennie.

```
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
const T& value, StrictWeakOrdering binary_pred);
```

A *lower_bound* algoritmus egy másik változata hasonló a korábban említetthez, azonban a < reláció helyett a *binary_pred* predikátum alapján rendezi az elemeket.

```
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
const T& value);
```

Az *upper_bound* algoritmus megadja a sorozatban az első olyan elemet, amely nagyobb a keresett *value* értéknél.

```
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                             const T& value, StrictWeakOrdering
binary_pred);
```

Az *upper _bound* algoritmus másik változata hasonló a korábban említetthez, azonban a < reláció helyett a *binary_pred* predikátum alapján rendezi az elemeket.

További algoritmusok rendezett sorozatokra:

- *equal_range,*
- *Rendezett sorozatok összefésülése: merge, inplace_merge,*
- *Halmazműveletek rendezett sorozatokon: includes, set_union, set_intersection, set_difference, set_symmetric_difference.*

Műveletek sorozat elemeken

*Ha nem az előzőekben ismer*tetett általános műveletekkel szeretnénk dolgozni, akkor sem kell bonyolult ciklusokat írnunk. Az alábbi algoritmusok a sorozat minden elemére végrehajtják a megadott *f* függvényobjektumot.

```
UnaryFunction    for_each(InputIterator    first,    InputIterator    last,
UnaryFunction f);
```

A *for_each* algoritmus minden sorozatbeli elemre végrehajtja az *f*-et, az *f* eredményét nem használja fel.

```
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryFunction f);
```

Az unáris *transform* algoritmus minden sorozatbeli elemre végrehajtja az *f*-et, és az *f* eredményét eltárolja a *result* sorozatban.

```
OutputIterator transform(InputIterator1 first, InputIterator1 last,
                         InputIterator2 first2, OutputIterator result,
                         BinaryFunction f);
```

A bináris *transform* algoritmus minden első és a hozzá tartozó második sorozatbeli elemre végrehajtja az *f*-et, és az *f* eredményét eltárolja a *result* sorozatban.

 © Ferenc Rudolf, SzTE

GENERIKUS KONTÉNEREK

A generikus konténerosztályok olyan objektumokat írnak le, amelyek más típusú objektumok tárolására szolgálnak. A konténerek előnye a nyelv beépített tárolóival szemben (pl. tömb) a rugalmasság. A konténer objektumok automatikusan átméreteződnek, ha új elemet adunk hozzájuk vagy törlünk belőlük. Ez a tulajdonság sokszor hasznos a gyakorlatban, ugyanis gyakran fordul elő, hogy nem ismert előre a tárolandó elemek darabszáma.
Az egyes adatszerkezeteknek megvannak a maga előnyei, illetve hátrányai. C++-ban a különböző igényekre különböző adatszerkezetekkel reprezentált konténereket alkottak. Néhány példa:

- *vector* (dinamikus tömb): gyors adatelérés, kis memóriaigény, lassú beszúrás, törlés (kivéve a végére/végéről),
- *list* (láncolt lista): gyors törlés, beszúrás, nem foglal felesleges területet, de nincs közvetlen adatelérés (kivéve a két végét),
- *map*: közepes gyorsaságú adatelérés, beszúrás, törlés.

Példa konténer és iterátor használatára

Mivel a generikus konténerek bármilyen típusú adatot tárolhatnak, valamint többféle adatszerkezetet használnak, szükség van egy olyan eszközre, amelynek segítségével egységesen bejárhatjuk, elérhetjük, kereshetjük egy konténer elemeit. Ez az eszköz az *iterator*. Az iterátor a bejárásért felelős, és független a konténer típusától.
Az iterátorok használatára nézzünk két példát, ami a halmazok bejárását mutatja be. A halmaz konténer minden elemet csak egyszer tárol (megfeleltethető a matematikai halmaz fogalomnak). Ha olyan elemet szúrunk be, ami már eleme a halmaznak, akkor azt egyszerűen figyelmen kívül hagyja.

```
#include <iostream>
#include <set>

using namespace std;

int main() {
	set<int> intset;
	for (int i = 0; i < 25; i++)
		for (int j = 0; j < 10; j++)
			intset.insert(j);
	cout << intset.size() << endl;
	return 0;
}
```

A program futtatása után a következő eredményt kapjuk:

```
10
```

A fenti program huszonötször beszúrja a számokat egytől tízig a halmazba, de ha lefuttatjuk, akkor a képernyőre kiírt érték (a halmaz mérete) 10 lesz az elemek egyedisége miatt.

Egyes alkalmazásoknál kihasználhatjuk a halmaznak ezt a tulajdonságát, például ha egy fájlban előforduló különböző szavak számát szeretnénk megszámolni úgy, hogy minden szó csak egyszer legyen számolva.

```
#include <iostream>
#include <fstream>
#include <set>
#include <string>
#include <iterator>
using namespace std;

int main(int argc, char* argv[]) {
	ifstream is("main.cpp");
	string szo;
	set<string> szavak;
	while (is >> szo)
		szavak.insert(szo);
	copy(szavak.begin(), szavak.end(), ostream_iterator<string>(cout,
"\n"));
	cout << endl << "Kulonbozo szavak szama: " << szavak.size() << endl;
	return 0;
}
```

A fenti program megszámolja a benne előforduló szavak számát, minden szót csak egyszer vesz figyelembe. A program létrehoz egy sztringeket tároló halmazt, és megnyitja a fájlt bemeneti adatfolyamként. Beolvassa az adatfolyamon található szavakat (ahol szónak két *whitespace* karakter közötti sztringet értünk), és beszúrja őket a halmazba. Ha a halmaz már tartalmazza a szót, akkor a beszúrást figyelmen kívül hagyja. Végül kiírja a halmaz elemeit a képernyőre és lekéri a halmaz elemszámát a *size* függvénnyel. Mivel minden szó csak egyszer szerepelhet, ezért az elemszám a szavak számával egyezik meg.
A program futtatása után a következő eredményt kapjuk:

```
"
"Kulonbozo
"\n"));
#include
(is
0;
<<
<fstream>
<iostream>
<iterator>
<set>
<string>
>>
argc,
argv[])
char*
copy(szavak.begin(),
cout
endl
endl;
ifstream
int
is("main.cpp");
main(int
```

 © Ferenc Rudolf, SzTE

```
namespace
ostream_iterator<string>(cout,
return
set<string>
std;
string
szama:
szavak
szavak.end(),
szavak.insert(szo);
szavak.size()
szavak;
szo)
szo;
using
while
{
}
Kulonbozo szavak szama: 42
```

A program érdekessége még az is, hogy a szavak kiírásakor rendezett sorozatot kapunk. Ez azért van, mert a halmaz a tárolást piros-fekete fával valósítja meg. Ez egy kiegyensúlyozott bináris fa. Beszúráskor eldönti, hogy a gyökér bal vagy jobb részfájába kell szúrni az új elemet (balra, ha kisebb), és ugyanezt a vizsgálatot végrehajtja rekurzívan az összes nem levél csúcsra is. Mivel beszúráskor figyel a sorrendre, gyors a halmazban való keresés, illetve rendezett a kiíratás.

Konténer kategóriák

A konténereket csoportosíthatjuk bizonyos tulajdonságaik szerint. Ezek között a csoportok között a fő különbség a tárolás módja, illetve a biztosított funkciók. Közös bennük, hogy mindegyik tároló flexibilis, azaz ha több elemet teszünk bele, mint az eddigi aktuális kapacitása, akkor automatikusan bővíti saját magát és lefoglalja a szükséges többlet memóriát. Három főbb konténer kategória különböztethető meg:

- Egyszerű sorozat konténerek: Egymást követő elemeket tárol. Az elemei szigorúan tartják a sorrendet, nem keverednek össze. Három ilyen konténer van megvalósítva az STL könyvtárban: *vector* (flexibilis tömb), *list* (láncolt lista) és a *deque* (*double ended queue* – kétvégű sor).
- Konténer adapterek: Adaptálják valamelyik egyszerű sorozat konténert, amelynek segítségével tárolják az elemeket, és valamilyen különleges funkcionalitást biztosítanak. Ilyenek a *queue* (sor), *stack* (verem) és a *priority_queue* (prioritásos sor).
- Asszociatív konténerek: Kulcsokat társítanak értékekkel. Ebbe a kategóriába tartozik a *set* (halmaz), *map* (leképezés), *multiset* és *multimap*.

Egyszerű sorozat konténerek

Vector

A *vector* valójában egy flexibilis tömb, örökli a tömb előnyös tulajdonságait. A közvetlen adatelérés az indexelés miatt nagyon gyors (konstans idejű). Előnye továbbá a tömörsége: magukon az adatokon kívül nincs szükség semmilyen egyéb adat tárolására (ellentétben pl. a láncolt listával).

Azonban az előnyök magukkal vonnak bizonyos hátrányokat is. A *vector* elejére, közepébe való beszúrás nagyon lassú tud lenni, mert az összes utána következő elemet át kell másolni a rákövetkező helyre. A *vector* végére való beszúrás viszont gyors marad.

List

A *list* egy kétszeresen láncolt lista. Lassú az adatelérés, mivel az adott indexű elem megtalálásához végig kell menni az összes őt megelőző elemen, ami nagyon lassú eljárás. Az előnye beszúráskor mutatkozik meg: új elem hozzáadása a listához (ami akár belső elem is lehet) csak néhány pointer átkötésével történik. A törlés hasonlóképpen gyors a beszúrásnál leírt okból kifolyólag.

Deque

A *deque* (*double ended queue*) egy kétvégű sor, amely a *vector*ra hasonlít abban, hogy tetszőleges elemet indexeléssel gyorsan el lehet érni (konstans időben), illetve a közepébe való beszúrás lassú. Az előnye, hogy az elejére vagy a végére történő beszúrás, illetve az onnan való törlés konstans időben megvalósítható.

A következő példában az *alakzatok* nevű *vector Alakzat* ősosztályra mutató pointereket tárol, különféle leszármazott típusú objektumokkal. Végül ezt iterátor segítségével bejárja meghívja az objektumok *megjelenit* metódusát, majd törli az elemeket.

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

class Alakzat {
public:
	virtual void megjelenit() = 0;
	virtual ~Alakzat() {};
};

class Kor : public Alakzat {
public:
	void megjelenit() {cout << "Kor::megjelenit" << endl;}
	~Kor() {cout << "Kor::~Kor" << endl;}
};

class Haromszog : public Alakzat {
public:
	void megjelenit() {cout << "Haromszog::megjelenit" << endl;}
	~Haromszog() {cout << "Haromszog::~Haromszog" << endl;}
};

class Negyzet : public Alakzat {
public:
	void megjelenit() {cout << "Negyzet::megjelenit" << endl;}
	~Negyzet() {cout << "Negyzet::~Negyzet" << endl;}
};

typedef std::vector<Alakzat*> Tarolo;
typedef Tarolo::iterator Iter;

int main() {
```

```
	Tarolo alakzatok;
	alakzatok.push_back(new Kor);
	alakzatok.push_back(new Negyzet);
	alakzatok.push_back(new Haromszog);

	for(Iter i = alakzatok.begin(); i != alakzatok.end(); i++)
		(*i)->megjelenit();

	for(Iter j = alakzatok.begin(); j != alakzatok.end(); j++)
		delete *j;
	return 0;
}
```

A fenti példa definiál egy *Alakzat* ősosztályt. Ez az osztály absztrakt osztály, mivel van *pure virtual* metódusa, ezért őt nem is lehet példányosítani. Származik belőle három osztály (Kor*, Haromszog, Negyzet*), mindegyik másként implementálja a virtuális *megjelenit* metódust.
Létrehozásuk után az alakzatok bekerülnek egy *Alakzat* ősosztályra mutató pointereket tartalmazó vektorba. A vektor végére való beszúrást a *push_back* metódus valósítja meg. Beszúráskor egy automatikus felfele történő típuskonverzió (*upcast*) történik, a leszármazott objektum bármikor biztonságosan őstípussá konvertálható. A példaprogram iterátorok segítségével bejárja a tárolót és egyesével meghívja a *megjelenit* metódust. Ez egy virtuális függvény (ráadásul absztrakt), ezért minden esetben a megfelelő leszármazott osztály metódusa fog végrehajtódni.
A vektorban nem objektumokat tároltunk, hanem pointereket, ezért az általuk mutatott memóriaterületet is fel kell szabadítani. A standard kimeneten látszik, hogy a destruktor is virtuálisan lett megvalósítva, tehát a megfelelő gyerek osztály destruktora fog hívódni. Ellenkező esetben mindig az ős destruktora hívódna meg.
A *main* függvény előtt definiálásra került két típus, a *Tarolo* és az *Iter* nevű típusok. Az első egy ősosztályra mutató pointereket tároló vektor, a második pedig ennek egy iterátora. Ez kényelmesebb, rövidebb írásmódot tesz lehetővé a programban.

Származtatás STL konténerből

Általános irányelv objektum orientált tervezésben, hogy amennyiben lehetséges, a kompozíciót előnyben kell részesíteni az öröklődéssel szemben. Azonban az STL-beli algoritmusok olyan sorozatot (konténert) várnak, melyeknek egy speciális interfészt implementálnak. Azért, hogy ez a feltétel biztosan teljesüljön, az ilyen esetekben származtatni érdemes a saját testreszabott konténereinket az STL-beli konténerekből.
Az alábbi példában a *vector<string>* konténer funkcionalitását bővítjük ki származtatással. Célunk egy olyan program írása, mely beolvas egy szövegfájlt, majd azt egy kis módosítással (sorszámozással ellátva) kiírja egy *stream*re.
A példa osztályunk a *FileEditor* lesz.

FileEditor.h

```
#ifndef _FILEEDITOR_H
#define _FILEEDITOR_H

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

```
#include <vector>

class FileEditor : public std::vector<std::string> {
public:
    FileEditor() {};
    FileEditor(const char* filename) {open(filename);}
    void open(const char* filename);
    void write(std::ostream& out = std::cout);
};

#endif
```

FileEditor.cpp

```
#include "FileEditor.h"
#include <fstream>
#include <algorithm>
#include <stdexcept>
#include <iterator>
using namespace std;

void FileEditor::open(const char* filename) {
    ifstream in(filename);
    if (!in.is_open())
        throw runtime_error("Nem nyithato meg!");
    string line;
    while(getline(in,line))
        push_back(line);
}

void FileEditor::write(ostream& out) {
    copy(begin(), end(), ostream_iterator<string>(out,"\n"));
}
```

FileEditor-test.cpp

```
#include <sstream>
#include <iomanip>
#include "FileEditor.h"
using namespace std;

int main(int argc, char* argv[]) {
    try {
        FileEditor file;
        if (argc > 1)
            file.open(argv[1]);
        else
            return 1;
        int i = 1;
        for (FileEditor::iterator it = file.begin(); it != file.end(); ++it,
++i) {
            ostringstream ss;
            ss << setw(2) << i << ": " << *it;
            *it = ss.str();
        }
        file.write();
    } catch(exception e) {
        cout << e.what() << endl;
        return 1;
```

 © Ferenc Rudolf, SzTE

```
    }
    return 0;
}
```

A FileEditor.h fájlban kerül definiálásra a *FileEditor* osztály. Ez az osztály a sztringeket tároló *vector* konténerből származik. Egy *open* és egy *write* metódus tartozik az osztályhoz. A konstruktor meghívja az *open* metódust. Az *open* metódus a paraméterben kapott fájlt nyitja meg bemeneti *stream*-ként. Ha nem létezik ilyen fájl, akkor kivételt dob, ellenkező esetben végigolvassa a fájlt és a sorokat egyesével beteszi önmagába, mivel ő egy *vector*. A *write* metódus pedig kimásolja a tartalmát a paraméterben kapott *output stream*-re.
A teszt fájl (FileEditor-test.cpp) parancssori paraméterként vár egy fájlnevet. Ha nem érkezik ilyen, akkor kilép a programból hibakóddal. Ellenkező esetben meghívja a *FileEditor* objektum *open* metódusát, melynek paraméterül a kapott fájlnevet adja át. Mivel a *FileEditor* osztály a *vector<string>* konténerből származik, ezért annak minden tulajdonságát örökli, így például az iterátora is a szokásos módon használható. Egy ciklusban végigjárja a *FileEditor* elemeit, és mindegyik elem elé beszúr egy növekvő számot, majd visszateszi az elemet. Tehát megszámozza a sorokat.
Egy példa kimenet néhány sora, ahol a FileEditor-test.cpp fájlt adtuk meg parancssori argumentumként:

```
1: #include <sstream>
2: #include <iomanip>
3: #include "FileEditor.h"
...
```

Iterátorok

A konténer adapter osztályokat kivéve minden tároló osztály támogatja az iterátor mechanizmust, amellyel bejárhatóak sorban a konténer elemei. Az iterátorok a konténerek belső osztályaként vannak megvalósítva (*<Container>::iterator, <Container>::const_iterator*). Az iterátort támogató konténerek mindegyike tartalmaz egy *begin* és egy *end* metódust. Ezek a metódusok iterátor objektumokat adnak vissza. Ha a konténerünk konstans, akkor a *begin* és *end* függvények kiterjesztései konstans iterátorokat hoznak létre.

Fordított iterátorok

Minden konténer támogatja a fordított iterátor mechanizmust is (*<Container>::reverse_iterator, <Container>::const_reverse_iterator*). Ezeket a fordított iterátorokat az *rbegin* és az *rend* függvényekkel hozhatjuk létre. A hagyományos iterátorokhoz hasonlóan a fordított iterátoroknak is van konstans változata.

Beszúró iterátorok

A konténerek három fajta beszúró iterátort támogatnak:

- *back_insert_iterator,*
- *front_insert_iterator,*
- *insert_iterator.*

A *back_insert_iterator*, illetve a *front_insert_iterator* osztályok olyan iterátorok, melyek konstruktora egy konténert vár, és olyan értékadás operatort valósítanak meg, amelyek a hagyományos értékadás helyett a *push_back* / *push_front* függvényt hívnak. Ezeket a beszúró

iterátorokat a könnyebb használathoz a *back_inserter* és a *front_inserter* függvények is előállítják. Az *insert_iterator* konstruktora szintén egy egyszerű sorozat konténert vár, valamint egy pozíció iterátort, és olyan értékadás operatort valósít meg, amely a hagyományos értékadás helyett az *insert* függvényt hívja meg. Az ilyen iterátort az *inserter* segédfüggvény is előállítja.
Nézzünk egy példát a beszúró iterátorokra:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <deque>
#include <list>
#include <iterator>
using namespace std;

int main() {

	int a[] = {1, 3, 5, 7, 11, 13, 17, 19, 23};
	int S = sizeof a/sizeof *a;

	deque<int> di;
	vector<int> vi;
	list<int> li;

	copy(a, a+S, front_inserter(di));
	print(di.begin(), di.end(), "di");

	copy(a, a+S, back_inserter(vi));
	print(vi.begin(), vi.end(), "vi");

	copy(a, a+S, back_inserter(li));
	list<int>::iterator it = li.begin();

	++it; ++it; ++it;

	copy(a, a+S, insert_iterator<list<int> >(li,it));
	//copy(a, a+S, inserter(li,it));
	print(li.begin(), li.end(), "li");

	return 0;
}
```

A fenti példaprogram létrehoz egy tömböt, melyben prímszámok vannak 1-től 23-ig. Ezután létrehoz három sorozat konténert: egy kétvégű sort, egy vektort és egy listát. A *front_inserter* segítségével beszúrja a tömb elemeit a *deque* objektumba úgy, hogy az új elemek a sor elejére kerülnek. Ezután a *vector*-ba is beszúrja az elemeket, de itt már az eredeti sorrendben. A listába is előbb beszúrja eredeti sorrendben, majd kér egy iterátort, amivel ellépked a harmadik elemig. Innen kezdve az *insert_iterator* segítségével még egyszer beszúrja a tömb elemeit. Mindegyik beszúrás után megjeleníti az eredményt a standard kimeneten.
A program futtatása után a következő eredményt kapjuk:

```
di: 23 19 17 13 11 7 5 3 1
vi: 1 3 5 7 11 13 17 19 23
li: 1 3 5 1 3 5 7 11 13 17 19 23 7 11 13 17 19 23
```

Egyszerű sorozat konténerek hasznos tagfüggvényei

Az egyszerű sorozat konténereknek számos hasznos tagfüggvényük van. Néhány fontosabb ezek közül:

- *template <class InputIterator> void* ***assign****(InputIterator first, InputIterator last);*
  - Új értékeket ad a konténer elemeinek. A benne szereplő elemeket egyszerűen eldobja, majd a paraméterben kapott *first* és *last* iterátor közötti elemeket átmásolja (az elsőt beleértve, az utolsót kihagyva).
- *void* ***assign****(size_type n, const T& u);*
  - Hasonlóan az előzőhöz, ez a tagfüggvény is új értéket ad a konténer elemeinek. *n* darab *u* elemre cseréli le a konténer tartalmát.
- *void* ***resize****(size_type sz, T c = T());*
  - Átméretezi a konténert akkorára, hogy *sz* darab elem elférjen benne. Ha *sz* kisebb, mint az eredeti méret, akkor a feleslegessé vált elemeket eldobja. Ha *sz* nagyobb, mint az eredeti méret, akkor kibővíti akkorára a tárolót, és az új helyeket a *c* objektummal feltölti (annyival, amennyi hiányzik az *sz* hossz eléréséhez).
- *void* ***push_back****(const T& x);*
  - Beszúr egy új elemet a tároló végére. Az új elem értéke a paraméterben kapott elem lesz. Gyakorlatilag eggyel megnöveli a konténer méretét. A *vector* esetében ez a memória újrafoglalását is okozhatja, amennyiben a beszúrással meghaladjuk az aktuális kapacitást. Ez az újbóli memóriafoglalás semlegesítheti a korábbi iterátorokat.
- *void* ***pop_back****();*
  - Eltávolítja az utolsó elemet a tárolóból, tehát eggyel csökkenti a konténer méretét, ezáltal semlegesítve az összes erre mutató iterátort.
- *iterator_insert* ***insert****(iterator position, const T& x);*
  - Bővíti a konténert egy új elem beszúrásával (*x*), amit a *position* hely elé szúr be. A *vector esetében* ez a művelet nem túl hatékony, mivel a *vector* belső adatszerkezete tömb, és a beszúrás miatt a beszúrt elem utáni összes elemet át kell másolni egy eggyel nagyobb pozícióra. Visszatérési értéke egy iterátor, ami az újonnan beszúrt elemre mutat.
- *void* ***insert****(iterator position, size_type n, const T& x);*
  - A *position* paraméterben kapott pozíció elé szúr be *n* darab elemet, mindegyiknek az *x* objektumot adja értékül.
- *template <class InputIterator> void* ***insert****(iterator position, InputIterator first, InputIterator last);*
  - A *position* pozíció elé szúrja be a *first* és *last* iterátorok közé eső elemeket. Az intervallum balról zárt, jobbról nyitott, tehát az elsőt beleértve, az utolsót kihagyva.
- *iterator* ***erase****(iterator position);*
  - Kitörli a paraméterben kapott pozíción lévő elemet. Az elemre meghívja a destruktort. *vector* esetén a tároló belsejéből való törlés nem hatékony, mivel az egész tömb átrendezésével jár. Minden iterátor érvénytelen lesz a *position* pozíció után.
- *iterator* ***erase****(iterator first, iterator last);*
  - Kitörli a *first* és *last* közötti intervallumba eső összes elemet. Az intervallum itt is balról zárt, jobbról nyitott. Csökkenti a tároló méretét a törölni kívánt elemek számával, minden egyes elemre meghívja a destruktort. A *vector* esetében a tároló

belsejéből való törlés nem hatékony, mivel az egész tömb átrendezésével jár. Minden iterátor érvénytelen lesz a *first* pozíció után.

- *void* ***swap****(c);*
    - Kicseréli a tároló elemeit a paraméterben kapott tároló elemeire, feltéve, hogy a két tároló elemei azonos típusúak. A méret különbözhet. A csere végrehajtása után a konténer elemei a *c* elemei, *c* elemei pedig a konténer elemei lesznek. Minden eddigi iterátor érvényben marad. A globális *swap* algoritmus ugyanezt valósítja meg.
- *void* ***clear****();*
    - A konténer összes elemét törli. Mindegyikre meghívja a destruktort és törlődnek a konténerből. A művelet végén a konténer mérete 0.
- *bool* ***empty****() const;*
    - A logikai igaz (*true*) értékkel tér vissza, ha a konténer üres, tehát a mérete 0, különben *false*-sal tér vissza. Ez a metódus nem módosítja a tároló tartalmát, a kiürítésre a *clear* függvényt kell használni.
- *size_type* ***size****();*
    - Visszaadja a konténer elemeinek számát. Ennek a száma nem feltétlenül egyezik meg a kapacitással. A kapacitás lekérdezésére a *capacity* metódus szolgál.

Konténer adapterek

A konténer adapterek adaptálják valamelyik egyszerű sorozat konténert, amelynek segítségével tárolják az elemeket, és új funkciót adnak hozzá.

Három konténer adapter típust különböztetünk meg:

- A *stack* (verem) egy speciális sor: az elemeket az elejére szúrja be és innen is veszi ki őket.
- A *queue* (sor) esetében az elejére szúrja be az új elemeket és a végéről veszi ki őket.
- A *priority_queue* (prioritásos sor) hasonlóan működik, mint a sima sor, csak itt a sorrendet nem a beszúrás, hanem a prioritás határozza meg.

Stack

A *stack* klasszikus verem funkcionalitást valósít meg (*LIFO – last-in first-out*) a következő metódusok segítségével:

- *push*: betesz egy elemet a verem tetejére,
- *top*: visszaadja a verem legfelső elemét, de nem veszi le,
- *pop*: levesz egy elemet a verem tetejéről, de nem adja vissza,
- *size*: méret, hány darab elem van a veremben,
- *empty*: üres-e a verem.

Alapértelmezésben a *deque*-et adaptálja. Nincsenek saját iterátorai, nem lehet tudni, hogy mi van a legfelső elem alatt.

Írjunk egy példaprogramot, amely beolvassa egy fájl tartalmát soronként, elmenti egy verembe a sorokat, majd végül kiolvassa a sorokat a veremből!

```
#include <fstream>
#include <iostream>
```

 © Ferenc Rudolf, SzTE

```
#include <stack>
#include <string>
//#include <vector>
//#include <list>
using namespace std;

int main() {
	ifstream in("main.cpp");
	stack<string> s;
	//stack<string, vector<string> > s;
	//stack<string, list<string> > s;
	string line;
	while(getline(in,line))
		s.push(line);
	while(!s.empty()) {
		cout << s.top() << endl;
		s.pop();
	}
	return 0;
}
```

Ha vermet szeretnénk használni egy programban, *include*-olni kell a <*stack*> header fájlt. Egy *ifstream* segítségével megnyitunk egy forrás fájlt, ami jelen esetben maga a program forráskódja. Készítünk egy *s* nevű *stack*-et, ami sztringeket tárol. Második paraméternek megadható, hogy *vector*-t vagy *list*-et adaptáljon az alapértelmezett *deque* helyett. Abban az esetben, ha megadjuk a második paramétert, annak meg kell adni sablonparaméterként, hogy milyen elemeket tárol, ami jelen esetben a *string* (lásd a kommentezett sorokat). Deklarálunk egy *line* nevű sztringet a fájl egy sorának tárolása céljából. Amíg a *getline* a fájlból be tud olvasni újabb sorokat, addig a verembe beteszi a kiolvasott sort. Végigolvassa a fájlt, majd kiíratja a verem tartalmát, amíg az *s* nem ürül ki teljesen. Értelemszerűen megfordul a sorrend, amikor az elemeket sorra kivesszük. Kiíratjuk a *top* függvénnyel azt az elemet, ami a verem tetején van, majd *pop* függvénnyel ki is vesszük a veremből. Fontos megjegyezni, hogy ha üres verem esetében lekérjük a legfelső elemet, akkor elszáll a program (nem dob kivételt és nem is tér vissza hibakóddal).

A példaprogram futtatása után kapott eredmény a fájl sorai fordított sorrendben:

```
}
	return 0;
	}
		s.pop();
		cout << s.top() << endl;
	while(!s.empty()) {
		s.push(line);
	while(getline(in,line))
	string line;
	//stack<string, list<string> > s;
	//stack<string, vector<string> > s;
	stack<string> s;
	ifstream in("main.cpp");
int main() {

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

```
#include <stack>
#include <iostream>
#include <fstream>
```

Queue

A *queue* klasszikus sor funkcionalitást valósít meg (*FIFO – first-in first-out*) a következő metódusok segítségével:

- *push:* betesz egy elemet a sor végére,
- *front:* visszaadja a sor első elemét,
- *back:* visszaadja a sor utolsó elemét.
- *pop:* levesz egy elemet a sor elejéről,
- *size*: visszaadja a sor méretét,
- *empty:* üres-e a sor.

Alapértelmezésben a *deque*-et adaptálja. A *queue*-nak sincsenek saját iterátorai a veremhez hasonlóan, le lehet kérdezni az első és az utolsó elemeket, a közteseket azonban nem.
Írjunk egy példaprogramot, amely beolvassa egy fájl tartalmát soronként, elmenti egy *queue*-ba a sorokat, majd végül kiírja a sorokat a queue-ból!

```
#include <fstream>
#include <iostream>
//#include <list>
#include <queue>
#include <string>
using namespace std;

int main() {
	ifstream in("main.cpp");
	queue<string> s;
	//queue<string, list<string> > s;
	string line;
	while(getline(in,line))
		s.push(line);
	while(!s.empty()) {
		cout << s.front() << endl;
		s.pop();
	}
	return 0;
}
```

Ha *queue*-t szeretnénk használni egy programban, *include*-olni kell a <*queue*> header fájlt. Egy *ifstream*-mel megnyitunk egy fájlt, ami most is maga a program forráskódja. Készítünk egy *s* nevű *queue*-t, ami sztringeket tárol. Második paraméternek megadható, hogy *vector*-t vagy *list*-et adaptáljon. Abban az esetben, ha megadjuk a második paramétert, annak meg kell adni sablonparaméterként, hogy milyen elemeket tárol, ami ebben a példában *string*. Deklarálunk egy *line* nevű sztringet a fájl egy sorának tárolása céljából. Amíg a *getline* a fájlból be tud olvasni sorokat, addig a sor végére teszi a kiolvasott sort a *push* metódussal. Végigolvassa a fájlt, majd kiírja a *queue* tartalmát, amíg az *s* nem üres. Kiíratjuk a *front* függvénnyel azt az elemet, ami a *queue* elején van, majd *pop* függvénnyel ki is vesszük a sorból. Ahogyan a stack esetében, itt is fontos megjegyezni, ha üres sor esetén lekérjük a legelső elemet, akkor elszáll a program (nem dob kivételt, és nem tér vissza hibakóddal).

 © Ferenc Rudolf, SzTE

A példaprogram futtatása után kapott eredmény a fájl sorai eredeti sorrendben:

```
#include <fstream>
#include <iostream>
//#include <list>
#include <queue>
#include <string>
using namespace std;

int main() {
	ifstream in("main.cpp");
	queue<string> s;
	//queue<string, list<string> > s;
	string line;
	while(getline(in,line))
		s.push(line);
	while(!s.empty()) {
		cout << s.front() << endl;
		s.pop();
	}
	return 0;
}
```

Priority_queue

A *priority_queue* prioritásos sor funkcionalitást valósít meg. Ha egy elem magasabb prioritással rendelkezik egy másiknál, akkor az az elem előrébb fog elhelyezkedni a sorban, mint a másik. A következő metódusok segítségével érhetők el a funkciói:

- *push:* betesz egy elemet a prioritási sorba, ahova a prioritása szerint illik. A legfontosabb elem kerül a sor elejére, a legalacsonyabb prioritású elem pedig a végére.
- *top:* visszaadja a sor első (a legnagyobb prioritású) elemét,
- *pop:* leveszi az első (a legnagyobb prioritású) elemet,
- *size*: visszaadja a prioritásos sor méretét,
- *empty:* üres-e a prioritásos sor.

Alapértelmezésben a *vector*-t adaptálja. A *priority_queue*-nak sincsenek saját iterátorai a veremhez és a sorhoz hasonlóan, le lehet kérni az első elemét, a többit azonban nem.
A következőkben bemutatásra kerül egy olyan példa, amely 10 darab 1 és 99 közötti véletlen számot tárol el egy prioritásos sorban, majd kiírja az elemeket a konzolra.

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

int main() {
	priority_queue<int> p;
	srand(time(0));
	for(int i = 0; i < 10; i++)
		p.push(rand() % 100);
	while(!p.empty()) {
		cout << p.top() << endl;
		p.pop();
	}
	return 0;
```

```
}
```

Ha *priority_queue*-t használunk egy programban, akkor a *<queue>* header fájlt kell include-olni. A prioritásos sorba a *push* függvénnyel rakja be a véletlen számot, majd a *top* függvénnyel kéri le a legnagyobb prioritású elemet – ami egészek esetében azt jelenti, hogy a nagyobb szám nagyobb prioritással rendelkezik –, és a *pop* függvény segítségével veszi ki az elemet a sorból. Így csökkenő sorrendben kapjuk meg a véletlen számokat.
Prioritásos sor használata során azonban lehetőség van arra is, hogy ne az alapértelmezett prioritást vegye figyelembe, meg lehet adni egy tetszőleges függvényobjektumot, ami definiálja, hogy két elem közül melyiknek nagyobb a prioritása. Készítsünk egy olyan példaprogramot, amelyben a prioritásos sor növekvő sorrendben tárolja az elemeket A nekünk megfelelő STL-beli függvényobjektum a *greater*. A következő program erre mutat példát:

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

int main() {
	priority_queue<int, vector<int>, greater<int> > p;
	srand(time(0));
	for(int i = 0; i < 10; i++)
		p.push(rand() % 100);
	while(!p.empty()) {
		cout << p.top() << endl;
		p.pop();
	}
	return 0;
}
```

A két példában csak egyetlen sor különbözik: az első esetben csak azt adtuk meg a prioritásos sornak, hogy egészeket fog tartalmazni (*priority_queue<int>*), a második esetben, már megadtuk a prioritásos sor második és harmadik sablonparaméterét is (*priority_queue<int, vector<int>, greater<int> >*). A második paraméter adja meg, hogy milyen egyszerű konténert adaptáljon, a harmadik paraméter pedig egy függvényobjektumot vár, ami alapján eldönti, hogy mely elemnek nagyobb a prioritása. A *greater* STL-beli függvényobjektum megadásával a nagyobb egészek alacsonyabb prioritással fognak rendelkezni, így a kiíratásnál már növekvő sorrendben láthatók a sor elemei.

Nézzünk a prioritásos sor használatára egy sokkal gyakorlatiasabb példát!

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

class Tennivalo {
	int prioritas;
	string leiras;
public:
	Tennivalo(string leiras, int prioritas = 1)
		: leiras(leiras), prioritas(prioritas) {}
	friend bool operator<(const Tennivalo& x, const Tennivalo& y) {
		return x.prioritas > y.prioritas;
```

```
	}
	friend ostream& operator<<(ostream& os, const Tennivalo& t) {
		return os << t.prioritas << " : " << t.leiras;
	}
};
```

Implementáltunk egy *Tennivalo* osztályt, amely egy teendő leírását és prioritását képes eltárolni. Minden tennivalót 1-1 objektumként fog ábrázolni a programunk. Az osztály konstruktora kap egy sztringet és egy prioritást, ami alapértelmezett esetben 1. Ezekkel az adatokkal inicializáljuk az adattagokat. Készítettünk egy kisebb operátort, amit a prioritásos sor fog használni ahhoz, hogy össze tudja hasonlítani az tennivalókat egymással. Alaptípusokra eleve értelmezett, de saját típusokra nekünk kell ezt megvalósítani. A *Tennivalo* osztály esetében akkor kisebb az első paraméter, mint a második, ha kevésbé fontos, azaz a prioritása nagyobb számértékű. Készítettünk egy kiírató operátort is, amely kiírja a teendő prioritását és szövegét.

```
int main() {
	priority_queue<Tennivalo> lista;
	lista.push(Tennivalo("Kirakni a szemetet",2));
	lista.push(Tennivalo("Megetetni a kutyat",1));
	lista.push(Tennivalo("Mosogatni",3));
	lista.push(Tennivalo("Takaritani",3));
	while(!lista.empty()) {
		cout << lista.top() << endl;
		lista.pop();
	}
	return 0;
}
```

A *Tennivalo* osztály funkcióinak vizsgálata céljából nincs más dolgunk, mint létrehozni egy prioritásos sort, ami tennivalókat tárol. A *push* függvény segítségével feltöltjük a listát tennivalókkal. Végül kiíratjuk a lista tartalmát a képernyőre. Amíg a lista nem üres, *top* függvénnyel lekérjük a legnagyobb prioritású elemet, majd kiíratjuk, és *pop* függvénnyel kivesszük a listából.
A példaprogram futtatása után kapott eredmény:

```
1 : Megetetni a kutyat
2 : Kirakni a szemetet
3 : Takaritani
3 : Mosogatni
```

Asszociatív konténerek

Az asszociatív konténerek objektum párokat tárolnak, konkrétan kulcs-adat párokat (asszociatív ~ társító). Négy asszociatív konténer típust különböztetünk meg:

- *map:* asszociatív tömb, kulcs-érték párokat tárol, gyors keresést és beszúrást biztosít,
- *multimap*: hasonló a *map*-hez, de megenged több azonos kulcsú elemet is,
- *set:* halmaz, kulcsokat tárol értékek nélkül, gyors keresést és beszúrást biztosít,
- *multiset:* hasonló a *set*-hez, de megenged több azonos kulcsú elemet is.

Map

A *map* a hagyományos tömbhöz hasonlít, csak ott egész számokat (*index*) vannak társítva objektumokhoz. A *map* viszont általánosabb, olyan asszociatív tömb, amely objektumokat társít objektumokkal. Indexelhető a [] operátorral. A *map* egy bináris piros-fekete fában tárolja elemeit. Ha nem létező kulcsú elemet kérünk le az index operátorral, akkor készít egyet. Minden tárolt elem egy páros, amit az *std::pair* ábrázol. Ennek két adattagja van, a *first* és a *second.*

A következő példaprogram segítségével megszámoljuk, hogy hányszor fordulnak elő különböző szavak egy fájlban, ahol egy szó alatt elválasztó karakterek közötti sztringet értünk:

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

typedef map<string,int> Map;
typedef Map::const_iterator MapIterator;

int main() {
	ifstream in("main.cpp");
	Map m;
	string szo;
	while(in >> szo)
		m[szo]++;
	for(MapIterator it = m.begin(); it != m.end(); ++it)
		cout << it->first << ": " << it->second << endl;
	cout << endl;
	MapIterator it2 = m.find("namespace");
	if (it2 != m.end()) {
		cout << it2->first << endl;
		std::pair<string,int> p = *it2;
		cout << p.first << endl;
	}
	return 0;
}
```

Ha egy programban szeretnénk használni *map*-et, *include*-olni kell a <*map*> header fájlt. Az egyszerűség kedvéért definiáltunk került két új típust *typedef* segítségével: *Map* és *MapIterator*. A *map* első sablonparamétere alapján a kulcs típusa sztring, a második sablonparaméter pedig azt mutatja, hogy a sztringekhez egész értékek lesznek hozzárendelve.
A program *input filestream*-et állít a fájlra, amivel beolvassuk magát a forráskódot. Létrehozunk egy *map*-et, ami *string* és *int* párokat fog tárolni. A beolvasott szavakat, mint kulcsokat eltároljuk a *map*-ben, és a szavakhoz olyan egészeket rendelünk, amelyek az adott szó előfordulásának számát mutatják. Ezt úgy érjük el, hogy megindexeljük a *map*-et az aktuális szóval. Ha még nem létezik a *map*-ben az adott kulcsú elem, akkor röptében készít neki az index operátor egy olyan kulcsú elemet, és az ahhoz tartozó 0-t tartalmazó adat értéket megnöveljük eggyel az inkrementáló operátorral. Ha már egy korábban előforduló szót olvasunk be, akkor az index operátor a már meglévő kulcsú adatot adja vissza, és annak előfordulási az értéket növeljük meg.

Kiíráskor a *map* elemeit iterátor segítségével járjuk be, a *first* adja vissza kulcs értékét, magát a szót, a *second* pedig az adott szó előfordulásainak a számát. Ha arra vagyunk kíváncsiak, hogy van-e egy bizonyos kulcsú elem, akkor a *find* függvényt használjuk. Ha szeretnénk megkeresni a pl. a *namespace* kulcsú elemet, akkor a *find* visszatérési értéke, ami egy iterátor, megmutatja az elem helyét. Ha a keresett elem nem eleme a *map*-nek, akkor az utolsó utáni elemre (*end*) fog mutatni az iterátor. Végül a sikeresen megtalált kulcs-adat párt a példa kedvéért kimásoljuk egy lokális *pair* objektumba és kiírjuk annak az első elemét is. A példaprogram futtatása után kapott eredmény:

```
!=: 2
": 1
":: 1
#include: 4
(it2: 1
*it2;: 1
++it): 1
<<: 9
<fstream>: 1
<iostream>: 1
<map>: 1
<string>: 1
=: 3
...
main(): 1
map<string,int>: 1
map<string,int>::iterator: 1
namespace: 1
p: 1
...

namespace
namespace
```

Multimap

A *multimap* olyan *map*, amely tartalmazhat több azonos kulcsú elemet. Kulcs szerint rendezve piros-fekete fában tárolja az elemeket. Képzeljünk el egy telefonkönyvet, ahol a név a kulcs. Ilyen esetekben feltehetjük, hogy több személynek is lehet ugyanaz a neve. Mivel több azonos kulcs megengedett, ezért az index operátor nem alkalmazható *multimap* esetén.
Nézzük meg az előző példa hogyan viselkedik a *multimap*-re adaptálva:

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

typedef multimap<string,int> MultiMap;
typedef MultiMap::iterator MultiMapIterator;
int main() {
    ifstream in("main.cpp");
    MultiMap m;
    string szo;
    while (in >> szo)
        m.insert(MultiMap::value_type(szo,1));
    for (MultiMapIterator it = m.begin(); it != m.end(); ++it)
```

```
        cout << it->first << ": " << it->second << endl;
    cout << endl;
    pair<MultiMapIterator,MultiMapIterator> itp = m.equal_range("cout");
    for (MultiMapIterator it2 = itp.first; it2 != itp.second; ++it2)
        cout << it2->first << endl;
    return 0;
}
```

A *multimap* használatához szintén a *map*-et kell *include*-olni. Mivel nincs index operátor, így most az *insert* függvény segítségével töltjük fel a *multimap*-et. Az *insert* egy *std::pair*-t vár paraméterül, ami alapesetben jelen példában így néz ki: *pair<string,int>(szo,1)*. Ez kissé kényelmetlen, mert mindig pontosan fel kell paraméterezni a pair sablont és típus függővé tesszük a kódunkat. Ezért definiálták a *map*-ben és a *multimap*-ben a következő típust:

```
typedef pair<const Key, Type> value_type;
```

Ennek segítségével kényelmesebben készíthetünk megfelelő pair objektumokat a programunkban. Ha szeretnénk egy olyan keresést indítani, mellyel megkapjuk az összes keresett kulccsal rendelkező elemet, akkor az *equal_range* metódust célszerű használni. Ez a metódus egy párossal tér vissza, ami két darab *MultiMapIterator*-t tartalmaz. Az első iterátor rámutat az első keresett kulcsú elemre, a második pedig az utolsó utáni találatra mutat. Egy *for* ciklus segítségével bejárjuk a *cout* kulcsú elemeket.

A példaprogram futtatása után kapott eredmény:

```
...
>>: 1
MultiMap: 1
MultiMap::iterator;: 1
MultiMap;: 1
MultiMapIterator;: 1
cout: 1
cout: 1
cout: 1
endl;: 1
endl;: 1
endl;: 1
for: 1
for: 1
...

cout
cout
cout
```

Set és multiset

A *set* egy matematikai értelemben vett halmaz megvalósítás, amely egy elemet csak egyszer tartalmazhat. Ahogy a *map* is, piros-fekete bináris fában rendezve tárolja az elemeket. A *multiset* olyan set, amely több azonos elemet is tartalmazhat.

Asszociatív konténerek hasznos tagfüggvényei

- *begin, end, rbegin, rend*: iterátorokat készítő metódusok,

 © Ferenc Rudolf, SzTE

- *size:* visszaadja a konténer méretét,
- *clear*: kiüríti a konténert,
- *count:* visszaadja, hogy az adott kulcsú elemből hány darabot tartalmaz a konténer,
- *empty:* üres-e a konténer,
- *erase:* elem törlése a konténerből,
- *insert:* elem beszúrása a konténerbe,
- *find:* adott kulcsú elem keresése a konténerben,
- *lower_bound:* az első egyenlő vagy nagyobb kulcsú elem keresése a konténerben,
- *upper_bound*: az első nagyobb kulcsú elem keresése a konténerben,
- *equal_range*: az előző kettő keresés eredménye egy *pair*-ben,
- *swap*: elemcsere másik konténerrel,
- *operator[ ]:* index operátor (csak *map* esetében).

KÖSZÖNETNYILVÁNÍTÁS

Elsősorban családomnak, a kedves feleségemnek és három gyermekemnek szeretném megköszönni a sok segítséget, jókedvet és türelmet, amire nagy szükségem volt a jegyzet elkészítése során. Ezen kívül köszönettel tartozom a Szegedi Tudományegyetem 2010/2011-es tanév I. félévében lezajlott Fejlett programozás MSc kurzus hallgatóinak a sok hasznos észrevételért és a jegyzet kidolgozásában való részvételért, különösen a következő hallgatók esetében: Somogyi Viktor, Dévai Richárd, Csatlós Gábor, Muhi Kornél, Koncz Balázs, Langó László, Bordé Sándor, Márton Péter, Mergl Zalán, Szabó Péter, Földesi Erika, Asztalos István Mihály, Szilasi István, Kovács Ede. Külön köszönet jár Kakuja-Tóth Gabriella PhD hallgatónak, Dr. Siket István és Siket Péter kollégáimnak az Objektum-orientált programozás fejezetben való részvételért és a teljes jegyzet szerkesztésében adott segítségért.
A jegyzetben szereplő több példaprogram alapját Bruce Eckel: Thinking in C++ könyvei és Bjarne Stroustrup: A C++ programozási nyelv könyve adta.

 © Ferenc Rudolf, SzTE

FELHASZNÁLT IRODALOM

- HBjarne Stroustrup: A C++ programozási nyelv.
  Kiskapu Kiadó, Addison-Wesley, 2001.
- Bruce Eckel: Thinking in C++, Volume One: Introduction to Standard C++
  Prentice Hall, 2000 (2nd edition)
- Bruce Eckel: Thinking in C++, Volume Two: Practical Programming
  Prentice Hall, 2003
- Matthew H. Austern: Generic Programming and the STL. Using and Extending the C++ Standard Template Library.
  Addison-Wesley, 1999
- Scott Meyers: Effective C++.
  Addison-Wesley, 2005 (3rd edition)
- Scott Meyers: More Effective C++: 35 New Ways to Improve Your Programs and Designs.
  Addison-Wesley, 1997 (2nd edition)
- Scott Meyers: Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library.
  Addison-Wesley, 2001
- International Standard for C++.
  ISO/IEC, 2003 (2nd edition)
- Andrei Alexandrescu: Modern C++ Design.
  Addison-Wesley, 2001