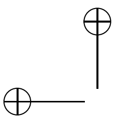
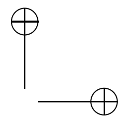
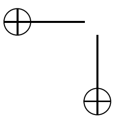
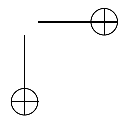
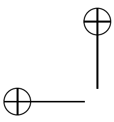
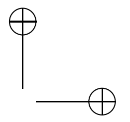



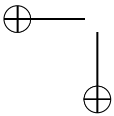
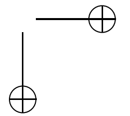
Fejlett programozási technikák

Antal Margit





TARTALOM

1 Általánosított programozás 7
- 1.1. Bevezetés 7
- 1.2. Komponens 7
- 1.3. C++, történelmi áttekintés 9
- 1.4. Általánosított programozás (Generic Programming) 10

2 A C++ programozási nyelv 15
- 2.1. Logikai típus 15
- 2.2. Névterek 16
- 2.3. Kimeneti/Bemeneti műveletek 17
- 2.4. Referencia (hivatkozás) típus 18
- 2.5. Dinamikus helyfoglalás 22
- 2.6. Kivételkezelés 24

3 Osztályok és objektumok 29
- 3.1. Implicit paraméterek 30
- 3.2. Statikus tagok 31
- 3.3. Konstruktor és destruktor 33
- 3.4. Objektumok 37
- 3.5. Helyben kifejtett (inline) tagfüggvények 39
- 3.6. Konstans tagfüggvények 40
- 3.7. A this pointer 40
- 3.8. Barát függvények és osztályok 41
- 3.9. Gyakori hibák 44
- 3.10. Feladatok 45

4 Operátorok túlterhelése 47
- 4.1. Megkötések az operátor-túlterhelésre nézve 49
- 4.2. Operátorfüggvények deklarációja és hívása 49

4.3. Különböző típusú operátorok túlterhelése 51
4.4. Feladatok 63

5 Származtatott osztályok 65
5.1. Bevezetés 65
5.2. Származtatás 67
5.3. A származtatott osztály példányának létrehozása 68
5.4. Adattagok felülírása 73
5.5. Metódusok felülírása - Polimorfizmus 74
5.6. Származtatott osztálybeli objektumok másolása 76
5.7. Absztrakt osztályok 78
5.8. Virtuális destruktor 79
5.9. Privát öröklés 79
5.10. Többszörös öröklés 81
5.11. Feladatok 82

6 A standard könyvtár 85
6.1. Bevezetés 85
6.2. STL komponensek 86
6.3. Konténerek 88
6.4. Iterátorok 92
6.5. Algoritmusok 98
6.6. Feladatok 110

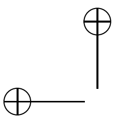
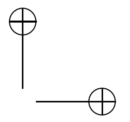
7 Tárolók 113
7.1. Tárolók jellemzése 113
7.2. Tárolók alapműveletei 114
7.3. Sajátos tárolóműveletek 116
7.4. Feladatok 127

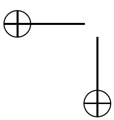
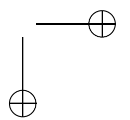
8 Iterátorok 129
8.1. Bevezetés 129
8.2. Iterátor kategóriák 130
8.3. Iterátor adapterek 134
8.4. Iterátorok jellemzői 141
8.5. Saját iterátorok készítése: Az egyszeresen láncolt lista 142
8.6. Feladatok 144

9 Sablonok 147
9.1. Bevezetés 147
9.2. Függvénysablonok 147
9.3. Osztálysablonok 152
9.4. Közönséges típusú sablonparaméterek 158

9.5. Statikus és dinamikus polimorfizmus 159
9.6. Feladatok 161

10 Adatfolyamok 163
10.1. Bevezetés 163
10.2. Adatfolyam osztályok 165
10.3. Manipulátorok 174
10.4. Formázás 175
10.5. Fájlfolyamok 180
10.6. Feladatok 189

1. FEJEZET

ÁLTALÁNOSÍTOTT PROGRAMOZÁS

1.1. Bevezetés

Az ipari forradalomnak közel 200 évre volt szüksége, hogy eljusson a kézműves korszaktól az automatizált tömegtermelésig. 1826-ban John Hall bevezette a cserélhető komponens elvét hadiipari eszközök előállítására. Közel egy évszázad múlva, 1901-ben Ransom Olds bevezeti a szerelőszalag elvét. Az autóiparban az első nagy sikert Henry Ford aratta 1908-ban az úgynevezett T modell bevezetésével, amelyet már elérhető áron forgalmaztak. 1913-ban áttértek a szerelőszalagon való előállításra, megháromszorozva ezáltal a termelést. A következő nagy lépés a szerelőszalag automatizálása volt. 1961-ben a General Motors volt az első, aki alkalmazta az első ipari robotot. A robotok csak a nyolcvanas évek elején váltak sikeressé, a mikroprocesszorok bevezetésével. Valami hasonló történik a programozási iparban is a komponensek bevezetésével.

Az általánosított programozás egy olyan programozási paradigma, amelynek célja olyan általános, újrafelhasználható szoftver termékcsaládok előállítása, amelyek lehetővé teszik sajátos termékek igény szerinti előállítását (generálását).

1.2. Komponens

A szoftverkomponens egy nagyon népszerű fogalom a mai szoftveriparban. Ezen komponenseket úgy határozhatjuk meg, mint olyan építőkockák, amelyek-

1826	1901	1980
Cserélhető komponensek	Szerelőszalag	Automatizált szerelőszalag

1.1. táblázat. Ipari forradalom

ből különböző szoftverek állíthatók elő. Ezen építőkockákat úgy kell megtervezni, hogy sokféleképpen illeszthetőek legyenek. Célunk a kódduplázás minimalizálása és az újrafelhasználás maximalizálása. Ezek a tulajdonságok jellemzik a komponenseket, de nem határozzák meg ezeket.

A komponens egy jól meghatározott termelési folyamat része. Például a tégla az építkezés alapkomponense és nem az autóépítésé. Ha C++ tárolókra van szükségünk, akkor használhatjuk az STL tárolókat. Ha GUI komponensekre van szükségünk, akkor használhatunk például JavaBeans komponenseket. Ha pedig nyelvfüggetlen osztott komponensre van szükségünk, akkor a CORBA technológia segíthet (Common Object Request Broker Architecture).

Az általánosított programozás nem a megszokott módon használja a komponenseket, éspedig nem kiválasztva azt egy komponenshalmazból hanem előállítva (kigenerálva) egy általános komponensből. Ennek a megközelítésnek az az előnye, hogy nem kell millió hasonló jellegű konkrét komponenset tárolni, hanem csak egy pár elemi komponenst, amelyből megadva a szükséges komponens jellemzőit, előállítható az. Ez a megközelítés emeli a kód absztrakciós szintjét, hiszen kódunk egy specifikációt tartalmaz és ha időközben megjelenik a specifikációt kielégítő jobb komponenes, akkor a forráskód módosítása nélkül azt fogja használni a programunk.

C++ nyelvben az általánosított programozás nyelvi eszköze a sablon. A sablont úgy is elképzelhetjük, mint a fordítóprogram egy kiegészítése, hiszen olyasmire használjuk (kódgenerálás), amit általában a fordítóprogramok szoktak végezni. Gyakorlatilag minden sablondefiníció gazdagítja a programozási környezetet. Rendszerfejlesztés szempontjából az általánosított programozás rendszercsaládok modellálását jelenti, amelyekből tetszés szerint konkrét rendszereket lehet gyártani.

Milyen előnyeink származhatnak a rendszerfejlesztés ilyen irányú megközelítéséből?

Elsősorban a szoftverek nagy része egy konkrét programozási nyelvben van megvalósítva. A szoftver tervezési részletei már formába vannak öntve, így ez már nem újrafelhasználható. Ennek következtében a szoftver áthelyezése egy másik platformra elég költséges lehet. Természetesen az általánosított programozás egyik hátránya a költséges fejlesztés, hiszen egyetlen konkrét termék sokkal olcsóbban és gyorsabban előállítható, mint egy termékcsalád. Azonban, ha hosszú

távon aktívak szeretnénk maradni a szoftverfejlesztési piacon, akkor mindenképpen megéri termékcsaládokban és nem egyedi termékekben gondolkodnunk.

1.3. C++, történelmi áttekintés

A nyolcvanas évek elején az AT&T Bell Labs-ban Bjarne Stroustrup kidolgozta a nyelv első változatát. Az első változat neve „C with classes", amelyet 1983-ban C++-ra változtatnak. Az első változat célja az objektumorientált programozási mód lehetővé tétele a C nyelvben. A szerző véleménye szerint: „Csak annyit akartunk elérni, hogy könnyebben és élvezetesebben írhassunk jól használható programokat. Kezdetben nem vetettük papírra rendszerezetten a fejlesztési terveket: egyszerre terveztünk, dokumentáltunk és alkottunk." A megjelenése óta a nyelv gyorsan terjedt és több szabványosításon is átesett. Szabványosításra olyankor van szükség, amikor túl sok fordítót írnak egy adott nyelvhez és ezek a fordítók különféle bővítményeket tartalmaznak, amelyek következtében összeférhetetlenné válnak a forrásnyelvi programkódok. Egyszóval a fordítók elburjánzása szükségessé teszi a szabványosítási folyamatot.

A C++ szabványosítások eredményeképpen a nyelv a következő elemekkel bővült:

- többszörös öröklés
- static és const tagfüggvények
- protected (védett tagok)
- sablonok (template)
- kivételkezelés
- futás idejű típusazonosítás (Run Time Type Identification)
- névterek (namespace)
- szabványos sablon függvénykönyvtár (Standard Template Library)

A szabványosítások következtében a C++ nyelv egy olyan programozási nyelvvé alakult, amely:

- támogatja az elvont adatábrázolást (absztrakt adattípus)
- támogatja az objektumorientált programozást
- támogatja az általánosított programozást

1.4. Általánosított programozás (Generic Programming)

Egy adatstruktúra, mint például a verem, egy általános fogalom, ezért megvalósításakor általánosságra kell törekednünk. Ebben az esetben ez azt jelenti, hogy jó lenne nem egy rögzített típusra implementálnunk, hanem általánosan. Természetesen erre már a szabványos C is nyújtott lehetőséget, hiszen használhattunk *void* ∗ típust. Az is igaz, hogy nem volt túl kényelmes a *void* ∗típussal dolgozni, de járható út volt. A Java ezt a problémát egy közös *Object* ősosztály bevezetésével oldja meg, amelyet kiegészít a primitív típusokhoz létrehozott csomagolóosztályokkal, lehetővé téve az általános adatstruktúrák megvalósítását. Ez az út C++ nyelvben is járható lenne, ha létezne egy közös ősosztály. Ilyen ősosztály viszont nem létezik C++-ban. A C++ szabványosításai során bevezették a paraméterezett típust, amely lehetővé teszi a típusfüggetlen adatstruktúrák kényelmes megvalósíthatóságát. Természetesen a Java Object alapú konténerek és a C++ paraméterezett típus nem ugyanolyan erősségű fogalom. Amíg az `Object` alapú konténerek egy úgynevezett „dinamikus polimorfizmust" (dynamic polimorphism) biztosítanak, amely azt jelenti, hogy egy adott típusra megírt konténer használható az adott típus összes leszármaztatott típusára, addig a C++ paraméterezett típus egy „statikus polimorfizmust" (static polimorphism) biztosít.

A paraméterezett típus használható úgy struktúra, mint osztály esetén, hiszen az utóbbi az előbbi kiterjesztésének tekinthető. Egy verem absztrakt adattípus például a következőképpen adható meg C++ nyelvben:

```
template <class T>
class Stack{
private:
   T * elements;
   int size;
   int sp;
public:
   Stack(int size)
   void push(T e);
   T pop();
   ...
};
```

A fenti verem típus úgy használható, ha példányokat hozunk létre. A példányosítás folyamatában meg kell adnunk viszont a T típusparaméter helyett a konkrét típust. Például:

```
Stack<int> si(10);
```

A fenti definíció a következő feladatokat rója a fordítóra:

- hozzon létre egy konkrét típust az általános veremtípusból, a mi esetünkben egy egészek tárolására alkalmas vermet `Stack<int>`
- foglaljon helyet egy egészeket tartalmazó verem-objektumnak
- hívja meg a `Stack<int>` osztály konstruktorát a 10-es paraméterrel

Gyakorlatilag, amikor egy általános osztályt példányosítunk, akkor kétféle példányosítás történik:

általános típus→sajátos típus
sajátos típus→objektum (példány)

Természetesen a fenti általános veremtípusból bármilyen konkrét veremtípus létrehozható. Ha például karaktereket tartalmazó veremre van szükségünk, akkor ezt a következőképpen hozhatjuk létre:

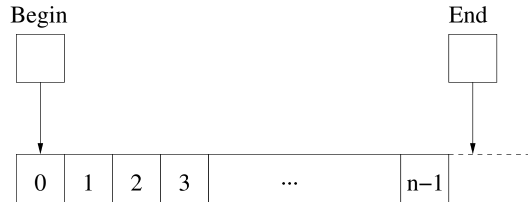
```
Stack<char> sc(40);
```

Ez a definíció is hasonló feladatok elé állítja a fordítót, amely létre fog hozni még egy konkrét veremtípust, amely most karakterek tárolására lesz alkalmas. Ha egy programon belül nagyon sokféle típusú vermet használunk, akkor ez automatikusan a kód felpuffadásához fog vezetni. Természetesen ez a probléma is orvosolható. Technikai megoldásra példát az [1] könyv 449. oldaláan találunk.

Nemcsak a felhasználói típusok élvezik az általános megadhatóságot, hanem az algoritmusok esetében is lehetővé válik, hogy úgy adjuk meg, hogy maximálisan együtt tudjon dolgozni a különböző adatstruktúrákkal. Tekintsük például a következő másolófüggvényt, amely egy forrástartomány elemeit másolja egy céltartományba.

```
template <class Input, class Output>
void mycopy(Input begin1, Input end1, Output begin2){
   while(begin1 != end1){
      *begin2 = *begin1;
      begin1++;
      begin2++;
   }
}
```

Amíg az első példában világos, hogy a T egy típust jelöl, addig a második példában az *Input* és *Output* szerepe nem annyira egyértelmű. A megértést megkönnyíti, ha pontosan tudjuk, hogy hogyan is működik egy általános célú másolófüggvény. Mindenképpen, ha tudjuk, hogy a forrástartomány elemei felsorolhatók (létezik egy bejárási sorrend), akkor a másolandó tartományt könynyedén azonosíthatjuk. Ez az azonosítás sokféleképpen megvalósítható, megadható az első elem címe és a másolandó elemek száma vagy megadható az első és az utolsó elem címe, illetve megadható az első elem címe és az utolsó utánié, amely egyben az első olyan elem, amelyet már nem kell másolni. A C++ szabványos függvénykönyvtár a harmadik lehetőséget választotta. Ennek következtében a tartományok félig nyitott tartományokként adhatók meg, ahogyan ezt az 1.1. ábra szemlélteti.



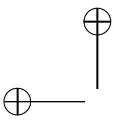
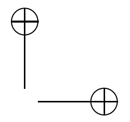
1.1. ábra. Félig nyitott tartomány

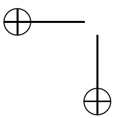
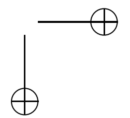
A mi másolófüggvényünk a `[begin1,end1)` tartományból másolja át az elemeket a `begin2` -vel kezdődő tartományba. A céltartomány esetében nem kell megadnunk a tartomány végét.

A mi általános másolófüggvényünk most már bármilyen tárolóból bármilyen más tárolóba át tudja másolni az elemeket. A forrástartomány lehet például egy láncolt lista, a céltartomány pedig egy klasszikus egydimenziós tömb. Ebben az esetben az Input típus egy listaelem-mutató típus lesz, az Output pedig egy klasszikus tömbelem-mutató. A következő példában, az egyszerűség kedvéért két egydimenziós tömb között fogunk másolást végrehajtani. Az algoritmus feltételezi, hogy a céltartomány fizikailag létezik, tehát az algoritmus nem végez helyfoglalást.

```
int x[]={ 10, 20, 30, 40, 50};
int y[ 5 ];
mycopy(x, x+5, y);
```

Vegyük észre, hogy a paraméterezett verem osztály használatakor megadtuk a konkrét T típust, amikor példányosítottuk az osztályt. A paraméterezett függvény esetében ez viszont nem szükséges, hiszen az aktuális paraméterekből egyértelműen kikövetkeztethető az `Input` és az `Output` típus. A mi esetünkben $int *$ fog megfelelni úgy az `Input`, mint az `Output` típusoknak.

2. FEJEZET

A C++ PROGRAMOZÁSI NYELV

A C++ programozási nyelvet a C programozási nyelv ismeretére építjük. Így csak az újdonságokat, illetve a különbségeket fogjuk ismertetni. Mivel az objektumorientált programozási paradigma is már ismert, ezér itt is csak ennek C++ megvalósítását ismertetjük. Ebben a részben a következő újításokat ismertetjük:

1. Logikai típus
2. Névterek
3. Kimeneti/Bemeneti műveletek
4. Referencia (hivatkozás) típus
5. Dinamikus helyfoglalás
6. Kivételkezelés

2.1. Logikai típus

A szabványos C nem tartalmaz külön logikai típust. A logikai műveletek eredményét az egész típus segítségével fejezzük ki. Minden nullátol különböző egész érték logikai igazat jelent, míg a nulla logikai hamisat jelent. C++ nyelvben létezik külön logikai típus, a `bool`. Ennek a típusnak két értéke lehet igaz (`true`), illetve hamis (`false`).

A logikai típust logikai kifejezésekre használjuk, illetve gyakran a függvény visszatérítési értékét is logikai típussal fejezzük ki. Például:

```
bool folytatas=(i < n);
bool nagyobb(int a, int b){ return a>b;}
```

2.2. Névterek

A szabványos C egyik alapvető problémája a névütközések, amelyek főképp nagy programok esetében kritikusak. Tételezzük fel, hogy az elkészítendő programunknak két létező függvénykönyvtárral kell együttműködnie. Mindkét függvénykönyvtárban van egy-egy `func` nevű függvény, amelyet a C fordító névütközésnek minősít és nem fog lefordítani. Az ilyen jellegű problémák kényelmes megoldására vezették be a névtereket. A Java nyelv ugyanezt a problémát a csomagokkal (packages) oldja meg.

A névterek logikailag összetartozó deklarációkat, definíciókat tartalmaznak. Természetesen két különböző névtér tartalmazhatja ugyanazt a függvénynevet, típusnevet, hiszen most már van lehetőségünk ezekre különbözőképpen hivatkozni. A névtér tagjait a következő jelölés használatával kell bevezetni:

```
namespace névtér_azonosító{
   //deklarációk é definíciók
}
```

A névterek segítségével programunk logikailag különálló részei jól elkülöníthetőek. A névtér egyben hatókör is. Ilyen értelemben a közönséges lokális hatókörök, globális hatókörök és maguk az osztályok is névtereknek tekinthetőek. Egy adott névtéren belül minősítés nélkül használhatunk minden egyes nevet, amelyet az adott névtérben vezettünk be. Ha más névterekből akarunk neveket használni, akkor ezeket minősítenünk kell a névtér és a hatókör operátor segítségével:

```
névtér_azonosító::név
```

```
namespace A{
   struct Jo{...};
   ...
}

namespace B{
   A::Jo v;
   ...
}
```

Ha egy névtérben levő összes nevet akarjuk elérhetővé tenni, akkor használhatjuk a `using` direktívát. Ezután már nem szükséges minden nevet külön minősíteni, ezeket úgy használhatjuk mintha ugyanabban a névtérben lennének deklarálva. Az előző példa a következőképpen nézne ki a using használatával:

```
namespace A{
   struct Jo{...};
   ...
}

namespace B{
   using namespace A;
   Jo v;
   ...
}
```

A névterek nyitottak, vagyis bármikor kiterjeszthetőek és akárhány forrásfájlban elhelyezhetőek. Például helyezzünk egy új típust az `A` névtérbe.

```
namespace A{
   struct Rossz{...};
   ...
}
```

A fenti kódrészlet egy `Rossz` nevű struktúrával terjeszti ki a névteret. A standard könyvtár az `std` névtérben van.

2.3. Kimeneti/Bemeneti műveletek

A C++ új lehetőségeket teremt a bemeneti és kimeneti műveletek megvalósítására, bevezetve erre két operátort és néhány szabványos adatfolyamot. A szabványos C `stdin, stdout, stderr` adatfolyamait a `cin, cout` és `cerr` adatfolyamokkal helyettesíthetjük. A `cin` egy bemeneti adatfolyam, a `cout` és `cerr` egy-egy kimeneti adatfolyam. Ezen adatfolyamok használata nagyon egyszerű, egy beolvasást például a következőképpen végezhetünk:

```
int i;
cin>>i;
```

A kíiratás is hasonlóképpen egyszerű:

```
cout<<i;
```

Figyeljük meg, hogy az operátorok mindig jelzik a művelet irányát. A `cin>>i` művelet esetében a beolvasás forrása a bemeneti adatfolyam `cin` és célja pedig az `i` nevű változó. A kíiratás `cout<<i` esetében a forrás az `i` a cél pedig a `cout` kimeneti adatfolyam. Úgy a cin mint a cout a szabványos névtér részei. Ha hatókör nélkül akarjuk használni, akkor kötelező a `using` direktíva használata. Egy teljes program a következőképpen nézne ki:

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

int main(){
   cout<<"Hello\n";
   return 0;
}
```

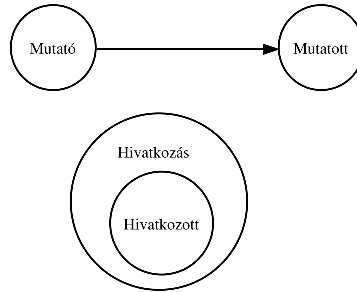
Megfigyelhetjük, hogy a C++ fejfájlok esetében a kiterjesztés megadása már nem kötelező. A szabványos C fejállomáanyai esetében sem kell használni a `.h` kiterjesztést, ezeket viszont a `c` betűvel kell kiegészíteni.

```
#include <stdio.h> //C nyelv

#include <cstdio> //C++ nyelv
```

A saját fejállományok esetében továbbra is használjuk a `.h` kiterjesztést.

2.4. Referencia (hivatkozás) típus

A C++ hivatkozás típus a konstans mutatóhoz hasonlítható, azaz nem is igazi változó, hiszen önállóan nem létezik, hanem csak a hivatkozott objektumon keresztül. Ez azt jelenti, hogy egy hivatkozás típusú változót úgy lehet létrehozni, ha rögtön megadjuk a hivatkozott objektumot is. A létrehozás pillanatától szorosan hozzátapad a hivatkozott objektumhoz és minden művelet, amit a



2.1. ábra. Mutató és hivatkozás összehasonlítása

referenciával végzünk, az a hivatkozott objektumra vonatkozik. Jól látható, hogy a referencia gyakorlatilag egy másik nevet jelent amelyen keresztül az objektum kezelhetővé válik. Egy adott referencia mindig szorosan hozzátapad a hivatkozott objektumhoz és nem választható le róla.

A mutató és hivatkozás közötti különbséget jól szemlélteti a 2.1 ábra.

Habár a hivatkozásokat főképp függvényparaméterként és függvény által visszaadott típusként használjuk, lehetséges az önálló referenciák deklarációja is. A következőkben ezeket mutatjuk be.

2.4.1. Önálló referenciák

```
1. int i = 10;
2. i++; // i=11
3. int ri& =i;//i=11  ri=11
4. ri++;//i=12  ri=12
```

A fenti program első sora egy egész típusú változót vezet be, amelynek neve `i` és kezdőértéke `10`. A második sor eggyel növeli a változó értékét. A harmadik sor egy `ri` nevű referenciát vezet be, amelyet kötelező módon inicializál az `i` változóval. Így tulajdonképpen `i` az igazi objektum és `ri` egy másik név, amelyen keresztül elérhető az `i` változó. A 4. sor eggyel növeli a hivatkozott változó értékét. Ha a 4. sor után kiíratnánk az `i` és `ri` változókat, akkor mindkét kiíratás ugyanannak az `i` változónak az értékét írná ki.

2.4.2. Referencia típusú függvényparaméter

A referenciák egyik elegáns alkalmazása a paraméterátadás. Függvények paramétereit kétféleképpen adhatjuk át: érték, illetve cím szerint. A cím szerinti paraméterátadást akkor használjuk, ha a függvénynek meg kell változtatnia a paraméter értékét. A cím szerinti paraméterátadás egyrészt lehetővé teszi a paraméter megváltoztathatóságát a függvény blokkjában, másrészt pedig gyorsítja a függvényhívást, hiszen tárolók esetén csak a tároló címe kerül átadásra. Feltevődik a kérdés, hogy milyen újdonságot hoz a referencia szerinti paraméterátadás. Gyakorlatilag csak kényelmesebbé teszi a programírást, hiszen ez a cím szerinti paraméterátadás egy másik megvalósítási formája. Hogy jobban megértsük, tekintsünk egy példát, amelyben a függvénynek meg kell változtatnia az átadott paramétereket. Legyen a függvény feladata két változó cseréje. Ezt szabványos C nyelvben a következőképpen készíthetnénk el:

```
void swap(int *a, int *b){
   int c = *a; *a= *b; *b = c;
}
```

Ha a paramétereket referenciaként adjuk át, akkor a függvényen belül kényelmesen hivatkozhatunk a cserélendő változókra a * operátor használata nélkül. Ez a következőképpen nézne ki:

```
void swap(int&a, int&b){
   int c = a; a =b; b = c;
}
```

Természetesen ezt C++ nyelvben sokkal általánosabban is kifejezhetjük függvénysablon segítségével:

```
template <class T>
void swap(T& a, T& b){
   T c = a; a =b; b=c;
}
```

A fenti függvénysablon segítségével bármilyen, olyan típusú változók értékei kicserélhetők, amelyekre helyesen működik az értékadó operátor.

2.4.3. Függvény által visszatérített referencia

Ha egy függvény referenciát térít vissza, akkor a függvény balértékként is használható. Tekintsük erre a következő példát:

Adott egy szigorúan pozitív elemeket tartalmazó, n ($n \geq 10$) elemű tömb. Készítsünk függvényt, amely helyettesíti a tömb 10 legnagyobb elemét -1 értékkel.

```
double& max(double *x, int n){
   double m = x[ 0 ]; int poz =0;
   for(int i=1;i<n;i++)
      if(x[ i ] > m){ m=x[ i ]; poz =i; }
   return x[ poz ];
}

int y[100], ny=100;
//y feltöltése

for(i=1; i<=10; i++)
   max(y, ny)=-1;
```

2.4.4. Konstans referenciák

Önálló referencia típusú változót inicializálni kell a deklaráció pillanatában. A referencia kezdőértéke kötelező módon változó kell legyen. Konstans referenciát viszont lehet változóval, vagy konstans kifejezéssel is inicializálni. A következő példák ezt szemléltetik:

```
int x = 10;
const int& r1 = 57+3;
const int& r2 = x;
```

Konstans referencián keresztül nem lehet megváltoztatni a hivatkozott objektumot (`r2++` hibának minősül). Konstans referenciát leggyakrabban függvényparaméterként használunk olyan esetben, amikor hatékony paraméterátadásra van szükségünk és a függvénynek nem szabad megváltoztatnia a paraméter értékét. Gyakorlatilag így hatékonnyá és biztonságossá tesszük a paraméterátadást.

Mutatók esetében lehet deklarálni mutatót konstans objektumra (`const int *p=&x`) és konstans mutatót objektumra (`int * const p=&x`). A konstans mutató esetében nem lehet megváltoztatni a mutató értékét, vagyis mindig ugyanarra az objektumra mutat, amelyet kezdőértékként beállítottunk (`p++, p--` hibának minősül). Ha viszont mutatót deklarálunk konstans objektumra akkor a mutatott tárterület a konstans, ezt nem lehet a mutatón keresztül megváltoztatni (`(*p)++` hibának minősül).

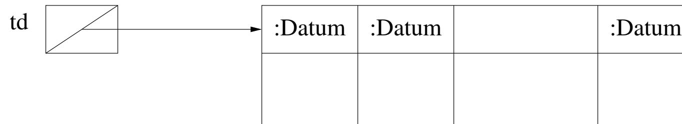
Referenciák esetében a konstans referencia mindig azt jelenti, hogy a hivatkozott objektum értékét nem lehet megváltoztatni a referencián keresztül. Itt nem is lenne értelme a mutatóknál létező második esetnek, amikor a mutatót minősítettük konstansnak, hiszen a referencián nem lehet műveletet végezni. A látszólag referenciára kiadott művelet mindig a hivatkozott objektumon hajtódik végre.

2.5. Dinamikus helyfoglalás

A helyfoglalásnak két típusa ismeretes: a statikus és a dinamikus helyfoglalás. E két helyfoglalás merőben különbözik egymástól. A statikus helyfoglalású változóknak a fordító végzi a helyfoglaláast, felszabadításuk automatikusan történik a hatáokör megszünésével. Ezeknek a változóknak a futó folyamat memóriájának a verem szegmens részében foglalódik hely. A statikus helyfoglalás gyors, hiszen nem vesz igénybe futási időt, hátránya pedig az, hogy pontosan kell ismernünk a változóknak szükséges tárterületet már a programírás pillanatában.

A dinamikus helyfoglalást egy igény szerinti helyfoglalásnak tekinthetjük, amely futás közben hajtódik végre a memória úgynevezett heap (halom) szegmensében. Természetesen, ha egy idő után már szükségtelenné válik a lefoglalt tárterület, akkor ez futásidőben felszabadítható. Amíg a statikus helyfoglalású változók kezelésének feladata teljesen a fordítóra hárul, addig a dinamikus helyfoglalású változók kezelése teljesen a programozó felelőssége.

Dinamikus helyfoglalásra és felszabadításra is új operátorokat vezetett be a C++ nyelv. Megkülönbözteti az egyszerű és a többszörös helyfoglalást, az előbbire a `new` az utóbbira pedig a `new[]` operátort vezette be. A felszabadításra



2.2. ábra. Dinamikus helyfoglalás objektumtömbnek

szintén két operátorunk van a `delete` és a `delete[]`. A következőkben tekintsünk egy pár példát a helyfoglalásra.

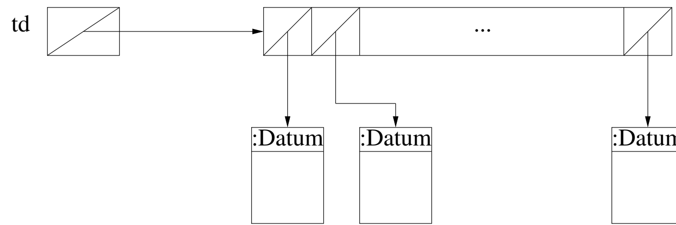
```
int *a = new int; //Helyfoglalás egyetlen egész számnak;
*a = 10;
delete a; //Felszabadítás

int *t = new int[ 10 ]; //Helyfoglalás egy 10 elemű
                        //egészekből álló tömbnek
for(int i=0;i<10;i++) t[ i ] = 10;
delete [ ] t;//Felszabadítás
```

Tételezzük fel, hogy létezik egy `Datum` osztályunk, amelynek van paraméter nélküli konstruktora. Készítsünk ebből az osztályból dinamikus helyfoglalású példányokat.

```
Datum *d= new Datum();
//Műveletek a *d objektummal
delete d;

Datum *td = new Datum[ 10 ];
//Műveletek a td tömbbel
delete [ ] td;
```

A dátumobjektumokat tartalmazó tömböt a 2.2. ábra szemlélteti.

A `new` operátor hatására nemcsak helyfoglalás történik, hanem egyben konstruktorhívás is. Pontosan ennek következtében, ha objektumtömböt akarunk létrehozni, akkor kötelező a paraméter nélküli konstruktor megléte, hiszen paraméterek megadására nincs lehetőség.



2.3. ábra. Dinamikus helyfoglalás

Ha nem megfelelő a paraméter nélküli konstruktor, akkor létrehozhatunk egy mutatótömböt, majd a mutatókat egyenként inicializálhatjuk a megfelelő konstruktorhívással. Ezt a 2.3. ábra szemlélteti.

```
int i;
Datum ** d= new Datum *[ 10 ];
for(i=0; i<10; i++)
   d[ i ] = new Datum(2005, 10, i+1);
//Műveletek
for(i=0; i<10; i++)
   delete d[ i ];
delete[ ] d;
```

2.6. Kivételkezelés

A kivételkezelés ma már minden modern programozási nyelv szerves része. Tulajdonképpen a hibakezelés általánosításának is tekinthető. A hibakezelésnek logikailag két különálló része van: a hibaesemény jelzése és a máshol észlelt hibák kezelése. Kivételkezeléssel oldható fel minden olyan hiba, amelyet helyben nem lehet megszüntetni. A kivételkezelés másik nagy előnye, hogy lehetővé teszi a hagyományos kód és a hibakezelő kód szétválasztását.

Tételezzük fel, hogy egy függvénykönyvtárat készítünk, amelyet nagyon sok programhoz fogunk illeszteni. A könyvtár létrehozója azonosítani tudja a futásidejű hibákat, de általában nem tudja helyben kezelni, mert ez a felhasználói program feladata. A függvénykönyvtár készítőjének az a feladata, hogy valami-

lyen formában jelezze az ilyen jellegű futásidejű hibákat. Ez általában kivételesemény formájában történik. Amíg a Java nyelvben a kivételes állapotot egy dinamikus helyfoglalású kivételobjektum ábrázolta, a C++ nyelvben a kivételeseményt jelezni lehet egy primitív típusú változó segítségével is.

Kivétel kiváltása: `throw`

```
class Error{...};
void f(){
   if(feltétel){
      throw Error()
   }
   ...
}
```

Kivétel lekezelése: `try-catch`

```
try{
   f();
}
catch(Error){
   //kivételkezelő blokk
}
```

Kivételt lekezelő utasítást csak olyan kódrészlet esetében használhatunk, amelyben előállhat a kivételes állapot. A mi esetünknkben az `f` függvény kiválthat `Error` típusú kivételt, tehát használhatjuk az `f` függvény hívását kivételkezelő blokkban (`try{}`). A `catch` utasítást csak `try` utasítás után használható. Egy `try` utasítást több darab `catch` is követhet, amennyiben a `try{}` blokkban elhelyezett utasítások többféle kivételt is kiválthatnak. A `catch` után zárójelben kötelező megadni a kivétel típusát. Nem kötelező, de megadható maga a kivételobjektum is. Abban az esetben adjuk meg a kivételobjektum nevét, ha valamilyen műveletet szándékozunk végezni a kivételobjektummal . A C++ nyelvben bármilyen primitív típusú változóval is lehet kivételt ábrázolni. Tanácsos azonban a kivételekre külön típusokat bevezetni, hiszen igazán így tudjuk a program kivételkezeléssel kapcsolatos részeit elkülöníteni.

A fentiek szemléltetésére készítsünk egy karaktervermet és lássuk el hibakezeléssel.

stack.h

```
#ifndef stack_h
#define stack_h
using namespace std;
class Stack{
public:
   class Stack_Overflow{};
   class Stack_Underflow{};
   virtual void push(char)=0;
   virtual char pop ()=0;
};

class Array_Stack : public Stack{
private:
   char *s;
   int top;
   int max_size;
public:
   Array_Stack(int _max_size);
   Array_Stack(const Array_Stack&);
   ~Array_Stack();
   void push(char c) throw (Stack_Overflow);
   char pop () throw (Stack_Underflow);
};
#endif
```

A fenti fejállomány két osztálydeklarációt tartalmaz. A `Stack` osztály valójában egy olyan absztrakt osztály, amely csak típusdefiníciókat és függvénydeklarációkat tartalmaz. Tehát támogatja az absztrakt adattípusokat, hiszen nem ad meg semmiféle információt az adatok ábrázolására, illetve a műveletek implementációjára vonatkozóan. Az ilyen absztrakt osztályokat felületeknek is nevezzük, mert felületet biztosítanak más osztályok számára. Erre az osztályra építjük rá az `Array_Stack` konkrét osztályt, amely az absztrakt osztály konkrét leszármazottja lesz, megvalósítva egy adott stratégiának megfelelően a verem felületet. Természetesen más lehetséges megvalósításokat is el tudunk képzelni. Lehetséges a vermet láncolt listával is ábrázolni. Az egyszerűség volt a döntő érv, amikor a tömbös implementáció szemléltetése mellett döntöttünk. Az osztályunknak két konstruktora van, az első konstruktor szerepe egyértelmű, a másodikat másoló

konstruktornak nevezzük és olyan esetekben használjuk amikor egy új vermet egy létező verem másolatával szeretnénk inicializálni. Ezt bővebben majd a következő fejezetben fogjuk kifejteni. Mivel a C++ nyelvben nem létezik automatikus szemétgyűjtés, ezért szükség van egy megsemmisítő metódusra, amely többek között az objektum dinamikus helyfoglalású részeit megsemmisíti. Ezt a metódust destruktornak nevezzük és a konstruktorral ellentétes műveleteket végez.

stack.cpp

```
#include "stack.h"
#include <iostream>
using namespace std;

Array_Stack :: Array_Stack(int _max_size){
   max_size = _max_size;
   s = new char[ _max_size ];
   top = 0;
}

Array_Stack :: Array_Stack(const Array_Stack& _as){
   this->max_size = _as.max_size;
   this->s = new char[ max_size ];
   this->top = _as.top;
}

Array_Stack :: ~Array_Stack(){
   cout<<"\nDestructor: "<<this->max_size<<"\n";
   delete []s;
}

void Array_Stack ::push(char c) throw(Stack_Overflow ){
   if(top >= max_size) throw Stack_Overflow();
   s[ top++ ] = c;
}
char Array_Stack :: pop() throw (Stack_Underflow){
   if(top == 0) throw Stack_Underflow();
   return s[ --top ];
}
```

test_stack.cpp

```
#include <cstdio>
#include "stack.h"
#include <iostream>
using namespace std;

int main(){
   Array_Stack as(100);
   try{
      printf("\n%c", as.pop());
   }
   catch(Stack:: Stack_Underflow x){
      cout<<"Stack_Underflow"<<"\n";
   }
   return 0;
}
```

3. FEJEZET

OSZTÁLYOK ÉS OBJEKTUMOK

A típus egy fogalom konkrét ábrázolása, amelyet egyértelműen meghatároz két halmaz: a típushoz tartozó értékek halmaza, illetve a típuson végezhető műveletek halmaza. Például a 2 bájtos előjeles egészek halmazához -32768..32767 közötti értékek taroznak, amelyeken a megszokott aritmetikai, relációs stb. műveletek értelmezhetők.

Minden programozási nyelv rendelkezik egy alapértelmezett típushalmazzal és lehetőséget biztosít ennek felhasználói típusok általi kiterjesztésére. Felhasználói típust C nyelvben a `struct` kulcsszóval vezethetünk be. Természetesen akkor kell új típust bevezetni, ha egy olyan fogalomra van szükség, amelynek nincs közvetlen megfelelője a beépített típusok között.

A C++ nyelvben a struktúra típus is átalakult, ugyanis már nemcsak adattagokat, hanem metódusokat is tartalmazhat. A C++ struktúra olyan osztály, amelynek minden tagja nyilvános. Osztályokat lehet deklarálni illetve definiálni. Az osztálydeklaráció csak a `class` kulcsszót és az osztály nevét tartalmazza. Például:

```
class Macko;
```

Az osztálydefiníciót általában két forrásfájlban szokás elhelyezni. A fejállományban az osztálydefiníció az adattagokat illetve a metódusdeklarációkat tartalmazza és a metódusok definicióit egy C++ forrásfájlban szokás elhelyezni úgy, ahogyan a verem modul esetében is tettük.

Osztálydefinícióra tekintsünk egy nagyon egyszerű példát, éspedig készítsük el a felhasználói dátum típust. Egy dátumot három egész számmal adhatunk meg. Első megközelítésben a dátum osztály ezt a három adatmezőt fogja tartalmazni, amelyeket a konstruktor inicializál. Ez a következőképpen nézne ki:

```
class Datum{
   private:
      int ev, ho, nap;
   public:
```

```
        Datum(int _ev, int _ho, int _nap);
};
```

Egy osztály tagjai háromféle láthatóságúak lehetnek:

- privát (private)
- védett (protected)
- nyilvános (public)

A láthatóság meghatározza ezen tagok elérhetőségét. Ennek következtében a privát tagok csak az osztályon belül érhetőek el, a védett tagok az osztályon belül és a leszármazott osztályokban, a nyilvános tagok pedig bárhol elérhetők. Ez a háromféle láthatóság teljesen úgy működik, mint Java nyelveben, az egyetlen különbség a megadási mód. Amíg Javaban minden egyes osztálytagra külön adjuk meg a láthatósági módosítót addig a C++ nyelvben csoportokra adhatjuk meg. Egy osztályon belül többször is megengedett ugyanazon láthatósági módosító, de lehetőleg kerüljük ezt és gyűjtsük egy helyre a privát, a védett és a nyilvános osztálytagokat, mert áttekinthetőbb lesz a programunk.

3.1. Implicit paraméterek

Egy teljesen specifikált dátum három értékkel adható meg. Ennek ellenére gyakran azt a kérdést tesszük fel, hogy hányadika van, vagyis ilyenkor csak a nap sorszámára vagyunk kíváncsiak, hiszen mindenkinek magától értetődő, hogy az aktuális hónapról és az aktuális évről van szó. A felsoroltak következtében gyakran célszerű, ha egy objektum kezdőértékét többféleképpen is megadhatjuk, ehhez viszont az szükséges, hogy bizonyos paraméterek implicit értéket kapjanak.

A verem osztály esetében például lehetne egy implicit méret, azaz ha a felhasználó történetesen nem specifikálja a verem méretét, akkor az az implicit értéknek megfelelő méretű lesz. Implicit paraméter megadása a következőképpen történik:

```
Stack(int _size = 100){ ... }
```

```
Stack s1, s2(30);
```

A fenti kódrészletben a verem osztály konstruktorának van egy implicit értékű paramétere. Az s1 verem objektum esetében a konstruktor nem kap paramétert, így ennek implicit értékét fogja használni, létrehozva egy száz elemű vermet. Az s2 objektum esetében már nem kerül felhasználásra a paraméter implicit értéke, hiszen deklarációkor specifikáltuk a méretet. Ennek következtében az s2 egy 30 elemű verem lesz.

A dátum esetében az implicit érték az aktuális dátum lehetne, vagyis ha egy dátum valamely komponense hiányzik, ezt az aktuális dátum megfelelő komponensével lehetne helyettesíteni. Ezt viszont lehetetlen implicit értékként betenni, hiszen naponta változik. A megoldás az lenne, hogy az implicit értéket csak jelzésre használjuk, és amennyiben a paraméter nem kapott értéket és ennek következtében az implicit értékkel maradt, helyettesítjük az aktuális dátum megfelelő komponensével. Tételezzük fel, hogy létezik egy mai_datum Datum típusú változó, amelyből beállíthatóak a dátum mezői. A Datum osztály konstruktora a következőképpen mutatna:

```
Datum(int _ev = 0, int _ho=0, int _nap = 0){
   ev  = (_ev  == 0) ? mai_datum.ev  : _ev;
   ho  = (_ho  == 0) ? mai_datum.ho  : _ho;
   nap = (_nap == 0) ? mai_datum.enap: _nap;
}
```

3.2. Statikus tagok

Az osztályok lehetőséget biztosítanak olyan tagok deklarációjára, amelyek a típushoz (osztályhoz) tartoznak és nem a példányokhoz. Az osztályhoz tartozó tagokat statikus tagoknak nevezzük és adat illetve metódus tagok lehetnek. A statikus adattagok létrejönnek még a példányok előtt és ezeket közösen használhatják a példányok. Ha egy osztály példányait szertnénk számlálni, akkor egy ilyen számláló típusú adattagot statikusnak kellene deklarálnunk, hiszen ez a típushoz tartozó adat és nem pedig a példányhoz. A statikus tagokat a static kulcsszóval vezetjük be és adattagok esetében ezek csak deklarációt jelentenek, a

létrehozást egy osztályon kívüli blokkban kell végezni. A következő példa szemlélteti a statikus tagok deklarációját, definícióját és használatát.

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

class InstanceCounter{
private:
   static int counter; //Deklaráció
   int i;
public:
   InstanceCounter(int _i = 0): i(_i){ counter++; }
   ~InstanceCounter(){ counter--; }
   static int getCounter(){ return counter;}
};

int InstanceCounter::counter = 0; //Definició

int main(){
   cout<<"Peldanyszam: "<<InstanceCounter::getCounter()<<endl;
   InstanceCounter o1, o2, o3;
   cout<<"Peldanyszam: "<<InstanceCounter::getCounter()<<endl;
   InstanceCounter * p = new InstanceCounter(10);
   cout<<"Peldanyszam: "<<InstanceCounter::getCounter()<<endl;
   delete p;
   cout<<"Peldanyszam: "<<InstanceCounter::getCounter()<<endl;
   return 0;
}
```

A statikus tagok nem a példányhoz, hanem az osztályhoz tartoznak. Amíg a példány tagjaira a `.` operátor segítségével hivatkozhatunk, addig a statikus tagokra a `::` hatókör operátor segítségével:

```
osztálynév :: statikus_tag
```

Az `InstanceCounter` osztály esetében az osztályon kívüli kódnak csak a statikus metódushoz van hozzáférése, hiszen a `counter` statikus adattag privát. A statikus metódusok nem férnek hozzá a példányszintű tagokat. Ez teljesen logikus, hiszen amint már említettük, ezek nem a példányon dolgoznak. Ezzel szemben a példányok nyugodtan használhatják a statikus tagokat. A mi példánk esete-

ben a konstruktor is meg a destruktor is használja a `counter` statikus adattagot. Mivel konstruktorhívás új példányok esetében történik és minden példányra csak egyszer, teljesen logikus, hogy itt növeljük a példányszámlálót. Destruktorhívás is csak egyszer történik minden példány esetében, ezért a példány megsemmisítése előtt eggyel csökkentjük a példányszámlálót.

3.3. Konstruktor és destruktor

A konstruktor feladata az objektum kezdőértékének beállítása. Kijelenthetjük, hogy a konstruktorok olyan metódusok lesznek, amelyek nem térítenek vissza semmilyen értéket, tehát csak mellékhatást fejtenek ki. A konstruktor mindig akkor hívódik meg, amikor már az objektumnak megtörtént a helyfoglalás és a lefoglalt tárterületet inicializálja. A konstruktor tehát nem foglal helyet az objektumnak. Ha az objektumnak történetesen vannak mutató típusú adattagjai, a konstruktor ezen adatmezőknek végezhet dinamikus helyfoglalást.

A destruktor az objektum felszabadítása előtt hajtódik végre és a konstruktorral ellentétes feladatokat végez. Leggyakrabban a destruktort olyan erőforrások felszabadítására használjuk, amelyeket a konstruktorban kötöttünk le. A leggyakoribb ilyen erőforrás a memória.

Egy osztálynak akárhány konstruktora lehet, de mindig csak egy destruktora. Mindekettő az osztály nevét viseli, a destruktor esetében ezt a nevet megelőzi a ~ szimbólum. Amíg a konstruktorok paraméterezhetőek, addig a destruktornak nem adhatunk paramétert.

Egy osztály konstruktorai három kategóriába sorolhatóak:

- alapértelmezett (implicit) konstruktor
- másoló (copy) konstruktor
- más konstruktor

Az első két kategóriát részletesen ismertetjük. A harmadik kategóriát azok a konstruktorok alkotják, amelyek nem sorolhatóak be az első kettőbe.

3.3.1. Alapértelmezett (implicit) konstruktor

Azt a konstruktort, amelyet paraméter nélkül hívhatunk meg, alapértelmezett konstruktornak nevezzük. Ezt a típusú konstruktor megadhatja a programozó vagy ha ez nem adná meg, akkor létrehoz egyet a fordító. A fordító csak abban az esetben generál konstruktort, ha a programozó egyáltalán nem adott meg semmiféle konstruktort. Ha a programozó megadott egy akármilyen típusú konstruktort, a fordító már nem fog konstruktort generálni. A csak implicit paraméterekkel rendelkező konstruktor is alapértelmezett konstruktor, hiszen ez is paraméter nélkül hívható. A fordító által generált konstruktor automatikusan meghívja az osztály típusú tagok és az ősosztályok alapértelmezett konstruktorait. Minden beépített típusnak van alapértelmezett konstruktora. Példák alapértelmezett konstruktorra:

```
class Stack{
private:
   char *elements;
   int size;
   int sp;
public:
   Stack(int _size = 100){...}
   ...
};
```

Vannak esetek, amikor a fordító által létrehozott alapértelmezett konstruktor nem megfelelő. A fenti verem osztály esetében az alapértelmezett konstruktor NULL-al inicializálná az `elements` adattagot, ami semmiképpen sem lenne elfogadható. Ha egy osztálynak konstans, vagy referencia típusú adattagjai vannak, akkor ezeket sem lehet alapértelmezett módon inicializálni. Például:

```
class A{
   const int  ca;
   const int& ra;
};
```

Mikor hívódik meg az alapértelmezett konstruktor?

```
Stack s;
```

```
Stack ts[ 10 ];
```

A fenti kódrészletben, úgy az egyszerű Stack típusú objektum esetében, mint a 10 elemű tömb esetében az implicit konstruktor fog meghívódni. Mivel 10 elemű tömbünk van, ezért összesen 11 darab konstruktorhívást eredményez a fenti kódrészlet. Ha a Stack osztálynak nem lenne implicit konstruktora, akkor a fenti kódrészlet fordítása hibát eredményezne.

3.3.2. Másoló konstruktor

Amint a neve is mutatja, a másoló konstruktort objektumok másolására használjuk. Ebben a részben azt fogjuk megvizsgálni, hogy mikor szükséges ilyen típusú konstruktort megadni, illetve hogyan történik az objektumok másolása másoló konstruktor hiányában. Tekintsük a következő példát:

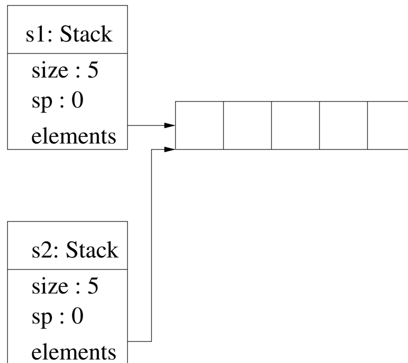
```
Datum d1 (2005, 10, 3);
Datum d2 =d1;
```

A fenti kódrészlet két különböző Datum típusú objektum létrehozását eredményezi. Mivel a d2 objektumot a d1 objektummal inicializáltunk, ezért ezen két objektum állapota meg fog egyezni, ugyanis megfelelő adatmezőik azonos értékeket tartalmaznak. A Datum osztályban nem adtunk meg másoló konstruktort. Ilyen esetben a `Datum d2 =d1;` utasítás a mezők bitenkénti másolatát eredményezi. A Datum osztály esetében ez a fajta másolás tökéletesen megfelel a célnak, hiszen az osztály csak primitív típusú adattagokat tartalmaz.

Nézzük meg hasonlóan jó lesz-e ez a fajta másolás az előző részben megadott Stack osztály esetében, illetve milyen rendellenességek történnének ha erre a konstruktorra hagyatkoznánk.

```
Stack s1(5);
Stack s2 =s1;
```

Amint az 3.1. ábra is szemlélteti, a másolás becsületesen végrehajtódott és ennek következtében a két objektum elements adattagja ugyanarra a tárhelyre mutat. Valószínűleg egyetlen programozó sem akarna ilyen veremmel dolgozni.



3.1. ábra. Bitenkénti másolás veremobjektumok között

Egy másik probléma akkor következik be, amikor megszűnik a két objektum hatásköre és destruktorhívás történik az s1 és s2 objektumokra. Az s1 objektumra szépen lefut a destruktor, felszabadítva az `elements`-nek lefoglalt tárterületet. Az s2 esetében azonban hibaüzenettel fog leállni a programunk, hiszen megpróbálja másodszor is felszabadítani ugyanazt a tárterületet.

Levonhatjuk a következtetést: olyan esetekben amikor az osztálynak van mutató típusú adattagja, másoló konstruktor megadása kötelező. A Stack osztály esetében ez a következőképpen adható meg:

```
class Stack{
   ...
public:
   Stack(const Stack& s);
   ...
};

Stack::Stack(const Stack& s){
   size = s.size;
   sp = s.sp;
   elements = new char[ size ];
   for(int i=0; i<size; i++)
      elements[ i ] = s.elements[ i ];
}
```

A másoló konstruktor általános alakja egy `X` osztály esetében: `X(const X&)` és a következő esetekben használódik:

1. Új objektum létrehozása egy létező lemásolásával.

```
X o1 =o2;  X o1 (o2);
```

2. Érték szerint átadott objektum.

```
void f(X o){ ... }
X obj;
f(obj);
```

Ebben az esetben a függvényhívás formális paramétere az `o` objektum az `obj` objektum másolataként jön létre.

3. Függvény által visszatérített objektum.

```
X f2(){ }
```

Ha objektumot térítünk vissza, a függvényhívás befejeztével meghívódik a másoló konstruktor.
4. Kivételek esetében

```
void f(){
   ...
   ....throw X();
}

try{
   f();
}
catch(X o){...}
```

3.4. Objektumok

Az objektumok a szabványos C változókhoz hasonlóan kétféleképpen hozhatók létre: statikusan és dinamikusan. A statikus helyfoglalású objektumok helyfoglalása és felszabadítása a fordítóprogram felelőssége. Úgy a helyfoglalás, mint a felszabadítás automatikusan történik, ahogyan a szabványos C automatikus

változóinak. A dinamikus helyfoglalású objektumok futás közben jönnek létre, a memória halom (heap) szegmensében és fölöslegessé válás esetében felszabadíthatóak. Ne feledjük, hogy a dinamikus helyfoglalás egy költséges (időigényes) művelet, ezért csak indokolt esetben használjuk. A következő kódrészlet példát ad mindkét helyfoglalású objektumra.

```
Datum d1, d2(2005, 10, 3);
Datum * d3 = new Datum;
Datum * d4 = new Datum(2005, 10, 3);
```

A fenti példában a d1 és d2 objektumok statikus helyfoglalásúak. A helyfoglalás fordítási időben történik és szintén a fordító gondoskodik arról, hogy konstruktorhívás is történjen az objektumok létrehozásakor. A d1 objektum esetében az implicit paraméterek állítódnak be az adatmezők kezdőérékeként, a d2 objektum esetében a megadott paraméterek. A d1 és d2 objektumok felszabadítása a fenti függvényblokk befejeztével automatikusan történik. A d3 és d4 két `Datum` típusú mutató, amelyek dinamikus helyfoglalású `Datum` típusú objektumokra mutatnak. A d3 esetében a konstruktor az implicit paraméterekkel inicializálja az objektumot, a d4 esetében pedig a megadott paraméterekkel. Ne felejtsük el felszabadítani a d3 és d4 objektumokat.

```
void f(){
   Datum d;
   ...
}
```

Ha egy objektumot függvény blokkjában definiálunk, akkor az objektum automatikusan megsemmisül a függvényhívás befejeztével. Ez azt jelenti, hogy belépéskor létrejön, vagyis helyfoglalás és konstruktorhívás hajtódik végre és kilépéskor megsemmissül, destruktorhívás és utána felszabadítás történik. Ebben az esetben az objektum a vermen jön létre és onnan fog törlődni a felszabadítás pillanatában. Ez a fajta helyfoglalás egy „olcsó" helyfoglalás.

Ha a függvényben az objektumot dinamikusan hozzuk létre, akkor úgy a helyfoglalást, mint a felszabadítást magunk kell végeznünk.

```
void f(){
```

```
        Datum * p = new Datum;
        ...
        delete p;
    }
```

A fenti példa alapvetően helyes, mert a függvényben létrehozott `Datum` típusú objektumot fel is szabadítja a függvényhívás befejeztével. Abban az esetben, ha a függvény célja egy dinamikus helyfoglalású objektum létrehozása lenne, akkor ezt az objektumot a függvénynek vissza kellene térítenie és a felszabadításra természetesen nem lenne szükség. Nézzük meg, hogy mi történik abban az esetben, ha az `f` függvényben kivétel váltódik ki. Mivel kivétel keletkezése esetén a vezérlés átkerül az `f`-et meghívó blokkra, gyakorlatilag a függvényhívás befejeződik. A függvény első változatában (statikus helyfoglalású objektum), a függvényhívás befejezésekor automatikusan meghívódik a destruktor és megsemmisül a `d` objektum. A második változatban viszont, ha a kivétel a `delete` előtt keletkezik, akkor a `p` által mutatott objektum életben marad és soha nem is lehet már felszbadítani.

3.5. Helyben kifejtett (inline) tagfüggvények

Osztályon belül definiált helyben kifejtett (inline) tagfüggvényeknek esetében a fordítóprogram a függvény hívása helyett közvetlenül beilleszti a függvény kódját. Az inline függvények gyorsítják a program végrehajtását, viszont megnövelhetik a kód méretét. Az inline kifejtés a makróhelyettesítéséhez hasonló. Általánosan elfogadott szabály, hogy csak a kevés utasítást tartalmazó függvényeket szokás `inline` függvényként megadni. Azt is meg kell jegyeznünk, hogy a fordítóprogram egyszerűen el is hanyagolhatja a programozó ilyen típusú kérését.

Egy függvényt akkor tekint helyben kifejtendőnek a fordítóprogram, ha függvény definícióját osztályon belül adjuk meg vagy ha az osztályon kívüli megadásnál használjuk az `inline` kulcsszót.

```
class Datum{
```

```
private:
   int ev, ho, nap;
public:
   ...
   int nap() const{ return nap; }
};
```

VAGY

```
class Datum{
private:
   int ev, ho, nap;
public:
   ...
   int nap() const;
};

inline int Datum::nap() const { return nap;}
```

3.6. Konstans tagfüggvények

A `const` kulcsszót függvény esetében is használhatjuk. Olyan függvényeket látunk el ezzel a kulcsszóval, amelyek nem változtathatják meg az objektum állapotát. Az objektum állapotának lekérdezését végző függvények ebbe a kategóriába tartoznak.

3.7. A this pointer

A this mutató csak példányszintű tagfüggvényekben használható, ez a neve az aktuális objektummutatónak, amelyen éppen hajtódik végre a függvény. A this mutatót használhatjuk az objektum tagjaira való hivatkozásokban, az osztály tagfüggvényeiben. Ez a mutató a függvény hívásakor kap értéket. Tekintsük a következő kódrészletet:

```
class A{
private:
   int a;
public:
   A(int a){ this->a = a;}
   f(int i){ a = 2*i; }
};

A o;
o.f(12);
```

A fenti kódrészletben definiáltunk egy `A` osztályt és példát adtunk az osztály példányosítására, utána pedig a példányon keresztül meghívtuk az `f` példánymetódust. Az `o.f(12)` hívást a fordító a következőképpen fordítja: `f(&o, 12)`. Gyakorlatilag a `this` a `&o` értéket fogja felvenni, ami nem más, mint annak az objektumnak a címe, amely a függvényt meghívta.

A tagfüggvényekben az osztálytagokra hivatkozhatunk a this megadása nélkül is, amennyiben egyértelmű ez a hivatkozás. Az `f` függvényben például az `a` adattagra a `this` nélkül hivatkoztunk. Ugyanígy viszont nem tudunk hivatkozni a konstruktorban, mert a lokális paraméter felülbírálja az osztály hatókörére nézve globális `a` adattagot. Ennek következtében a konstruktorban kötelező az értékadás baloldalán használni a `this` mutatót.

3.8. Barát függvények és osztályok

3.8.1. Barát függvények

A C++ nyelv lehetővé teszi egy osztályon kívüli függvénynek, hogy elérje az objektum összes privát és védett tagját a barát mechanizmuson keresztül. Annak ellenére, hogy ez megsérti az egységbe zárás objektumorientált paradigmát, elég gyakran használt technika. Ha egy függvénynek meg akarjuk engedni, hogy hozzáférjen a privát és védett adattagokhoz is, akkor a függvényt a `friend` kulcsszóval kötelező deklarálni az osztályon belül. Teljesen mindegy, hogy ezt a barát deklarációt az osztály privát vagy nyilvános részébe helyezzük.

```
class Matrix{
private:
   int rows, cols;
   double ** elements;
public:
   ...
   friend Matrix MatrixSum(const Matrix& A, const Matrix& B);
   ...
};
```

A fenti példában a `MatrixSum` függvény nem tagfüggvénye a `Matrix` osztálynak, hanem baráti viszonyban van az osztállyal. A `MatrixSum` függvényen belül nem használható a `this` mutató sem, hiszen ennek csak a példányszintű tagfüggvényekben van értéke. A `MatrixSum` függvény feladata a paraméterül kapott két mátrix összegének kiszámítása, tehát ez a függvény a paraméterein dolgozik.

Felmerül a kérdés, hogy mi a különbség a barát függvények és a statikus függvények között. Induljunk ki abból, hogy mit is jelent, ha egy függvény tagfüggvénye (példány szintű) egy osztálynak. Ez alapvetően három dolgot jelent:

1. *A függvény hozzáférhet az osztály deklarációjának privát részeihez.*
2. *A függvény az osztály hatókörébe tartozik.*
3. *A függvényt az osztály egy példányán keresztül hívjuk meg (a függvényben használható a this mutató)*

Ha egy tagfüggvényt statikusnak (*static*) deklarálunk, akkor a fenti három tulajdonságból csak az 1. és a 2. érvényes. A statikus tagfüggvények meghívása lehetséges példányok nélkül is. Ilyenkor az osztály nevét használjuk a hatókör operátorral.

Ha egy tagfüggvényt barátnak (`friend`) deklarálunk, akkor a fenti három tulajdonságból csak az első érvényes. Tehát a függvény ilyen fajta kötődése az osztályhoz még lazább, mint a statikus függvényeké.

3.8.2. Barát osztályok

Néha a barát relációt nemcsak egy függvényre akarjuk megadni, hanem egy osztály összes tagfüggvényére. Tekintsük a láncolt listaelem (`ListElement`) és a láncolt lista (`List`) közötti kapcsolatot. Azt szeretnénk, ha a láncolt lista hozzáférne a listaelem osztály minden eleméhez. Ezt a következőképpen tehetjük:

```
class List;
class ListElement{
private:
   int key;
   ListElement * next;
   friend class List;
   ...
};

class List{
private:
   ListeElement * head;
public:
   bool find(int key);
   ...
};
```

Abban az esetben, ha csak az osztály egy függvényének akarjuk megengedni a hozzáférést, akkor ezt megfelelőképpen kell deklarálnunk.

```
class List;
class ListElement{
private:
   int key;
   ListElement * next;
   friend class List::find;
   ...
};
```

3.9. Gyakori hibák

1. A `delete` operátor használata olyan esetekben, amikor a mutató egy statikus helyfoglalású változó.

```
int i=10, *pi = &i
delete pi;
```

2. Tömbök helytelen felszbadítása. A tömb létrehozásakor lefut a konstruktor a tömb minden egyes elemére. Felszabadításkor, ha a `delete [ ]` operátort használjuk, akkor ez destruktorhívást eredményez minden egyes tömbelemre, amelyet majd követ a memória felszbadítás. Ha csak a `delete` operátort használjuk, akkor általában a memória felszbadítás nem lesz teljes.

```
Datum * td = new Datum[ 5 ];
delete td;//Helytelen
```

3. Egy pointer felszabadítás utáni értéke általában nem NULL. Hiba ezt feltételezni.

```
delete ptr;
if(! ptr) {...} //Vagy igen, vagy nem
```

4. A megfelelő zárójelek használata a new operátor esetében.

```
T * ptr = new T(3);
T * ptr = new T[ 3 ];
```

A fenti példa első sora egyetlen `T` típusú objektum létrehozását eredményezi, amelyet egy egész típusú paraméterrel hívhatunk. A második sor egy 3 elemű tömb létrehozását eredményezi. Ebben az esetben a `T` típusnak kell létezzen implicit konstruktora.

5. Egy helyfoglalásnak mindig egy felaszbadítás felel meg.

```
T *q, * p=new T[ 10 ];
for(int i=0; i<10; i++){
```

```
    q= &p[ i ];
    delete q;//Helytelen
}
```

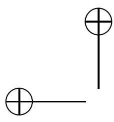
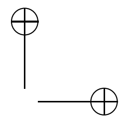
3.10. Feladatok

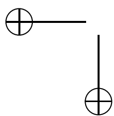
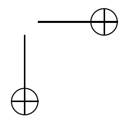
1. Implementálja a sor adatstruktúrát körpuffer segítségével. Az implementálást végezze egy rögzített primitív típusra.
2. Implementálja az egyszeresen láncolt lista adatstruktúrát kétféleképpen:
   - a listaelem típust valósítsa meg belső osztállyal
   - a listaelem típust valósítsa meg külső osztállyal, amely barátja a lista osztálynak
3. Adott a következő program:

```
#include <iostream>

int main(){
   std::cout<<"Hello"<<endl;
}
```

   A `main` függvény megváltoztatása nélkül módosítsuk a programot úgy, hogy a következő kimenetet adja:

```
Start
Hello
Stop
```

4. FEJEZET

OPERÁTOROK TÚLTERHELÉSE

A legtöbb esetben függvényeinket különböző nevekkel látjuk el. Amikor azonban a függvények lényegében ugyanazt a műveletet végzik különböző típusú objektumokon, kényelmesebb lehet ugyanúgy elnevezni azokat. Azt, hogy különböző típusokra vonatkozó műveletekre ugyanazt a nevet használjuk, túlterhelésnek (overloading) nevezzük. A túlterhelt függvénynevek kényelmi szempontokat szolgálnak. A fordítóprogramok szemszögéből nézve ezen függvények esetében csak a név közös, a szignatúra különböző, ezért a túlterhelés már fordításidőben feloldható. A fordítóprogram a túlterhelt függvények halmazából úgy választja ki a megfelelő változatot, hogy megkeresi azt a függvényt, amelyiknél a hívás paraméter-kifejezésének típusa a legjobban illeszkedik a függvény formális paramétereire. A túlterhelés feloldása természetesen független a függvények deklarációs sorrendjétől.

A gyakran használt fogalmakra, kifejezésekre a különböző szakterületek rövidítéseket használnak. Így például a matematikában a + szimbólum az összeadás jele. Ezt a + szimbólumot használjuk a programozási nyelvekben is a primitív típusú változók összeadására. Jó lenne, ha a felhasználói típusokkal végzett műveleteket is ugyanolyan természetesen fejezhetnénk ki, mint a primitív tpusúakat. Ezt teszi lehetővé az operátorok túlterhelése.

A kölönböző operátorfüggvényeket két osztály segítségével szemléltetjük. Az egyik a `Matrix` osztály a másik pedig a `Complex` osztály. A következő részben szemléltetjük ezen osztályok legszükségesebb elemeit, az adatábrázolást és a konstruktor/destruktor műveleteket. A többi műveletet fokozatosan vezetjük be.

```
class Matrix{
private:
   int rows, cols;
   double ** elements;
public:
   Matrix(int rows=3, int cols = 3);
```

```
      Matrix(const Matrix &);
     ~Matrix();
   };

   Matrix :: Matrix(int rows, int cols){
      this->rows = rows;
      this->cols = cols;
      this->elements = new double *[ this->rows ];
      for(int i=0; i<rows; i++)
         this->elements[ i ] = new double[ this->cols ];
   }

   Matrix :: Matrix(const Matrix & m){
      int i, j;
      this->rows = m.rows;
      this->cols = m.cols;
      this->elements = new double *[ this->rows ];
      for(int i=0; i<rows; i++)
         this->elements[ i ] = new double[ this->cols ];
      for(i=0; i<m.rows; i++)
         for(j=0; j<m.cols; j++)
            this->elemenst[ i ][ j ] = m.elements[ i ][ j ];
   }

   Matrix :: ~Matrix(){
      for(int i=0; i<this->rows; i++)
         delete [ ] this->elements[ i ];
      delete [ ] this->elements;
   }
```

A `Complex` osztály még egyszerűbb, itt minden függvényt `inline` függvényként adunk meg.

```
   class Complex{
      double re, im;
   public:
      Complex(double re= 0, double im=0){
         this->re = re; this->im = im; }
   };
```

4.1. Megkötések az operátor-túlterhelésre nézve

Az ismert operátorok kifejezések elemei. Az operandusok száma szerint az operátorokat osztályozhatjuk unáris, bináris és ternáris operátorokra. Szabványos C nyelvben az egyetlen három operandusú operátor a ?: operátor, az összes többi unáris és bináris. Természetesen az operátorokat jellemzi a precedencia is, amelyet egy precedencia táblázatban szokás összefoglalni([1], 159. oldal). A következőkben felsoroljuk, hogy mit nem lehet tenni az operátorok túlterhelésével.

1. Nem lehet új operátorszimbólumot bevezetni.
2. Bizonyos operátorokat nem lehet túlterhelni:
   - `sizeof`
   - `::` (hatókör)
   - `.` (tagkiválasztás)
   - `.*`(tagkiválasztás a tagra hivatkozó mutatón keresztül)
   - `?:`(feltételes operátor)
   - `typeid` (futás idejű típusinformáció)
3. Kiterjesztéssel nem változtatható meg az operátor precedenciája.
4. Nem változtatható meg az operátor operandusainak száma

4.2. Operátorfüggvények deklarációja és hívása

4.2.1. Deklaráció

Egy `X` osztályhoz tartozó operátorfüggvényeket kétféleképpen adhatunk meg, tagfüggvényként és nem tagfüggvényként, amelynek legalább egy `X` típusú paramétere van. Például:

```
class X{
   X operator+(const X&);
};

X operator+ (const X&, const X&);
```

A fenti X osztályhoz megadtuk az összeadás bináris operátort kétféleképpen. A tagfüggvényként való megadás esetében csak egy paraméterünk van, a második paraméter értelemszerűen a this. Az összeadás eredményét mindkét esetben visszatérítjük, létrehozva egy új X típusú objektumot. A paraméterek esetében azért használtunk referenciát mutató helyett, mert ez az operátorfüggvények természetesebb használatát eredményezi.

4.2.2. Hívás

```
X x1, x2;
X x3 = x1+x2;
```

Az x1+x2 kifejezést a fordító átalakítja x1.operator+(x2) hívássá tagfüggvény esetében és operator+(x1, x2) hívássá, nem tagfüggvény esetében.

Megjegyzés:

Mivel az operátorfüggvények alapvetően kétféleképpen adhatók meg, felmerül a kérdés, hogy mikor érdemes tagfüggvényként megadni. Előnyös, ha minél kevesebb függvény fér hozzá egy adott objektum belső adatábrázolásához. Ez úgy érhető el, ha csak azokat az operátorfüggvényeket adjuk meg tagfüggvényként, amelyeknek meg kell változtatniuk az objektum állapotát. Ilyen lesz például az értékadó operátor. Azokat az operátorfüggvényeket, amelyek új értéket állítanak elő, nem tagfüggvényként fogjuk megadni. Ilyen operátor például az összeadás.

4.3. Különböző típusú operátorok túlterhelése

4.3.1. Aritmetikai operátorok

Értelmezzük az összeadás műveletét a `Complex` és `Matrix` típusokra. Mivel az összeadás egy új értéket állít elő, ezért nem tagfüggvényként fogjuk megadni. Az adattagokhoz való hatékony hozzáférés elősegítése érdekében, az operátorfüggvényt barát függvénynek deklaráljuk.

```
class Complex{
   ...
   friend Complex operator+(const Complex& a,
                            const Complex& b);
};

Complex operator+(const Complex& a, const Complex & b){
   return Complex(a.re+b.re,a.im+b.im);
};
```

A Matrix osztály esetében is hasonlóképpen fogunk eljárni.

```
class Matrix{
   ...
   friend Matrix operator+(const Matrix& a,
                           const Matrix& b);
};

Matrix operator+(const Matrix& a, const Matrix& b){
   //Méretek egyezésének ellenőrzése
   Matrix c(a.rows,a.cols);
   for(int i=0; i<a.rows; i++)
      for(int j=0; j<a.cols; j++)
         c.elements[ i ][ j ] =
         a.elements[i ]+b.elements[ i ][ j ];
   return c;
}
```

4.3.2. Értékadó operátorok

Az értékadó operátor megváltoztatja az objektum állapotát, hiszen általa új értéket kap az objektum. Ennek következtében az értékadó operátort tagfüggvényként fogjuk implementálni. Az értékadó operátor lehet egyszerű (=) vagy összetett (op=). Kezdetben tekintsük csak az egyszerű értékadó operátort és azt, hogy milyen esetben hívódik meg.

```
Complex z1(1,2), z2(3,4);//Két konstruktorhívás
Complex z3 = z1; //Másoló konstruktor
z1=z2; // értékadó operátor
z1 = z2+z3;//+ operátor majd értékadó operátor
```

A fenti példa első sora két komplex szám definícióját tartalmazza, amelyek egy-egy konstruktorhívást jelentenek. A második sorban egy újabb komplex szám definiciója következik, amely esetében a másoló konstruktor fog meghívódni. Ha a programozó nem definiált másoló konstruktort, akkor a fordító generált egyet, amely bitenként lemásolja az objektumot, ezzel inicializálva az újonnan létrehozottat. A harmadik és negyedik sorokban viszont nincs szó új objektum létrehozásáról, egyszerűen értékadást kell végezni két létező objektum között. Ilyen esetekben, mindig az értékadó operátor hívódik meg. Ha a programozó nem deklarál értékadó operátort, akkor a fordító generál egyet. A fordító által generált konstruktor egyszerűen lemásolja a jobboldali objektum mezőit, majd visszatérít egy referenciát a baloldali objektumra. Osztály típusú adattagok esetében, ha az illető osztálynak van másoló operátora, akkor ez meg fog hívódni az adott mező másolása esetében.

A `Complex` típus esetében a fordító által generált másoló konstruktor tökéletesen megfelelő. Ha viszont a `Matrix` osztályt tekintjük, akkor láthatjuk, hogy a bitenkénti másolás nem lesz megfelelő, hiszen így az `elements`, mutató típusú mező helytelenül inicializálódik, az értákadás bal oldalán levő objektum elements mezője, illetve a jobb oldalon levő objektum `elements` mezője is ugyanarra a címre fog mutatni. Ilyen esetekben, kötelező értékadó operátort definiálni.

Az értékadó operátor általános alakja, egy `X` osztály esetén, a következő:

```
X& operator=(const X&)
```

A `Matrix` osztály esetében ez a következőképpen nézne ki:

```
Matrix& Matrix::operator=(const Matrix & m){
   if(this != &m){
      int i, j;
      if(this->rows != m.rows || this->cols != m.cols ){
          //Memória felszabadítás
          for(i=0; i<this->rows; i++)
             delete[] elements[ i ];
          delete [] elements;
          //Inicializálás+helyfoglalás
         this->rows = m.rows;
         this->cols = m.cols;
         this->elements = new double *[ this->rows ];
         for(int i=0; i<rows; i++)
            this->elements[ i ] = new double[ this->cols ];
      }
       //Tartalmi másolás
      for(i=0; i<m.rows; i++)
         for(j=0; j<m.cols; j++)
            this->elements[ i ][ j ] = m.elements[ i ][ j ];
   }
   return *this;
}
```

A fenti értékadó operátor blokkja egy feltétellel kezdődik, `this != &m`, azaz az értékadás baloldalán levő objektum különbözik a jobb oldalon levő objektumtól. Ez értelemszerűen kivédi a következő alakú hívások esetében a végrehajtódást:

```
Matrix A(3,3);
A = A;
```

Ez után következik a két mátrix méreteinek ellenőrzése. Abban az esetben, ha ezek nem egyeznek, memória felszabadítást illetve új helyfoglalást kell végezni az értékadás bal oldalán levő mátrixnak. Ez természetesen egy időigényes művelet, ezért elképzelhető lenne egy olyan implementáció is, amelyben csak az azonos méretű mátrixok esetében engedélyezzük az értékadást. Ebben az esetben

csak a tartalmi másolást kell elvégezni. A fenti implementációban általánosságra törekedtünk, ezért választottuk ezt az implementációt. A tartalmi másolás után visszatérítjük az aktuális objektumot. Erre azért van szükség, hogy az értékadás műveletét láncoltan is lehessen használni. Például:

```
Matrix A, B, C;
...
C = B = A;
```

A fenti példában az értékadás a megszokott módon, jobbról balra haladva, hajtódik végre. Először a B mátrix veszi fel az A mátrixot, majd ezen értékadás eredményét felveszi a C mátrix. A műveletsor végén mindenik mátrix az A-val lesz egyenlő.

Most pedig tekintsünk egy példát az összetett értékadó operátorra.

```
inline Complex& Complex :: operator+=(const Complex& z ){
   re += z.re; im += z.im;
   return *this;
}
```

Ha összehasonlítjuk a fenti += operátort a Complex osztály egyszerű összeadás operátorával, akkor azt láthatjuk, hogy a += operátor sokkal hatékonyabb, hiszen nem hoz létre új objektumot, az összeadás eredménye a hívó objektumban fog tárolódni.

```
Complex z1(1,2), z2(2,3);
z1+=z2;
```

A fenti kódrészletben a `z1+=z2` hívás `z1.operator+=( z2)` hívássá alakul és az összeadás eredménye a `z1` objektumban lesz.

4.3.3. Osztályok kanonikus alakja

A felhasználói típusokat jó úgy definiálni, hogy a típust függvényargumentumként is használni lehessen, valamint biztonságosan végezhetőek legyenek a

következő műveletek: adott típusú változók deklarációja, értékadás. Ezért az osztálynak tartalmaznia kell:

- implicit konstruktort: `X::X()`
- másoló konstruktort: `X :: X(const X&)`
- értékadó operátort: `X& X :: operator=(const X&)`
- destruktort: `X::~X()`

4.3.4. Index operátor

Ez az operátor a tároló típusú osztályok jellemzője. Olyan bináris operátor, amelynek egyik operandusa a tároló, a másik pedig egy tárolón belüli pozíció. Olyan tárolók esetében szokás ezt a műveletet megadni, amelyeknek biztosítaniuk kell a közvetlen(direkt) elérést. Ennek következtében egy dinamikus tömbhöz vagy egy mátrix osztályhoz mindenképpen szükséges ilyen műveletet megadni. A verem esetében egyáltalán nincs szükség ilyen műveletre, hiszen ez egy szekvenciális hozzáférésű adatszerkezet. A láncolt lista is egy szekvenciális hozzáférésű adatszerkezet, ez esetben sem indokolt a művelet megadása. Ennek ellenére bizonyos láncolt lista megvalósítások megadják ezt a műveletet.

Egy tömb elemét használhatjuk értékadás bal illetve jobb oldalán egyaránt. Ezért az operátornak kötelező balértéket visszatéríteni. Az index operátor általános alakja egy `X` osztály esetében a következő lesz:

```
X& X :: operator [](int index);
```

Nézzük meg, hogyan is implementálhatnánk ezeket a műveleteket egy egyszerű egydimenziós tömb osztály és a mátrix osztály esetében.

```
template <class T>
class Vector{
   T * elements;
   int size;
```

```
public:
   Vector(int _size=100): size(_size){
      elements = new T[size];
   }

   ~Vector(){
      delete[] elements;
   }

   ...

   int getSize() const { return size;}
   T& operator[](int index){ return elements[ index ]; }
   T  operator[](int index) const { return elements[ index ]; }
};
```

A második index operátor megadása azért szükséges, hogy konstans példányokat is létrehozhassunk. Például: `const Vector<int> v;` Ebben az esetben nem szükséges referenciát visszatéríteni, hiszen az indexkifejezés nem szerepelhet értékadás bal oldalán (`const` módosítójú függvény!)

Általában az index operátort úgy implementálják, hogy nem végez indexellenőrzést, aminek következtében nagyon gyors. Szokás megadni egy függvényt is az operátor mellett, amely szintén hozzáférést biztosít a tároló elemeihez, viszont itt már a biztonság az elsőrendű kritérium, ezért a függvény hibakezelést is végez.

Használat:

```
int i;
Vector v(10);
for(i=0; i<v.getSize(); i++) v[ i ] = i;
for(i=0; i<v.getSize(); i++) cout<<v[i]<<"\t";
```

A mátrix osztály esetében az index operátort a következőképpen adhatjuk meg:

```
class Matrix{
   int rows;
   int cols;
   double ** elements;
public:
```

```
   ...
   double * operator[](int index){ return elements[ i ]; }
   ...
};
```

Ebben az esetben a matrix osztály biztosítja a sorok indexelését, visszatérítve egy double * mutatót, amelyre már automatikusan alkalmazható az alapértelmezett index operátor.

4.3.5. Növelés és csökkentés

Ezeket az operátorokat úgy előtagként, mint utótagként is alkalmazhatjuk. Természetesen csak akkor definiáljuk, ha valamilyen értelmes műveletet jelent az adott típusra. Tekintsük például a már ismertetett dátum típust. A dátum növelése és csökkentése értelmes műveleteknek tűnnek.

Ha előtagként akarjuk használni az operátort, akkor ezt egy X osztály esetében a következőképpen kell megadni:

```
X operator++();
```

utótag esetében pedig

```
X operator++(int a);.
```

Utótagként való használat esetében a paraméter átadása automatikusan történik. Ennek egyetlen szerepe az előtag és utótag operátorok megkülönböztetése. Az előtag operátor implementálása hatékonyabb, hiszen a megnövelt objektumot kell visszatéríteni. Az utótag operátor esetében először le kell menteni az aktuális objektumot, megnövelni, majd a lementett objektumot visszatéríteni.

```
Datum& Datum :: operator++(){
   // az aktuális dátum objektum növelése
   return *this;
}

Datum Datum :: operator++(int a){
```

```
    Datum d = *this;
    // az aktuális dátum objektum növelése
    return d;
}
```

A két operátorfüggvény a visszatérített típusban is különbözik, hiszen amíg az előtag operátor esetében ez egy referencia, az utótag operátor esetében egy objektum.

```
Datum d1,d2;
d2 = ++d1;
d2 = d1++;
```

Az első értékadás eredményeképpen megnövekszik a `d1`, és ezt a növelt értéket felveszi a `d2`. Így az értékadás után mindkét objektum állapota azonos lesz. A második értékadás eredményeképpen a `d2` először felveszi a `d1` aktuális értékét, majd a `d1` megváltozik.

Az utótagként való implementáció esetében hiba lenne referenciát téríteni vissza, mert ez egy lokális, vermen tárolt objektum-referencia lenne, amely a függvényhívás után rögtön megszűnik. Ha viszont objektumot térítünk vissza, akkor mindig másolat készül a vermen levő objektumról és az térítődik vissza.

4.3.6. Inserter/Extractor operátorok

Úgy az inserter, mint az extractor operátor egy-egy bináris operátor lesz, amelynek első paramétere egy kimeneti, illetve egy bemeneti adatfolyam. A második paraméter a tulajdonképpeni tartalom, amit ki szeretnénk küldeni az adatfolyamba, illetve amibe be szeretnénk olvasni az adatfolyamból. Mivel ezeket az operátorokat láncoltan is szeretnénk használni, ezért kötelező módon vissza kell téríteniük egy referenciát az adatfolyamra.

A `Complex` osztály esetén ez a két operátor így nézne ki:

```
class Complex{
    ...
    friend ostream& operator<<(ostream& os, const Complex& z);
```

```
    friend istream& operator >>(istream& is, Complex& z);
};

ostream& operator<<(ostream& os, const Complex& z){
    os<<z.re<<"+"<<z.im<<" i";
    return os;
}


istream& operator>>(istream& is, Complex& z){
    is>>z.re>>z.im;
    return is;
}
```

Használat:

```
Complex z1(1,2), z2(3,4);
cout<<z1<<endl<<z2<<endl;
cin>>z1;
cout<<z1<<endl;
```

4.3.7. Konverziós operátorok

Tekintsük az `X` osztályt és egy adott `T` típust. Kétféle átalakításról beszélhetünk:

a.) `T` típus átalakítása `X` típusúvá

b.) `X` típus átalakítása `T` típusúvá

Az a.) típusú átalakítást olyan konstruktorral valósíthatjuk meg, amelynek egy `T` típusú paramétere van `X :: X(const T&)`. A `Complex` osztály esetében ez a következőképpen nézne ki:

```
Complex::Complex(double re){ this->re = re; this->im = 0}

Complex z = 2;
```

Természetesen az implicit értékű paraméterekkel rendelkező konstruktorunk is ugyanezt teszi, amennyiben egy paraméterrel hívjuk. Meg kell jegyeznünk, hogy

ha a konstruktornak minden paramétere implicit értékű, akkor a fent megadott újabb konstruktort nem is engedélyezi a fordító.

A b.) típusú átalakításra be kell vezetnünk egy konverziós operátorfüggvényt. Az ilyen operátorfüggvények alakja: `X :: operator T()`. A konverziós operátorfüggvények esetében nem kell megadni a visszatérítési típust, ez értelemszerűen `T` lesz.

A Complex osztály esetében megadhatjuk a valóssá való átalakítást.

```
Complex :: operator double(){ return re; }
```

Ezek után a `Complex` típusú változók nyugodtan szerepelhetnek valós kifejezésekben, mert átalakíthatóak valós számokká. Például helyesnek minősül a következő kifejezés:

```
Complex z (3,5);
double d = 10+z*2;
```

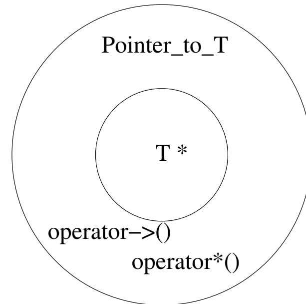
4.3.8. Explicit konstruktor

Az egy paraméterrel hívható konstruktorok mindig átalakítást végeznek a paraméter típusa és az adott osztály típus között. Bizonyos esetekben kívánatos ez a viselkedés, bizonyos más esetekben pedig nem kívánatos. Tekintsük a következő két objektumdeklarációt:

```
Complex z = 2;
Stack s = 2;
```

Mivel mindkét osztálynak van egyparaméteres konstruktora, ezek a deklarációk konstruktorhívásokká alakulnak implicit módon. Az első egy `Complex(2)`, míg a második egy `Stack(2)` hívássá fog alakulni. Amíg az első esetben értelmesnek tűnik ez az átalakítás, addig a második esetben értelmetlen egész számot veremmé alakítani. Ha ezt az implicit átalakítást el szeretnénk kerülni, érdemes az egyparaméteres konstruktorokat `explicit` kulcsszóval deklarálni.

```
class Stack{
   ...
```



4.1. ábra. Burkolt mutató

```
   explicit Stack(int n){...}
};
```

Ha az egyparaméteres konstruktorunk explicit, akkor a `Complex z=2` és a `Stack s=2` fordítási hibának minősül. A továbbiakban csak `Complex z(2)` és `Stack s(2)` deklarációk lesznek engedélyezettek.

4.3.9. Közvetlen (direkt) hozzáférés (*) és közvetett (indirekt) hozzáférés (->)

A * és -> operátorok túlterhelésének fő alkalmazása az intelligens mutató (smart pointer) típusok létrehozása. Az intelligens mutató olyan objektum, amely mutatóként viselkedik. Tekintsük a következő intelligens pointer osztályt, amely gyakorlatilag egy mutatónak képez burkolatot. Ezt szemlélteti a 4.1. ábra.

```
class Pointer_to_T{
   T* p;
public:
   explicit Pointer_to_T(T * p){ this->p=p; }
   T* operator->(){ return p; }
   T& operator*(){ return *p; }
};
```

Használat:

```
struct T{
   int a;
   double b
   T(int a, double b){ this->a = a; this->b = b; }
};

T * a = new T(2, 5.5);
Pointer_to_T ptr(a);
cout<<ptr->a<<endl<<ptr->b<<endl;
(*ptr).a =7;
```

Intelligens (okos) mutatókról a következő helyeken olvashatunk: ([3] 172. oldal, [6] 365. oldal).

4.3.10. Függvényhívás operátor: ()

A `függvénynév(aktuális_paraméter_lista)` úgy tekinthető, mint a `()` bináris operátor, amelynek bal operandusa a függvénynév, a jobb operandusa pedig az aktuális paraméterlista. A függvényhívás operátort olyan osztályok esetében terheljük túl, amelyek példányai függvényként viselkednek. Egy ilyen osztálynak általában csak egy művelete van és a függvénymutató általánosításának tekinthető, hiszen nem egy függvényt fogunk paraméterként átadni, hanem egy olyan objektumot, amely függvényként viselkedik. Az egyetlen előnyük a függvénymutatókkal szemben, hogy objektum mivoltukból kifolyólag képesek adatok tárolására is, ezért a művelet végrehajtása során megváltoztathatják állapotukat. Szemléltetésre tekintsük a következő példát, amelyben a függvényhívás operátor szerepe megnövelni a paramétere értékét egy előre beállított értékkel.

```
class AddValue{
   int value;
public:
   AddValue(int _value){ this->value = _value;}
   void operator()(int& element){ element += value;}
};

int main(){
   int i;
```

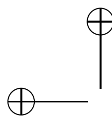
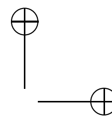
```
    int x[] ={ 1, 2, 3, 4, 5};
    AddValue ob(10);
    for(i=0; i<5; i++)
       ob(x[ i ]);
    for(i=0; i<5; i++)
       cout<<x[ i ] <<"\t";
    cout<<endl;
    return 0;
}
```

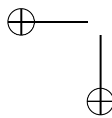
Ebben a példában az objektum csak a növekmény értékét tárolta. Módosíthatjuk osztályunkat úgy, hogy számlálja például azt, hogy hány elemet módosít a növekmény értékével. Ennek elérésére bevezetünk egy számláló (`counter`) adattagot és ennek lekérdezésére egy `getCounter()` műveletet.

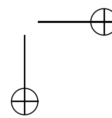
```
class AddValue{
   int value;
   int counter;
public:
   AddValue(int _value){ this->value = _value; counter= 0;}
   void operator()(int& element){ element += value;
                                  counter++;}
   int getCounter() const { return counter; }
};
```

4.4. Feladatok

1. Készítsen egy Matrix osztályt kétdimenziós tömbök ábrázolására. Készítsen operátorfüggvényeket, amelyek a szabványos mátrixműveleteket biztosítják a lehető legkényelmesebb módon.
2. Készítsen egy dinamikus tömb osztályt. Használja az operátorok felülírását a tömbelemek elérésésre. Az osztály legyen kanonikus alakban.
3. Készítsen egy bejáró osztályt a Matrix osztályhoz, amely a beállítás függvényében ennek sor, illetve oszlopfolytonos bejárását teszi lehetővé.





5. FEJEZET

SZÁRMAZTATOTT OSZTÁLYOK

5.1. Bevezetés

Objektumorientált programozásban az osztályok között háromféle társítási kapcsolatot értelmeztünk:

- ismeretségi kapcsolat
- tartalmazási kapcsolat
- származtatási kapcsolat (öröklődés)

A kapcsolatok szemléltetésére nézzük meg, hogy a következő fogalmak hogyan viszonyulnak egymáshoz.

1. síkidom, kör, háromszög, négyzet, téglalap
2. autó, kerék, kormány, motor, váz
3. termelő, fogyasztó, termékraktár (klasszikus termelő-fogyasztó probléma)

Származtatási kapcsolat

Az első csoportból rögtön kiemelhető a síkidom, amely egy általános fogalom, a kör, háromszög, négyzet, téglalap pedig sajátos síkidomok. Természetesen a téglalap és a négyzet között is van egy ehhez hasonló viszony, itt viszont két konkrét fogalom között van a viszony, a téglalap általánosabb, mint a négyzet.

Ezeket a fogalmakat, ha osztályokkal implementálnánk, akkor származtatási viszonyt használnánk a a síkidom osztály és a konkrét síkidomok (téglalap, kör, négyzet) között.

Tartalmazási kapcsolat

A második csoportban levő elemek tartalmazási, rész-egész kapcsolatban vannak egymással. Az autó az egész, amelynek részei a kerék, a kormány, a motor meg a váz is. A részek együttesen alkotják az egészet. A tartalmazási kapcsolat esetében megkülönböztetünk laza és szoros tartalmazási kapcsolatot. A laza tartalmazás esetében a rész túlélheti az egészet, szoros tartalmazás esetében viszont nem. Az autókerék túlélheti az autót, az emberi agy viszont nem éli túl az embert (biológiai értelemben).

Ismeretségi kapcsolat

Az ismeretségi kapcsolatot a klasszikus termelő-fogyasztó problémával szemléltetjük. Tételezzük fel, hogy van egy adott kapacitású raktárhelyiség, egy termelő és egy fogyasztó vállalat. A termelő a termelt javakat a raktárban helyezi el, ha van szabad hely, a fogyasztó pedig innen fogyaszthat, amennyiben van termék. Úgy a termelőnek, mint a fogyasztónak hozzá kell férnie a raktárhoz, amely sem a termelőhöz sem pedig a fogyasztóhoz nem tartozik. Sőt, a raktár önállóan is létező fogalom. Ilyen esetben ismeretségi kapcsolatról beszélünk.

Kapcsolatok implementálása

Az *ismeretségi kapcsolatatot* általában mutató vagy referencia segítségével valósítjuk meg. Ez a kapcsolat amolyan laza kapcsolatot fejez ki a két osztály között. A példányok egymástól függetlenül is léteznek és a futásidő bizonyos idejére kapcsolatba kerülnek egymással, bizonyos tevékenység elvégzése céljából. Tekinstük az `A` és `B` osztályokat, ahol az `A` osztály ismeri a `B` osztályt. Az ismeretségi kapcsolat megvalósítására mutatót fogunk használni.

```
class B{
```

```
};

class A{
   B * pb;
public:
   A(...);
   ...
};
```

A szoros tartalmazási kapcsolatot általában úgy implementáljuk, hogy a tartalmazó osztálynak lesz egy tartalmazott típusú adattagja, amely a tartamazóval egyszerre jön létre és ezzel együtt hal el.

```
class B{...};
class A{
   B b;
};
```

A laza tartalmazási kapcsolatot az ismeretségi kapcsolathoz hasonlóan mutatóval vagy referenciával valósítjuk meg.

5.2. Származtatás

A származtatás az egyetlen olyan osztályok közötti viszony, amelyet minden objektumorientált nyelv támogat, vagyis nyelvi szinten egyértelműen kifejezhető. A C++ nyelv kétféle öröklést támogat, nyilvános és privát öröklést. Ezen felül egy osztálynak akárhány ősosztálya lehet, ezt pedig többszörös öröklésnek nevezzük. A nyilvános öröklés a gyakrabban használt származtatási kapcsolat, így ennek bemutatásával kezdjük.

```
class Sikidom{ ... };
class Kor : public Sikidom{ ... };
```

Látható, hogy a származtatott osztályt a következő szintaxis szerint deklaráltuk:

```
class származtatott_osztály : <láthatósági módosító> ősosztály{ .. };
```

A privát örökítés szintaxisa is hasonló, csak a láthatósági módosító `private` lesz. Ezt a fajta származtatást később ismertetjük.

A származtatott osztály örökli az ősosztály adatait és metódusait. Ennek következtében minden adatmező, amelyet egy ősosztálybeli példány tartalmaz, jelen lesz az utódosztálybeli példányban is. Az utódosztály örökli az ősosztály metódusait is. Mivel egy osztály metódusai (műveletei) meghatározzák az adott osztálybeli példányok viselkedését, ezért az utódosztály örökli az ősoszály által meghatározott viselkedésmódot. Az utódosztály tartalmazni fogja az ősosztály műveleteit.

Egy származtatott osztály a következőket teheti az ősosztállyal:

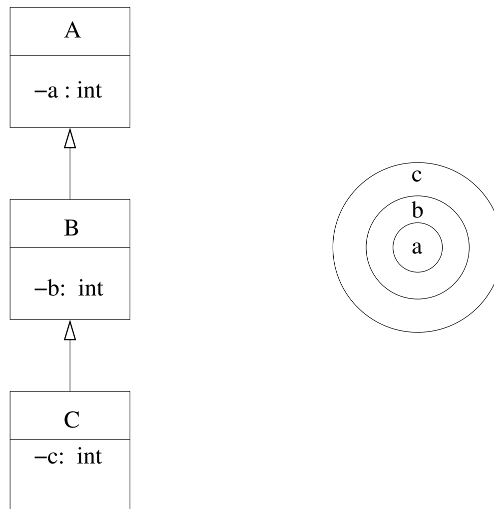
- új adatokkal bővíti az ősosztályt
- új metódusokkal bővíti az ősosztályt
- felülírja az ősosztály adatait
- felülírja az ősosztály metódusait

5.3. A származtatott osztály példányának létrehozása

A származtatott osztálybeli objektumok úgy épülnek fel, hogy sorra, az ősosztálytól kezdődően meghívódnak a konstruktorok, minden konstruktor felépítve a neki megfelelő réteget.

Tekintsük példaként a 5.1. ábrán látható háromszintes osztályhierarchiát. Minden osztály egyetlen adattagot deklarál. Az `A` osztály konstruktorának feladata inicializálni az `a` adattagot, a `B` osztály konstruktora a `b` adattagot inicializálja, a `C` osztályé pedig a `c` adattagot. Egy `C` típusú objektum létrehozásához három konstruktorhívásra van szükségünk. Az objektum három rétegből épül fel a következő sorrendben:

- Az `A` osztály konstruktora inicializálja a legbelső réteget



5.1. ábra. Példány réteges felépítése

- A `B` osztály konstruktora inicializálja a középső réteget
- A `C` osztály konstruktora inicializálja a legkülső réteget

Amíg a konstruktorok belülről kifele építik fel az objektumot, addig a destruktorok fordított sorrendben hívódnak meg. Először a legkülső réteg destruktora, majd kívülről befele haladva egészen a legbelső réteg destruktoráig.

Tekintsük a következő kódrészlet, amelyben mindhárom osztálynak van alapértelmezett konstruktora.

```
class A{ int a; };
class B : public A { int b; };
class C: public B { int c; };
int main(){
   C c;
   return 0;
}
```

Ez a következő metódushívásokat eredményezi:

A()-->B()-->C()-->~C()-->~B()-->~A()

Ha az ősosztálynak nincs alapértelmezett konstruktora, akkor a származtatott osztály konstruktorában kötelező meghívni az ősosztály konstruktorát. Az ennek átadandó paraméterekről szintén a származtatott osztálynak kell gondoskodnia. A következő kódrészlet ezt szemlélteti:

```
#include <iostream>
using namespace std;
class A{
protected:
   int a;
public:
   A(int pa) : a(pa) { cout<<"A konstruktor:"<<a<<endl; }
   ~A(){ cout<<"A destruktor"<<endl; }
};

class B: public A{
protected:
   int b;
public:
   B(int pa, int pb) : A(pa), b(pb) {
      cout<<"B konstruktor:"<<a<<": "<<b<<endl;
   }
   ~B(){ cout<<"B destruktor"<<endl; }
};

class C: public B{
protected:
   int c;
public:
   C(int pa, int pb, int pc) : B(pa, pb), c(pc) {
      cout<<"C konstruktor:"<<a<<": "<<b<<": "<<c<<endl;
   }
   ~C(){ cout<<"C destruktor"<<endl; }
};
int main(){
   C c(1, 2, 3);
   return 0;
}
```

A fenti kódrészletben egyik osztálynak sincs alapértelmezett konstruktora, ezért a származtatott osztályok konstruktorait úgy paramétereztük, hogy gondoskodhassanak az ősosztály adattagjainak inicializálásáról is. Ha a származtatott osztályokban is hozzá akarunk férni az ősosztály adattagjaihoz, akkor ezeket a védett (`protected`) láthatósági módosítóval kell megadni.

A származtatott osztály konstruktorát a következő szintaxis szerint kell megadni:

```
    származtatott_osztály_konstruktor : ősosztály1(argumentumlista),
ősosztály2(argumentumlista)...{}
```

A származtatott osztály konstruktora és az elsőként megadott ősosztály konstruktora között a ':' szimbólum áll. Amennyiben több közvetlen ősosztály van, ezeket a ',' szimbólummal választjuk el egymástól. A fenti példában a `B` osztálynak egyetlen ősosztálya van, a konstruktorát mégis úgy adtuk meg, hogy `B(int pa, int pb): A(pa),b(pb){...}`. A ':' szimbólum utáni listát taginicializáló listának is nevezzük és kötelező módon a következőket kell tartalmaznia:

1. ősosztályok inicializálása (amelyeknek nincs alapértelmezett konstruktora)
2. felhasználói osztály típusú adattagok inicializálása (amelyeknek nincs alapértelmezett konstruktora)
3. referencia típusú adattagok inicializálása
4. konstans adattagok inicializálása

Nem kötelező, de ugyancsak itt végezhető a primitív típusú adattagok inicializálása, a többihez hasonlóan `adattag_név (inicializáló kifejezés)`. Minden adattag inicializálását lehetőleg végezzük taginicializáló lista segítségével. Az előnyök szemléltetése végett tekintsük a következő példát [11]:

```
class A{
   string s1, s2;
public:
```

```
   A(){ s1="Hello, "; s2="világ"; }
};
```

Egy `A` típusú példány létrehozásakor valójában a konstruktor úgy fut le, hogy először minden olyan adattagot, amelynek kezdőértékéről nem rendelkeztünk a taginicializáló listában, azt az alapértelmezett konstruktor segítségével építi fel a fordító, utána pedig lefut a konstruktor blokkja, elvégezve az itt megadott értékadásokat is. Tehát a fenti példa esetében a következő hívásokat eredményezi:

```
A() : s1(), s2() {s1="Hello, "; s2="világ"; }
```

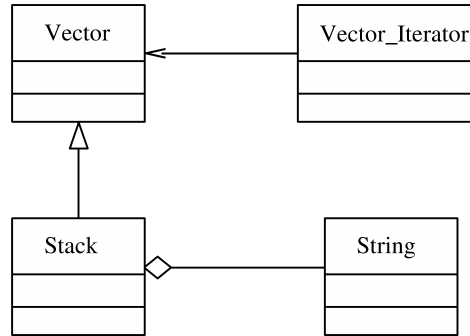
- `s1() - string` osztály alapértelmezett konstruktora
- `s2() - string` osztály alapértelmezett konstruktora
- `s1="Hello, "; - string` osztály értékadó operátor
- `s2="világ"; - string` osztály értékadó operátor

Ha viszont a konstruktor blokkjában szereplő értékadás helyett, a taginicializáló listát választjuk, ez tömörebb és egyben gyorsabb kódot is eredményez, kikerülve a fölösleges értékadó operátorfüggvények hívását. Az `A` osztály konstruktorát tehát adjuk meg így:

```
A() : s1("Hello, "), s2("világ") {}
```

A fentiek szemléltetésére tekintsük a következő konkrét osztályokat, amelyekben a `Stack` osztály a `Vector` utódja, a `Stack` tartalmazási kapcsolatban van a `String` osztállyal. Ezen kívül a `Stack` osztálynak van egy beépített típusú adattagja (`_top`), egy konstans adattagja (`_MAX_SIZE`). A `Vector` osztályhoz készített `Vector_Iterator` osztály ismeretségi kapcsolatban van a bejárandó `Vector` tárolóval, ezt a kapcslatot egy referenciával valósítottuk meg. A 5.2 ábra ezen osztályok közötti viszonyokat szemlélteti.

```
class Vector{ public: Vector(int len);...};
class String{ public: String(char *);...};
class Stack: private Vector{
String _name; //felhasználói típus
   const int _MAX_SIZE; //konstans
```



5.2. ábra. Osztálydiagramm

```
   int _top;
public:
   Stack(int len,char * name) : Vector(len), _name(name),
      _MAX_SIZE(len), _top(0){}
//...
};

class Vector_Iterator{
   Vector& _vec; //referencia
   int _act;
public:
   Vector_Iterator(const Vector& v): _vec(v), _act(0){}
//...
};
```

5.4. Adattagok felülírása

Egy származtatott osztályban felülírhatjuk az ősosztálybeli adatokat. Amennyiben valamit felülírunk, az felülbírálja a felülírtat és az utódosztálybeli felülírásnak megfelelően fog viselkedni. A helyzet valamelyest a globális-lokális változókhoz hasonlít, a lokális mindig felülbírálja a globálist. Ha például egy adattagot írunk felül, akkor ez azt jelenti, hogy két azonos nevű adattagunk van, az egyik az ősosztály hatókörébe tartozik, a másik pedig a származtatott osztály hatókö-

rébe. Ha nem minősítjük az adattagot, akkor az az utódosztálybeli deklarációnak megfelelő lesz. Az ősosztálybeli azonos nevű adattagot az `Ősosztály::adattag` minősítéssel érjük el.

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

class A{
protected:
   int a;
public:
   A(int a = 10){ this->a = a; }
};

class B : public A{
   int a;
public:
   B(int a = 10): A(a-10) { this->a = a;}
   void f(){
      cout<<"1. a="<<a<<endl;
      cout<<"2. a="<<A::a<<endl;
   }
};
main(){
   B b;
   b.f();
   return 0;
}
```

A fenti program eredménye:
1. a=10
2. a=0

5.5. Metódusok felülírása - Polimorfizmus

A származtatott osztály felülírhatja (felülbírálhatja) az ősosztálybeli metódust. Felülírásról csak akkor beszélünk, ha a metódusok ugyanolyan paraméterlistával rendelkeznek. Tekintsük példaként az `Alkalmazott` és a `Manager` osztályokat. Mivel a `Manager` is egy sajátos alkalmazott, ezért a két osztály közötti

viszonyt származtatással fogjuk megvalósítani. Tételezzük fel, hogy mindkét osztálynak van egy-egy `print()` nevű metódusa, amely az objektum állapotát a szabványos kimenetre írja. Ha a `print()` metódustól polimorfikus viselkedést várunk el, akkor ezt a `virtual` kulcsszóval kell deklarálni. Amíg a Java nyelvben a függvények alapértelmezetten polimorfikus viselkedésűek, addig a C++ nyelv esetén ezt expliciten be kell kapcsolni a `virtual` kulcsszóval. Az ilyen metódusok esetében futásidőben dől el pontosan a hívott metódus címe, ezért ezt dinamikus kötésnek nevezzük. A nem virtuális metódusok esetében már fordításidőben eldönthető a hívott metódus címe, ezért ezt a fajta kötést statikus kötésnek nevezzük.

```
class Alkalmazott{
public:
   virtual void print();
   ...
};

class Manager: public Alkalmazott{
public:
   virtual void print();
   ...
};
```

A C++ nyelvben az objektum elérhető mutatón, referencián vagy egyszerű objektum típusú változón keresztül. Ezek közül csak referencián és mutatón keresztül valósul meg a polimorfizmus. Tekintsük a következő példát, amelyben egy alaposztálybeli mutatón keresztül először egy alaposztálybeli objektumot érünk el, utána pedig egy utódosztálybelit. A `print()` metódus hívásánál polimorfikus viselkedés fog bekövetkezni, mindig az aktuális, mutatott objektumnak megfelelő `print()` metódus kerül meghívásra.

```
Alkalmazott * pa;
Alkalmazott a;
Manager m;
pa = &a;
pa->print(); //Alkalmazott::print()
pa = &m;
pa->print(); //Manager::print()
```

A második példában használjunk alaposztálybeli referenciát:

```
void f(Alkalmazott& ra){
   ra.print();
   ...
}
```

A függvényt rendre az `a` és `m` objektumokkal hívjuk.

`f(a), Alkalmazott::print()` hívása
`f(m), Manager::print()` hívása

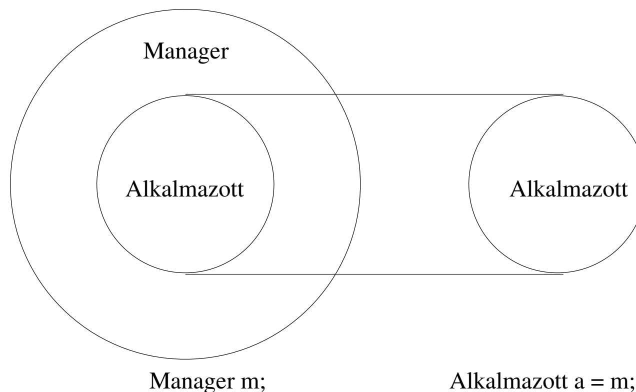
A polimorfizmus egyik leghasznosabb vonása, hogy lehetővé teszi közös ősosztállyal rendelkező objektumok ugyanazon tárolóba való behelyezését. Tekintsük a következő példát:

```
#define SIZE 3
Alkalmazott * t[ SIZE ];
t[ 0 ] = new Alkalmazott();
t[ 1 ] = new Manager();
t[ 2 ] = new Alkalmazott();
for(int i=0; i<SIZE; i++)
   t[ i ]->print();
```

A fenti kódrészlet létrehoz egy `Alkalmazott` típusú mutatókból álló tömböt, feltölti `Alkalmazott` és `Manager` típusú objektumokkal, majd bejárva a tárolót kiíratja ezeket az objektumokat. Mivel a `print()` függvény egy virtuális függvény, ezért a `t[i]->print()` hívás először az `Alkalmazott::print()` majd a `Manager::print()`, aztán ismét az `Alkalmazott::print()` hívását fogja jelenteni.

5.6. Származtatott osztálybeli objektumok másolása

Egy adott osztálybeli objektumok másolását a másoló konstruktor és az értékadó operátor határozza meg. Ha az osztály nem definiálja ezt a két metódust,



5.3. ábra. Másolás

akkor a fordító generál egy-egy ilyen metódust, amely alapértemezett módon működik, bitenkénti másolást végezve.

Nézzük meg, hogyan végezhető a másolás egy alaposztálybeli és egy utódosztálybeli objektum között, illetve ez milyen nemkívánt mellékhatásokkal járhat.

```
1. Manager m;
2. Alkalmazott a1 = m;
3. Alkalmazott a2;
4. a2 = m;
```

A 2. sorban az Alkalmazott osztály másoló konstruktora fog meghívódni és átmásolja az `m` objektum alaposztálybeli adattagjait az `a1` objektumba. Ez a másolás lehetséges, mert egy utódosztálybeli példány mindig tartalmazza az összes alaposztálybeli adattagot is. A 4. sorban az Alkalmazott osztály értékadó operátor hívódik meg. Ez is átmásolja az alaposztálybeli adattagokat az `a2` objektumba. Ez a fajta másolás a célobjektum típusának megfelelő szelet átmásolását jelenti, ahogy az 5.3. ábra is szemlélteti és automatikusan felszeletelődéssel jár. Ha nem ezt a fajta viselkedésmód a kívánatos, akkor mutatók használata ajánlott.

5.7. Absztrakt osztályok

Az előző részben az `Alkalmazott` és a `Manager` osztályok konkrét fogalmakat ábrázoltak, ezért ezeket konkrét osztályokkal valósítottuk meg. Vannak estek, amikor absztrakt fogalmakat kell ábrázolnunk, ezeket absztrakt osztályokkal valosíthatjuk meg. Az absztrakt osztály a következőket tartalmazhatja:

- konkrét adattagok
- konkrét műveletek
- absztrakt műveletek (legalább egyet kötelező módon)

Egy absztrakt osztályt nem lehet példányosítani. Attól lesz absztrakt az osztály, ha van neki legalább egy absztrakt művelete (függvénye). Az absztrakt függvények mindig virtuálisak is, hiszen az egyetlen céljuk, hogy kényszerítsék az utódosztályokat ezek implementálására. A C++ nyelvben ezeket a műveleteket tiszta virtuális függvényeknek nevezzük.

Ajánlott az interfész típusú absztrakt osztályok használata, vagyis olyan osztályoké, amelyek tiszta viselkedésmódot határoznak meg azáltal, hogy csak absztrakt műveleteket tartalmaznak.

Konkrét példának tekintsük a síkidom és a kör fogalmkat. Amíg a kör az egy konkrét fogalom, addig a síkidom egy elvont fogalom. Ennek következtében a síkidom osztályt absztrakt osztályként deklaráljuk.

```
class Sikidom{
public:
   virtual void kirajzol()=0;// tiszta virtuális függvény
};

class Kor : public Sikidom{
public:
   virtual void kirajzol(){ ... }
};
```

5.8. Virtuális destruktor

A származtatott osztályok példányait nagyon gyakran alaposztálybeli mutatón keresztül kezeljük. Ha például egy tároló objektumait akarjuk megsemmisíteni, akkor ezt így végezhetjük:

```
1. Alkalmazott ** pa = new Alkalmazott * [ 10 ];
2. //pa[ 0 ],..., pa[ 9 ] feltöltése Manager és Alkalmazott
   //típusú objektumokkal
3. for(int i=0; i<10; i++){
4.    pa[ i ]->print();
5.    delete pa[ i ]; //Destruktorhívás
6. }
7. delete [] pa;
```

Az 5. sorban destruktorhívás történik. Felvetődik a kérdés, hogy egy `Manager` típusú objektum megsemmisítése esetében melyik destruktor hívódik meg. Sajnos az `Alkalmazott` osztálybeli destruktor kerül meghívásra, mert csak a virtuális metódusok esetében érvényesül a polimorfizmus. Ennek következtében, ha a `Manager` osztály konstruktora valamilyen erőforrásokat köt le, akkor ezt az `Alkalamazott` osztály destruktora nem fogja felszabadítani, ami erőforráspazarláshoz vezet.

A problémára a megoldást a virtuális destruktorok jelentik. Ez garantálja, hogy alaposztály típusú mutatón keresztül felszabadított származtatott objektumok esetében mindig a mutatott objektumnak megfelelő destruktor hívódik meg. Szabályként alkalmazandó:

Ha egy osztálynak van virtuális függvénye, akkor a destruktorátis virtuálisnak deklaráljuk.

5.9. Privát öröklés

Nyilvános örökítés esetében az utódosztály örökli az ősosztály funkcionalitását. Ennek következtében az utódosztálybeli példányok tartalmazzák az ősosztály által definiált viselkedésmódot. Azt is modhatjuk, hogy bármely utódosztálybeli

példány használható minden olyan helyen, ahol egy ősosztálybeli példány elegendő. Vagyis az utódosztálybeli példányok összeférhetőek az ősosztálybeli példányokkal. A `Manager` típusú példányok, mint az `Alkalmazott` osztály leszármazottai, biztosítják az `Alkalmazott` viselkedésmódját.

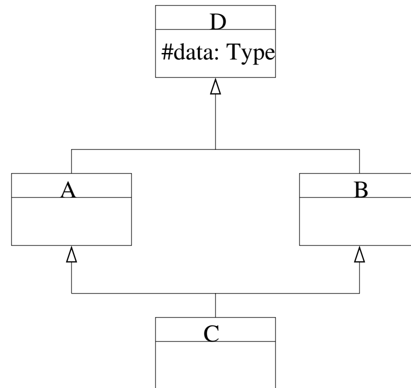
A privát örökítés ennek az ellentétje. Amíg a nyilvános örökítés továbbvitte az ősosztály viselkedésmódját az utódosztályra, addig a privát örökítés teljesen elzárja ezt a funkcionalitást azáltal, hogy minden ősosztálytól átvett nyilvános tag (adat+metódus) priváttá alakul az utódosztályban. A privát öröklésnek hozzáférés-szűkítő hatása van. A privát örökítés szintaktikailag hasonló a nyilváoshoz, a `public` kulcsszó helyett a `private` kulcsszót használjuk.

```
class B : private A{...};
```

Konkrét példaként tekintsük a standard könyvtárban megadott dinamikus tömb osztályt ($vector < T >$). Ez az osztály biztosít egy index operátort, amely végrehajtási időre van optimizálva és nem biztosít hiba/kivételellenőrzést. Ezt a műveletet úgy tudnánk biztonságossá tenni, ha a privát származtatás segítségével "becsomagolnánk" a $vector < T >$ osztályt egy `SafeVector` osztályba, amelyben elzárnánk a nem biztonságos indexelési műveletet és készítenénk az új osztálynak egy biztonságos index operátort.

```
template <class T>
class SafeVector : private vector<T>{
...
public:
   class Overflow{};
   class Underflow{};
   T& operator [] (int index){
      if(index < 0) throw Underflow();
      if(index >= vector<T>::size()) throw Overflow();
      return vector<T>::operator[](index);
   }
...
};
```

A fenti példában a privát örökítést azért használtuk, hogy az ősosztály adatait illetve metódusait átvegyük. Ugyanezt megtehettük volna úgy is, hogy tartal-



5.4. ábra. Többszörös örökítés

mazási kapcsolatot alakítunk ki a két osztály között, amely egy lazább kapcsolatot eredményez a két osztály között.

5.10. Többszörös öröklés

Egy osztálynak több közvetlen bázisosztálya lehet. Az ilyen osztályokból származó példányok több felületet örökölnek, biztosítják mindkét ősosztály szolgáltatásait, kiegészítve ezeket újakkal. Ennek egyetlen előnye a gazdag szolgáltatáskészlet. A Java nyelv nem biztosította a többszörös örökítést, de biztosította a több interfész implementálását. Lényegében a Java azt vette át a C++ nyelvből, amit problémamentesen lehet használni, vagyis általában problémamentesen lehet felületet átvenni.

A többszörös örökléssel kapcsolatosan felmerülő problémát az okozza, hogy egy osztály nemcsak viselkedésmódot definiál, hanem egy tartalmat is, amelyet az osztály adattagjai határoznak meg.

Tekintsük a 5.4 látható osztályhierarchiát: A `D` közvetlen bázisosztálya az `A` és a `B` osztályoknak, így mindkét osztály örökli a `data` nevű adatmezőt. A `C` osztálynak közvetlen ősosztálya az `A` és a `B` osztályok, amely által a `C` osztály kétszer fogja tartalmazni a `data` nevű adatmezőt. Valószínűleg ez nem célja egyetlen

programtervezőnek sem, hogy valami két példányban szerepeljen. A funkcionalitás (metódusok) átvétele nem okoz ilyen jellegű problémát, mert ezek mindenképp csak egyszer szerepelnek, akárhány példányt hozunk létre bármely osztályból. Az adattagok viszont komoly problémát jelenthetnek.

Természetesen, mint mindenre, erre is született megoldás. A virtuális bázisosztályok, bár elég költségesen, de megoldják ezt a problémát, garantálva, hogy minden átörökített adat csak egyetlen példányban fog megjelenni a leszármazott osztály példányaiban.

```
class A{ ...};
class A : public virtual D{...};
class B : public virtual D{...};
class C : public virtual A, public virtual B{...};
```

Általánosan elfogadott, hogy a bonyolult osztályhierarchiák helyett javasolt az interfészek használata. Habár a C++ nyelv nem vezeti be az interfész típust, ez tökéletesen szimulálható, olyan absztrakt osztályokkal, amelyek csak tiszta virtuális függvényeket tartalmaznak. Így megelőzhető az ismétlődő bázisosztályok által megjelenő probléma.

Megjegyzés: A privát és a többszörös öröklést megfontoltan kell alkalmazni, csak olyan esetekben, ha ez másképpen nem oldható meg.

5.11. Feladatok

1. Adott a következő két osztály:

```
class A{
...
public:
   A();
   virtual double getData() const;
};
```

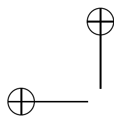
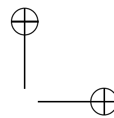
```
class B : public class A{
...
public:
   B();
   double getData() const;
};
```

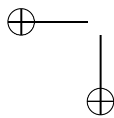
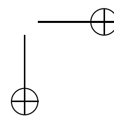
Az alábbi metódushívások mindenikére mondja meg, hogy pontosan melyik osztályban definiált metódus fog meghívódni.

```
A * p1 = new A;
A * p2 = new B;
A a;
B b;

p1->getData();
p2->getData();
a.getData();
b.getData();
```

2. Készítsen egy absztrakt, interfész típusú osztályt, amely a sorra jellemző alapvető műveleteket tartalmazza (betesz, kivesz, üres állapot lekérdezése). Származtasson ezen osztályból két konkrét osztályt, amelyek tömb illetve láncolt lista segítségével valósítják meg a sor adatstruktúrát.

3. Privát örökítés segítségével készítsen egy biztonságosabb dinamikus tömb osztályt.

6. FEJEZET

A STANDARD KÖNYVTÁR

6.1. Bevezetés

Az általánosított programozás (generic programming) a számítástechnika azon területe, amely a következő célkitűzéseket követi:

- megtalálni az algoritmusoknak a leghatékonyabb és egyben legáltalánosabb ábrázolását
- megtalálni az adatstruktúrák legáltalánosabbban megadható formáját
- olyan segédfogalmakat vezetni be, amelyek segítségével az algoritmusok illeszthetők az adatstruktúrákhoz

Összesítve elmondhatjuk, hogy olyan formában kell megadni az adatstruktúrákat és algoritmusokat, hogy azok közvetlenül felhasználhatóak legyenek a szoftverépítésben. Mindezek érdekében tekintsünk még egy pár ötletet:

- Az algoritmust úgy kell megadni, hogy minimális követelményt támasszon az adatokkal szemben, amiken végrehajtódik.
- A konkrét algoritmust egészen addig általánosítani, amíg még nem okoz hatékonysági csökkenést.
- Ha a legáltalánosabb forma nem minden esetben alkalmazható, akkor ezt ki kell egészíteni azon sajátos esetekkel, amelyeket az általános alak nem fed le.

Nagyon sokszor hallani a generikus programozásról és a generatív programozásról. Nézzük meg, pontosan miben is különbözik ez a két fogalom. A generikus programozás elsődleges célja a fogalmak megadásának legáltalánosabb módja. Adatstruktúra- családok illetve algoritmus-családok megadására koncentrál. Ezzel szemben a generatív programozás ezen kívül még magába foglalja a konkrét példányok létrehozásának folyamatát (generálását) is az általános fogalomcsaládokból.

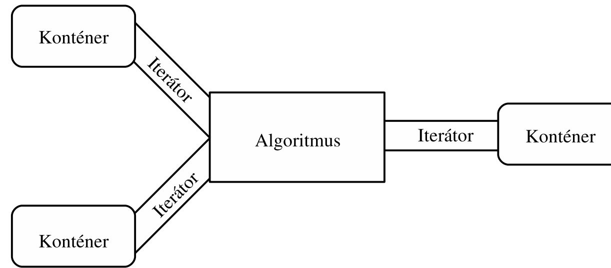
6.2. STL komponensek

A standard könyvtárat nem egy ember tervezte, sőt egyáltalán nem egy tervezett könyvtár. Inkább mondhatjuk, hogy egy válogatás több programozó munkájából. Az eredmény egy nemhomogén komponensgyűjtemény, amely különböző tervezési stílusok jegyeit hordozza magán. Egyik legszembetűnőbb különbség a `string` tároló és a többi tároló között van. Amíg a `string` tárolót biztonságos használatra tervezték, addíg az összes többi tárolónál a hatékonyság az elsőrendű szempont.

A standard könyvtár legfontosabb három komponense a tárolók (konténerek), bejárók (iterátorok) és algoritmusok.

Az olyan osztályokat, amelyek objektumok tárolásával foglalkoznak, tárolóknak nevezzük. A tárolók a tárolási funkció mellett karbantartó műveleteket is biztosítanak. Implementáció szempontjából a tárolt elemek alkothatnak egy tömböt, egy láncolt listát vagy tárolásuk rendezetten történhet valamilyen kulcs alapján.

A bejárok lehetővé teszik, hogy minden konténert egyformán dolgozhassunk fel. Ezt úgy valósítják meg, hogy létezik egy egységes bejáró interfész, amelyet minden egyes tároló megvalósít, ezáltal biztosítva az egységes feldolgozás lehetőségét. Az egyik művelet az egységes bejáró interfészből a ++ operátor, amely a következő elemet jelenti az adott tárolóból. Természetesen ez tömb esetében a következő címen levő elemet jelenti, láncolt lista esetében az aktuális listaelem következő mutatójának megfelelő elemet illetve fa struktúrával ábrázolt tároló esetében valamely kiválasztott bejárási stratégia szerinti következő elem. A fel-



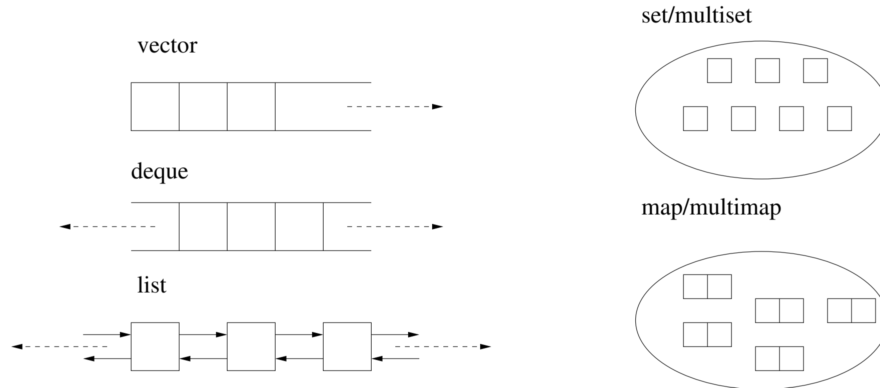
6.1. ábra. STL komponensek

használónak nem kell foglalkoznia az implementációs részletekkel, hiszen a tároló típusától függetlenül, mindig ugyanúgy érhető el a következő elem. Az iterátorok által biztosított interfész a mutatókéhoz hasonló, ezért szokás az iterátort általánosított mutatónak is nevezni.

A standard könyvtár algoritmusai úgy vannak elkészítve, hogy maximálisan együtt tudjanak dolgozni a tárolókkal. Általában ezek az algoritmusok minden tárolóra (esetleg ennek részhalmazára) alkalmazhatóak. Az alkalmazhatóságot az iterátorok biztosítják megteremtve a kapcsolatot a tároló és az algoritmus között.

Első ránézésre úgy tűnik, hogy az általánosított programozás és az objektumorientált programozás homlokegyenest ellenkező programozási paradigmák. Ez csak részben igaz. Amíg az objektum összezárja az adatokat az ezeken végezhető műveletekkel, addíg az általánosított programozás ismét szétválasztja ezeket. A valóság egy picit árnyaltabb, mert az általánosított programozás nem törekszik a teljes művelethalmaz leválasztására, csak a nagyon általános műveletek megadását szorgalmazza az osztályokon kívül, a lehető legáltalánosabb módon. Az adott tárolóra specifikus műveleteket továbbra is tagfüggvényekként implemetálhatjuk. Ha viszont egy művelet elég sok tárolóra jellemző, akkor ezt illik algoritmussal megadni.

Az 6.1. ábra egy olyan algoritmust szemléltet, amelynek bemenetét két tároló képezi és kimenete egy harmadik tároló lesz. Egy konkrét példa lehetne két számsorozat összefésülése, amelyek közül az egyik egy tömbben van, a másik egy láncolt listában és a kimenetet egy bináris rendezőfában helyezzük el.



6.2. ábra. STL konténer típusok (Josuttis [2] nyomán)

6.3. Konténerek

A standard könyvtár kétféle tárolót biztosít: sorozatokat és asszociatív tárolókat. Ezeket a 6.2. rajz szemlélteti.

A `sorozatok` jellemzője, hogy minden elemnek van egy rögzített pozíciója. Ezt a pozíciót csak a beszúrás helye és ideje határozza meg, teljesen független a beszúrt elem értékétől. Ha például egy tömbhöz vagy egy láncolt listához hozzáfűzünk (a végéhez) 5 elemet, akkor ezek az elemek a hozzáfűzés sorrendjében fognak szerepelni a tárolóban. A standard könyvtár háromféle sorozatot biztosít: `vector, deque, list`.

Az asszociatív tárolók valamilyen rendezési kritérium szerinti rendezettséget biztosítanak. Ebben az esetben a beszúrt elem nem függ a beszúrás idejétől, hanem csak az elem értékétől. Ezen tárolók esetében a beszúrás helye nem is adható meg. A szabványos könyvtár a következő asszociatív tárolókat tartalmazza: `set, multiset, map, multimap`.

Az asszociatív tárolók rendezettsége nem jelenti, hogy ezen tárolók elsődleges célja az elemek rendezése. Természetesen rendezni a sorozatokat is lehet. A rendezettség előnye, hogy gyorsabb keresést biztosít az adott tárolóban. Ezen tárolók gyakorlatilag logaritmikus időigényű keresést biztosítanak.

6.3.1. Sorozatok

vector

A `vector` egy dinamikus tömöt megvalósító sablon osztály. Legfontosabb tulajdonságai:

- tárolt elemeknek közvetlen elérést biztosít, vagyis a tömb indexelhető
- a tömb végénél végzett hozzáfűzés és törlés gyors ($\Theta(1)$ időigényű)
- a tömb bármely más pozíciójában a törlés és a hozzáfűzés lassú, mert elemek áthelyezésével jár ($O(n)$ időigényű)

A következő példaprogram szemlélteti a `vector` tároló egyszerű használatát:

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

int main(){
   int i;
   vector<int> v;
   for(i=0; i<5; i++)
      v.push_back(i);
   for(i=0; i<v.size(); i++)
      cout<<v[ i ]<<' ';
   cout<<endl;
}
```

deque

A deque a 'double-ended queue" rövidítése. Olyan dinamikus tömb, amely mindkét végénél terjeszkedhet. Ennek következtében a következő alapvető tulajdonságokkal rendelkezik:

- a tárolt elemeknek közvetlen elérést biztosít, vagyis a tömb indexelhető
- a tömb mindkét végénél végzett hozzáfűzés és törlés gyors ($\Theta(1)$ időigényű)

– a tömb bármely más pozicíójában a törlés és a hozzáfűzés lassú, mert elemek áthelyezésével jár ($O(n)$időigényű)

Tekintsük a következő példaprogramot a használatra:

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

int main(){
   int i;
   deque<int> v;
   for(i=0; i<5; i++)
      v.push_front(1+i*0.75);
   for(i=0; i<v.size(); i++)
      cout<<v[ i ]<<' ';
   cout<<endl;
}
```

`list`

A `list` tároló egy kétszeresen láncolt listával van implementálva. Mivel a lista elemeit elérni csak szekvenciálisan lehet, ezért a lista nem egy közvetlen elérésű adatstruktúra és ennek következtében nem biztosít index operátort. Ha egy tároló biztosít index operátort, akkor illik ezt konstans időigényű művelettel megvalósítani.

A listák erőssége abban rejlik, hogy a beszúrás, illetve a törlés a lista bármelyik pozíciójában gyors. Tekintsük a következő példaprogramot a lista használatára:

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

int main(){
   char i;
   list<char> l;
   for(i='a'; i<='z'; i++)
      v.push_back(i);
```

```
    while(! l.empty()){
        cout<<l.front()<<' ';
        l.pop_front();
    }
    cout<<endl;
}
```

`Közönséges C stílusú tömbök`

A közönséges C stílusú tömbök nem STL konténerek, hiszen nem rendelkeznek olyan tagfüggvényekkel, mint a `size()` vagy az `empty()`. Az STL rendszert viszont úgy tervezték, hogy ezekre a tömbökre is használhatók legyenek az algoritmusok.

6.3.2. Asszociatív tárolók

Az asszociatív tárolók egy adott rendezési kritérium szerint rendezve tárolják elemeiket. Alapértelmezetten a tárolók a $<$ művelettel hasonlítják elemeiket. A tároló létrehozásakor viszont megadható olyan függvény, amely meghatározza a rendezési kritériumot.

A szabvány nem írja elő, hogy ezeket a tárolókat milyen adatstruktúrákkal kell implementálni, csak a különböző műveletek időigényét rögzíti. Az asszociatív tárolókat általában kiegyensúlyozott bináris keresőfákkal ábrázolják. A standard könyvtár a következő asszociatív tárolókat tartalmazza:

`set`

A `set` olyan adathalmaz, amelyben az elemek értékeik alapján rendezettek. Minden elem csak egyszer szerepelhet a halmazban.

`multiset`

A `multiset` a `set` tárolóhoz hasonló azzal a különbséggel, hogy megengedi az elemek többszörös előfordulását.

`map`

A `map` tárolót még asszociatív tömbnek is nevezzük, amelybe `(kulcs, érték)` elempárokat helyezhetünk el, `kulcs` szerinti rendezettségi sorrendben. Azért nevezzük asszociatív tömbnek, mert olyan tömb, amelyben az index bármilyen típusú lehet. Minden `kulcs` csak egyszer szerepelhet az asszociatív tömbben.

`multimap`

A `multimap` a `map` tárolóhoz hasonló, de megengedi a kulcsértékek többszörös előfordulását. A tárolót jellegzetesen szótárak megvalósítására alkalmazzák.

Minden asszociatív tárolónak van egy opcionális sablonparamétre, amelyen keresztül átadható a rendezési kritérium. Az asszociatív tárolókat iterátorok segítségével dolgozzuk fel, ezért a konkrét példákat majd az iterátorok bemutatása után adunk.

6.3.3. Konténer-adapterek

Az adaptív tulajdonság alkalmazkodókészséget jelent. A konténer adapterek gyakorlatilag különleges igényeknek megfelelő, átalakított tárolókat jelentenek. Ilyen tárolók a `stack, queue, priority_queue`. A verem (`stack`) és a sor (`queue`) tárolókat tipikusan a `dequeue` tárolóból, míg a prioritási sort (`priority_queue`) a `vector` tárolóból adaptálják.

6.4. Iterátorok

Az iterátor egy olyan objektum, amely képes egy elemhalmaz bejárására. Ezen elemhalmaz lehet a standard könyvtár valamely tárolója, avagy ennek valamely részhalmaza. Az iterátor egy pozíciót határoz meg egy adott tárolóban. Ennek következtében az iterátor inicializálása egy asszociáció, amelyen keresztül hozzárendeljük egy tároló valamely pozícóját. Az iterátort gyakorlatilag tekint-

hetjük a tömbhöz rendelt mutató általánosításának is, hiszen alapvetően ugyanazt a műveletethalmazt biztosítja, mint ezen mutatók.

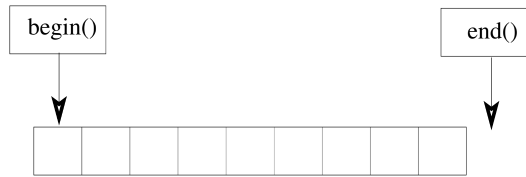
Minden iterátor alapvetően a következő művelethalmazt biztosítja:

1. * operátor, az tároló aktuális pozícióján levő elemet téríti vissza
2. ++, −− operátorok, az aktuális pozíció növelése illetve csökkentése, amelynek következtében az új aktuális pozíció a tároló következő illetve megelőző eleme lesz
3. ==, != operátorok, annak ellenőrzése, hogy két iterátor azonos vagy különböző pozíciókat ábrázolnak egy adott tárolón belül
4. =, értékadó operátor, lehetővé teszi adott pozíción levő elem megváltoztatását

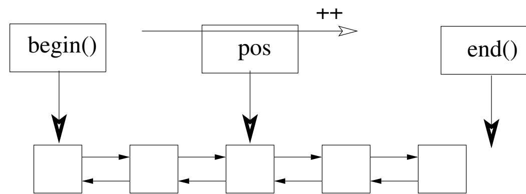
Láthatjuk, hogy a fenti műveletek, pontosan a tömbre állított mutatók működését jelentik. Eddig csak az iterátor interfészről beszéltünk, amely alapvetően meghatározza ezen objektumok viselkedésmódját. Az iterátor implementációja teljesen tárolóspecifikus. Másképpen kell a következő elemet elérni egy dinamikus tömbben, egy láncolt listában, vagy akár egy kiegyensúlyozott bináris keresőfában. Amíg a dinamikus tömb iterátort egy T * pointerrel ábrázolhatjuk, ahol T a tárolt elemek típusa, addig a láncolt lista esetében ez egy listaelem mutató lesz és bináris fa esetében egy csúcsmutató. A következő elem meghatározása is ennek megfelelően fog történni.

Az STL iterátorok úgy vannak megvalósítva, hogy maximális hatékonysággal együtt tudjanak dolgozni a tárolókkal. A leggyakoribb megvalósítás egy beágyazott osztály, amely az iterátorokra jellemző műveleteket implementálja. A tárolók is ismerik az iteráror típust és két alapvető műveletet biztosítanak, amely gyakorlatilag biztosítja a tároló bejárhatóságát:

– `begin()` - Egy olyan iterátor objektumot térít vissza, amely a tároló legelső pozícióját reprezentálja.
– `end()` - Egy olyan iterátor objektumot térít vissza, amely a tároló végét reprezentálja.



6.3. ábra. A *begin()* és az *end()* tömb konténer esetében



6.4. ábra. A pos iterátor mozgása a lista tárolóban

A tároló vége az utolsó elem mögött van, gyakorlatilag ez az objektum az utolsó utáni pozíciót reprezentálja. Közönséges C stílusú tömb esetében az 6.3. ábra szemlélteti a `begin()` és az `end()` által visszatérítendő pozicíókat.

A fenti két függvény a `[begin(), end())` félig nyitott tartományt határozzák meg. A tartomány ezen megadási módja a következő előnyökkel jár:

- egyszerűen megadható a tartomány vége: `iterator != end()`
- az üres tartomány feltétele is nagyon egyszerűen adható meg: `begin() == end()`

Az eddigiek szemléltetésére tekintsük a listáknál adott példát, amelyben a lista bejárását iterátor segítségével végezzük. Az 6.4. rajz szemlélteti a bejárás folyamatát.

```
1. list<char> l;
2. for(char c='a'; c<='z'; c++)
3.    l.push_back(c);
4. list< char > :: const_iterator pos;
5. for( pos =l.begin(); pos != l.end(); ++pos )
6.    cout << *pos<<' ';
7. cout<<endl;
```

A fenti példában az iterátor deklarációja a 4. sorban van. Minden tároló definiál egy `iterator` és egy `const_iterator` típust a tároló osztályon belül. Például a `list` tároló esetében ez a következőképpen nézne ki:

```
template < class T >
class list{
public:
   typedef ... iterator;
   typedef ...const_iterator;
   ...
};
```

A pontos iterátortípus implementációfüggő, a bevezetett típusnevek viszont lehetővé teszik, hogy tárolón kívül is használjuk ezeket a típusneveket. Mivel ezek a típusnevek minden tároló esetében, a tároló osztályon belül vannak bevezetve, ezért ennek megfelelően a hatókör operátor segítségével hivatkozhatunk rájuk.

Az 5. sorban három iterátorműveletet használtunk a következő sorrendben: értékadó operátor, összehasonlító != operátor és ++, növelő operátor. Ezek mind szabványos iterátor műveletek. A 6. sorban a tárolt elem elérésére a * operátort használtuk.

A tárolónak ez a fajta bejárása tárolófüggetlen. Ez azt jelenti, hogy bármely tárolót ugyanezzel a konstrukcióval fogunk bejárni. Az egyetlen különbség az iterátor deklarációja lesz, itt kell megadni a pontos tárolótípust.

Példák asszociatív tárolókra

Készísünk egy egész számokat tartalmazó halmaz (`set`) tárolót. Helyezzünk bele elemeket, majd járjuk be a tárolót kiíratva a tárolt elemeket, amelyek növekvő sorrendben fognak megjelenni a kimeneten.

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

int main(){
```

```
   typedef set<int> IntSet;
   IntSet s;
   for(int i=0; i<100; i++)
      s.insert(rand());
   IntSet::const_iterator pos;
   for(pos = s.begin(); pos != s.end(); ++pos)
      cout<<*pos<<'\t';
   cout<<endl;
   return 0;
}
```

A második példát adjuk a `map` tárolóra. Ebbe a tárolóba elempárokat kell belehelyeznünk. Az elempár létrehozására a `make_pair` függvénysablont használhatjuk, amely két bármilyen típusú elemből egy elempárt alkot, összezárva ezeket egy struktúrába. A struktúra mezőire a `first` és a `second` adattagokkal hivatkozhatunk.

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main(){
   //Tároló létrehozása és feltöltése
   typedef map<string, double> ArLista;
   ArLista l;
   l.insert(make_pair("paradicsom",10.5));
   l.insert(make_pair("alma",6.5));
   l.insert(make_pair("hagyma",4.5));
   l.insert(make_pair("padlizsan",15.3));
   //Tároló bejárása
   ArLista:: iterator it;
   for(it = l.begin(); it != l.end(); ++it)
      cout<<it->first<<" : "<<it->second<<endl;
   return 0;
}
```

A fenti példában a `map` tárolóba az `insert` tagfüggvény segítségével helyeztük be az elemeket. Egy másik lehetőség lenne a tároló asszociatív tömbként való kezelése és az elemeknek az index operátorral való behelyezése. Az asszociatív tömbben az index bármilyen típusú lehet és ha egy nem létező indexű elemnek

értéket adunk, akkor ez az új elempár behelyezését eredményezi. A fenti programban a feltöltést helyettesíthetjük a következő utasítássorozattal.

```
l["paradicsom"]=10.5;
l["alma"]=6.5;
l["hagyma"]=4.5;
l["padlizsan"]=15.3;
```

6.4.1. Iterátor kategóriák

Az iterátorok az alapvető műveleteken kívül rendelkezhetnek extra képességekkel. Ezek a képességek a tároló belső struktúrájától függenek. Az STL általában csak azokat a műveleteket biztosítja, amelyeket hatékonyan lehet implementálni. Ha a tároló direkt elérésű (például a `vector` és a `deque`), akkor az iterátorra is értelmezettek lesznek a direkt elérés műveletei.

A standard könyvtár tárolói a következő két kategóriájú iterátorokat biztosítják:

1. Kétirányú bejáró

A kétirányú bejáróra értelemzhető úgy a ++, mint a −−művelet. Ilyen erősségű iterátort biztosítanak a `list, set, multiset, map, multimap`

2. Közvetlen hozzáférést biztosító bejáró

Ez a bejáró rendelkezik a kétirányú bejárók összes tulajdonságával, ezt kiegészítve a közvetlen hozzáférés műveleteivel. Ezek a közönséges mutatókra alkalmazható műveletek lesznek: `it+n, it-n , it1<it2, it1>it2`. A `vector` és a `deque` tárolók biztosítanak ilyen erősségű bejárót.

Az általánosított programozás alapelve, hogy olyan kódot készítsünk, amely maximálisan tárolófüggetlen. Például a következő kódrészlet bármely tárolóra használható:

```
for(pos = coll.begin(); pos !=coll.end(); ++pos){

}
```

Ez a kódrészlet viszont csak közvetlen elérésű tárolókra:

```
for(pos = coll.begin(); pos <coll.end(); ++pos){
```

```
}
```

A két kódrészlet között az egyetlen különbség az iterátorok összehasonlításában van. Az első kódrészletben a $!=$ operátort használtuk, ezt a műveletet a kétirányú bejáró is biztosítja. A második kódrészletben a $<$ operátorral hasonlítottuk össze a bejárókat, ezt a műveletet viszont csak a közvetlen hozzáférésű tárolók iterátorai biztosítják.

6.5. Algoritmusok

6.5.1. Bevezetés

A standard könyvtár algoritmusai tárolók feldolgozására íródtak. Olyan alapvető műveleteket biztosítanak, mint rendezés, keresés, módosítás és különféle numerikus feldolgozások. Az algoritmusok nem tagfüggvények, hanem olyan globális függvények, amelyek többféle tárolóra is alkalmazhatóak.

Minden algoritmus egy vagy több tartományon dolgozik. Egy tartomány lehet egy egész tároló, vagy annak egy része. A tartományokat két iterátorral határozzuk meg, ami nagy rugalmasságot biztosít és kevés biztonságot eredményez. A hívónak kell gondoskodnia arról, hogy a két iterátor egy érvényes tartományt határozzon meg, vagyis a megadott kezdeti pozícióból kiindulva eljuthatunk a megadott végpozícióig. A programozó feladata úgy megadni a két iterátort, hogy ezek ugyanarra a tárolóra hivatkozzanak és a kezdőpozíció ne legyen a végpozíció után.

Minden algoritmus félig nyitott tartományon dolgozik, amelyet $[begin, end)$-el jelölünk. A félig nyitott tartomány előnyeit már az előző részben megfogalmaztuk. Mindezek szemléltetésére tekintsük a következő példát:

```
#include <iostream>
#include <algorithm>
#include <list>

using namespace std;
```

```
int main(){
   int i;
   list<int> l;
   list<int>::iterator it1, it2;
   for(i=10; i<=30; ++i)
      l.push_back(i);
   it1 = find(l.begin(), l.end(), 20);
   it2 = find(l.begin(), l.end(), 25);
   cout<<"max: "<<*max_element( it1, it2)<<endl;
   return 0;
}
```

A fenti példában egy láncolt listát feltöltünk elemekkel, majd két keresés segítségével meghatározunk egy tartományt. Ennek a tartománynak meghatározzuk a legnagyobb elemét.

6.5.2. Az algoritmusok jellemzése

1. Az algoritmusokat hatékonyságra tervezték, nem biztonságra.
2. Az algoritmusok egy vagy több tartományon dolgoznak
3. Általában az első tartományt két iterátorral határozzuk meg.
4. Ha több tartományon dolgozik az algoritmus, akkor a másodlagos, illetve harmadlagos tartományok esetében elégséges megadni a tartomány elejét, hiszen az elsőként megadott tartomány elemszáma egyértelműen meghatározza ezen tartományok végét.
5. A hívónak gondoskodnia kell arról, hogy a céltartomány megfelelő méretű legyen.
6. A hívónak kell gondoskodnia arról, hogy a megadott tartományok érvényesek legyenek.
7. Az algoritmusok felülírásos üzemmódban dolgoznak. Átkapcsolhatunk beszúró üzemmódba úgy, hogy speciális, beszúró iterátorokat használunk (`insert_iterator`).

6.5.3. Függvényobjektumok

Mindent, ami függvényként viselkedik, azt függvénynek nevezhetünk. Ha olyan objektumot hozunk létre, amely függvényként viselkedik, akkor ezt függvényobjektumnak nevezzük. Mit is jelent a függvényként való viselkedés? A funkcionális viselkedés azt jelenti, hogy a függvényhez hasonlóan hívható. Például:

```
f(arg1, arg2, ...)
```

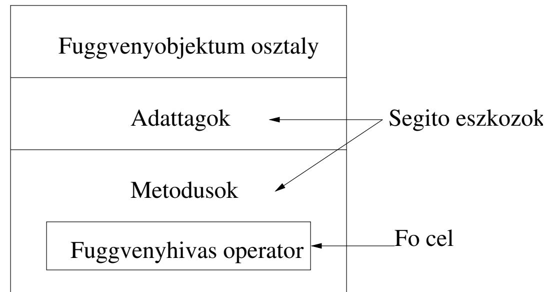
Ahhoz, hogy egy osztály példányai függvényként viselkedhessenek, az osztálynak túl kell terhelnie a függvényhívás operátort.

```
class X{
...
public:
   Típus operator()(arg)
   ...
};
```

Az `X` osztály példányai függvényobjektumok lesznek, hiszen függvényként viselkednek:

```
X fo;
...
fo(arg);
```

A `fo(arg)` hívás gyakorlatilag a `fo.operator()( arg)` hívással egyenértékű. Nézzük meg, hogy mivel "okosabb" a függvényobjektum, mint a közönséges függvény?



6.5. ábra. Függvényobjektum

1. A függvényobjektum egy intelligens függvény, amely függvény és objektum egyszerre. Ennek következtében hívható függvényként és állapotot is tárol. Ezt az állapotot megváltoztathatja a függvényhívások következtében. Ha nem változtatja állapotát, akkor csak függvényként viselkedik.A függvényobjektum típus szerkezetét az 6.5. ábra szemlélteti.

2. A függvényobjektumok gyorsabbak, mint a közönséges függvények. Létrehozásuk fordítási időben történik, több részletet kap a fordító a függvényről, amelynek következtében jobb kódot tud generálni.

Példák függvényobjektumok használatára:

1. Egy egész számokat tartalmazó tároló minden elemét növeljük 10-el.

2. Egy egészeket tartalmazó tároló minden elemét növeljünk egy állandó értékkel. (állandó-fordítási időben ismert érték)

3. Egy egészeket tartalmazó tároló minden elemét növeljünk egy értékkel. Határozzuk meg a tárolóban levő elemek számát.

Mindhárom feladatban használjuk a `for_each` algoritmust. Ez az algoritmus egy adott műveletet hajt végre egy megadott tartomány minden egyes elemére. A `for_each` algoritmus egy lehetséges implementációja a következő:

```
template <class Iterator, class Operation>
Operation for_each(Iterator first, Iterator last,
                        Operation op)
{
   while(first != last){
      op(*first);
      ++first;
```

```
    }
    return op;
}
```

1. Megvalósítás

```
void add10(int& elem){
    elem += 10;
}

int x[] = {0, 1, 2, 3, 4};
for_each(&x[0], &x[5], add10);
```

2. Megvalósítás

Egy függvénysablont készítünk. Ebben a függvénysablonban azonban, a megszokottól eltérően, nem típust adunk át paraméterként, hanem egy értéket. A mi esetünkben ez egy közönséges egész szám lesz, a hozzáadandó érték.

```
template< int value>
void add(int & elem){
    elem += value;
}

int x[] = {0, 1, 2, 3, 4};
for_each(&x[0], &x[5], add<10>);
```

3. Megvalósítás

Egy függvényobjektum osztályt készítünk, amelyben a függvényhívás operátor fogja a növelést végezni.

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

class AddValue{
private:
    int value;
    int counter;
public:
```

```
    AddValue(int _value) : value(_value), counter( 0){}
    int getCounter(){ return counter; }
    void operator()(int& elem){ elem += value; ++counter;}
};

int main(){
    int x[]={ 1, 2, 3, 4, 5 };
    AddValue fo(10);
    fo = for_each(&x[0], &x[5], fo);
    cout<<fo.getCounter()<<endl;
    return 0;
}
```

Most pedig tekintsünk egy másik példát függvényobjektumra. Vezeték és keresztnévből álló személyneveket szeretnénk alfabetikus sorrendben elhelyezni egy tárolóban. A `set` tároló biztosítja az elemek rendezett tárolását. Jelen esetben személyeket kell összehasonlítani és ennek alapján elhelyezni a tárolóban. A set tároló lehetőséget nyújt az elemtípus és a rendezési kritérium megadására.

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

using namespace std;

struct Szemely{
    string vezeteknev;
    string keresztnev;
    Szemely(const string pvezeteknev="",
            const string pkeresztnev="" ):
    vezeteknev(pvezeteknev), keresztnev(pkeresztnev){}
};

class SzemelyHasonlitas{
public:
    bool operator()(const Szemely& sz1, const Szemely& sz2){
        if(sz1.vezeteknev == sz2.vezeteknev)
            return sz1.keresztnev < sz2.keresztnev;
        else
            return sz1.vezeteknev < sz2.vezeteknev;
    }
};

typedef set<Szemely, SzemelyHasonlitas> SzemelyHalmaz;
```

```
int main(){
   SzemelyHalmaz s;
   Szemely sz;
   while(true){
      if(cin>>sz.vezeteknev){
         cin>>sz.keresztnev;
         s.insert(sz);
      }
      else break;
   }

   SzemelyHalmaz :: iterator it;
   for(it = s.begin(); it != s.end(); ++it)
      cout<<it->vezeteknev<<" "<<it->keresztnev<<endl;
   return 0;
}
```

6.5.4. Az algoritmusok osztályozása

Különböző algoritmusok különböző célokat szolgálnak. Céljuk alapján a következő algoritmuskategóriákat határozhatjuk meg:

1. Nem módosító algoritmusok
2. Módosító algoritmusok
3. Törlési algoritmusok
4. Áthelyező algoritmusok
5. Rendezési algoritmusok

6.5.5. Nem módosító algoritmusok

Ezen algoritmusok sem az elemek sorrendjét, sem pedig ezek értékeit nem változtatják meg. Ide tartoznak a keresési algoritmusok, minimum, maximum meghatározó algoritmusok, tartomány elemszámát meghatározó algoritmus stb. Amennyiben a `for_each` algoritmusnak olyan műveletet adunk át, amely nem változtatja meg a tartomány elemeit, a `for_each` is nem módosító algoritmusnak minősül. Gyakran használt algoritmus két tartomány lexikografikus összehasonlítása. Tekintsük erre a következő példát:

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

int main(){
   string s1("alma");
   string s2("almafa");
   cout<<lexicographical_compare(s1.begin(), s1.end(),
s2.begin(), s2.end())<<endl;
   return 0;
}
```

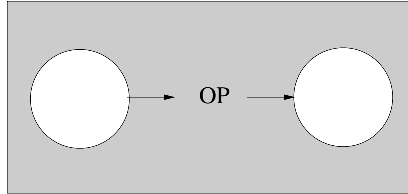
6.5.6. Módosító algoritmusok

A módosító algoritmusok megváltoztatják a tartomány elemeit. Amennyiben a `for_each` algoritmusnak átadott művelet módosítja a paraméterét, a `for_each` módosító algoritmusnak számít. Ebbe a kategóriába tartoznak a másoló algoritmusok (`copy`), amelyek a céltartományt módosítják meg.

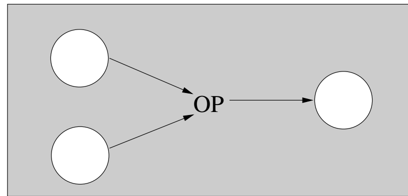
```
vector<int> v;
int x[ 100 ];
//v feltöltése max. 100 elemmel
copy(v.begin(), v.end(), &x[ 0 ]);
```

Nagyon gyakran használjuk a `transform` algoritmust, amely kétféleképpen is viselkedhet:

a. Módosítva másol elemeket egy forrástartományból egy céltartományba. Ezt a viselkedésmódot az 6.6. ábra szemlélteti.



6.6. ábra. A transform algoritmus (1)



6.7. ábra. A transform algoritmus (2)

```
#include <iostream>
#include <algorithm>
#include <string>
#include <cctype>
using namespace std;
int main(){
   string s1("alma");
   transform(s1.begin(), s1.end(), s1.begin(), ::toupper);
   cout << s1<<endl;
   return 0;
}
```

b. Két tartományra elemenként alkalmaz valamely megadott műveletet, az eredményt egy harmadik tartományba helyezve. Ezt a viselkedésmódot az 6.7. ábra szemlélteti.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <cstdlib>
```

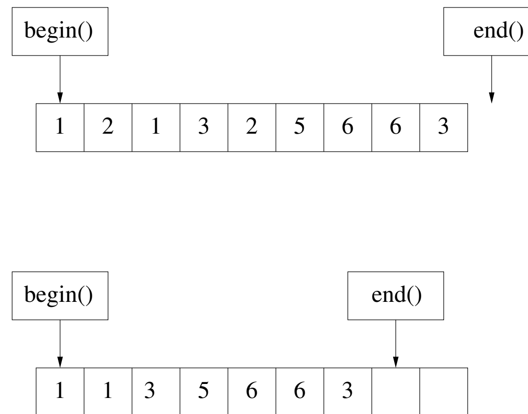
```
using namespace std;

int myOperation(int a, int b){
   return a+b;
}
int main(){
   vector<int> v1, v2;
   int i;
   for(i=0; i<10; ++i){
      v1.push_back(rand() % 100);
      v2.push_back(rand() % 100);
   }
   vector<int>v3(v1.size());
   transform(v1.begin(), v1.end(), v2.begin(),
      v3.begin(), myOperation);
   for(i=0; i<v3.size(); ++i)
      cout<<v1[i]<<" "<<v2[i] <<" "<<v3[i]<<endl;
   return 0;
}
```

6.5.7. Törlési algoritmusok

A törlés a módosítás egy sajátos formája. A törlési algoritmusok (`remove` algoritmuscsalád) a törlést felülírással valósítják meg. A törölt elemeket felülírják a tartomány következő elemeivel és visszatérítik az új tartományvéget. A törlési algoritmusok nem változtatják meg a tartományban levő elemek számát, vagyis csak logikai törlést végeznek (lásd 6.8. ábra) Szükség esetén a fizikai törlés is megvalósítható.

6.5.8. Áthelyező algoritmusok

Az áthelyező algoritmusok az elemek sorrendjét módosítják. Természetesen ezek az algoritmusok nem alkalmazhatók asszociatív konténerekre. Ebbe a kategóriába tartozik a `reverse`, `rotate`, `random_shuffle`, `next_permutation`, `prev_permutation` stb. algoritmusok. A következő példa egy halmaz permutációit határozza meg. Mivel a lexikografikusan legkisebb halmazból indulunk ki, ezért elégséges a `next_permutation` használata.



6.8. ábra. A remove algoritmus

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

void print(int *x, int n){
   int i;
   for(i=0; i<n; ++i)
      cout<<x[ i ]<<" ";
   cout<<endl;
}

int main(){
   int x[]={ 1, 2, 3};
   int i;
   print(x, 3);
   while(next_permutation(&x[0], &x[3])){
      print(x,3);
   }
   return 0;
}
```

6.5.9. Rendezési algoritmusok

A rendezési algoritmusok is módosítják az tárolók elemeit. A rendezés viszont időigényesebb, mint az előző kategóriába tartozó egyszerű átrendező algoritmusok. Természetesen ezek az algoritmusok sem alkalmazhatók asszociatív konténerekre. A szabványos könyvtár több rendezési algoritmust is biztosít.

A `sort()` függvény általában a gyorsrendezés algoritmussal (Quicksort) van implementálva. A legrosszabb esetben $O(n^2)$ időigényű, különben $O(n * log\ n)$.

A `partial_sort()` függvény kupacrendezéssel (Heapsort) van megvalósítva. Ez garantálja a $O(n * log\ n)$ aszimptotikus időigényt. A gyakorlatban általában legalább kétszer lassúbb, mint a gyorsrendezés. A kupacrendezést úgy is használhatjuk, hogy először kupaccá alakítjuk az adathalmazt `make_heap()`, majd erre alkalmazzuk a `sort_heap()` algoritmust.

A `stable_sort()` az összefésüléses rendezés implementációja. Mivel ez a rendezés megőrzi az azonos értékű elemek sorrendjét, ennek következtében ez egy stabil rendezés. Egyetlen hátránya az előző rendezésekkel szemben, hogy plusz tárigényre van szüksége.

A C++ szabvány nem írja elő pontosan, hogy melyik algoritmust kell használni a fenti függvények implementációjához, csak a függvények komplexitását rögzíti. A `sort()` SGI implementációja egy úgynevezett introsort algoritmust használ, amely kikerüli a legrosszabb esetet, átváltva ilyenkor kupacrendezésre.

A rendezési algoritmusoknak meg kell adnunk a rendezendő tartományt, amelyet két iterátorral azonosítunk és opcionális paraméterként megadhatjuk az összehasonlító függvényt is. Ha nem adunk meg összehasonlító függvényt, akkor az algoritmus a "<" operátort fogja használni. A különböző rendezési algoritmusokat érdemes összehasonlítani a következő példaprogrammal, amely időmérést is tartalmaz.

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

int main(){
clock_t start, stop;
```

```
    vector<int> v1, v2, v3;
    v1.reserve(1000000);
    v2.reserve(1000000);
    v3.reserve(1000000);
    int i;
    for(i=0; i<1000000; ++i){
       int x = random();
       v1.push_back(x);
       v2.push_back(x);
       v3.push_back(x);
    }

    start = clock();
    sort(v1.begin(), v1.end());
    stop = clock();
    cout<< "Quicksort:"<<(double)(stop-start)
              / CLOCKS_PER_SEC <<endl;

    start = clock();
    stable_sort(v2.begin(), v2.end());
    stop = clock();
    cout<< "Mergesort:"<<(double)(stop-start)
              / CLOCKS_PER_SEC<<endl;

    start = clock();
    make_heap(v3.begin(), v3.end());
    sort_heap(v3.begin(), v3.end());
    stop = clock();
    cout<< "Heapsort:"<<(double)(stop-start)
              / CLOCKS_PER_SEC<<endl;
    return 0;
}
```

6.6. Feladatok

1. Olvassunk be egy sorokra tördelt szöveget, amelyet egy `vector<string>` típusú tárolóban helyezünk el. Keretezzük be a szöveget és írassuk ki a szabványos kimenetre.

Példa:

Bemenet:

```
Ez egy keretezendő
szöveg. E köré kell
keretet helyezni.
```

Kimenet:

```
************************
* Ez egy keretezendő   *
* szöveg. E köré kell  *
* keretet helyezni.    *
************************
```

2. Tervezzünk meg és valósítsunk meg egy permutált tárgymutatót. A következő példa szemlélteti az ilyen tárgymutatót.

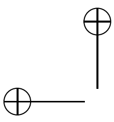
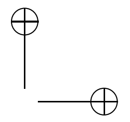
Bemenet:

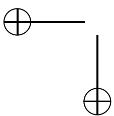
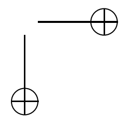
```
kicsi kutya tarka
van füle és farka
```

Kimenet:

```
van füle                és farka
van füle és             farka
van                     füle és farka
                        kicsi kutya tarka
kicsi                   kutya tarka
kicsi kutya             tarka
                        van füle és farka
```

3. Olvassunk be a szabványos bemenetről egy mondatot és állítsuk elő a mondat szavainak összes lehetséges permutációját.

7. FEJEZET

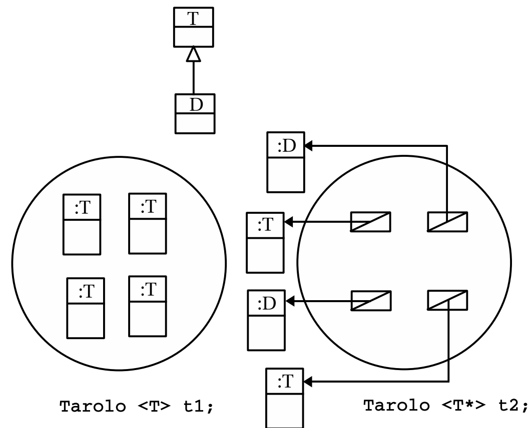
TÁROLÓK

7.1. Tárolók jellemzése

Az STL célja az, hogy egyszerre hatékony és általános algoritmusokat kínáljon. A hatékonyság érdekében, a gyakran használt adatelérő függvények esetében, a tárolók nem használnak virtuális függvényeket. Ennek következtében mindenik tároló támogatja az alapvető műveleteknek egy bizonyos halmazát. Azon műveletek, amelyek nem minden tárolóban valósíthatóak meg hatékonyan, nem szerepelnek ebben a közös adathalmazban. A bejárhatóságot sem egy közös felülettosztállyal valósítják meg, hanem minden tároló biztosít egy olyan iterátor típust, amely lehetővé teszi a bejárhatóságát és amelyre alkalmazhatóak az alapvető iterátorműveletek. Következtetésül elmondhatjuk, hogy nincs közös bázisosztályuk sem a tárolóknak, sem pedig a bejáróknak, ennek következtében pedig nincs futásidejű típusellenőrzés.

Az STL tárolói a következő előnyös tulajdonságokkal rendelkeznek:

- egyszerűek és hatékonyak
- minden tároló biztosítja a szabványos műveleteket
- minden tároló saját bejárót biztosít, amelyek lehetővé teszik bizonyos műveletek végrehajtását szabványos néven és jelentéssel
- a tárolók alapértelmezetten homogének; heterogén tárolókat úgy készíthetünk, hogy egy homogén mutatókat tartalmazó tárolót hozunk létre, amelyben a mutató típusa egy közös bázisosztály



7.1. ábra. Homogén és heterogén C++ tárolók

7.2. Tárolók alapműveletei

Minden tárolónak van alapértelmezett konstruktora, másoló konstruktora és destruktora. Ezen kívül minden konténer inicializálható egy iterátorokkal azonosított tartománnyal. Ezt a konstruktort használjuk, ha a tárolót valamely másik tároló résztartományával akarjuk inicializálni.

Konstruktor és destruktor:

```
Container c; //üres tároló létrehozása
Container c1(c2);//másoló konstruktor
Container c(beg, end);//tároló létrehozása valamely más tároló megadott résztartományából
~Container();//destruktor
```

Példák a konstruktorok használatára:

```
list<int> l;
//l feltöltése
vector<float> v(l.begin(), l.end());

int x[]={ 1, 20, 2, 10, 5};
set<int> s(x, x+3);
```

Mérettel kapcsolatos műveletek:

Minden tároló három mérettel kapcsolatos műveletet biztosít:

1. `size()`, a tároló aktuális elemeinek számát adja vissza.
2. `empty()`, a tároló üres voltának lekérdezése.
3. `max_size()`, a tárolóba behelyezhető maximális elemszámot tartalmazza. Ez az érték implementációfüggő.

Összehasonlítási műveletek:

A tárolókat össze is lehet hasonlítani. Az összehasonlításokat a megszokott összehasonlító operátorokkal végezzük: ==, !=, <, <=, >, >=. Az összehasonlításokat a következő szabályok alapján kell végezni:

1. Csak azonos típusú tárolókat lehet összehasonlítani.
2. Két tároló akkor egyenlő, ha egyenlő elemeket ugyanolyan sorrendben tartalmaznak. Az elemek összehasonlítására az == operátort használjuk.
3. A konténereket összehasonlíthatjuk a megszokott relációs operátorok segítségével.

A következő példaprogram az összehasonlításokat szemlélteti:

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

main(){
   vector<int> v1;
   for(int i=0; i<10; i++)
      v1.push_back(i);
```

```
    vector<int> v2(v1.begin(), v1.end());
    cout<<(v1 ==v2)<<endl;

    vector<int> v3(v1.rbegin(), v1.rend());
    cout<<(v1<v3)<<endl;

    return 0;
}
```

Más műveletek:

`insert(pos, elem)`, beszúrja a pos pozícióba az elem másolatát
`erase(beg, end)`, törli a két iterátorral megadott tartományból az elemeket
`clear()`, törli a tároló összes elemét
`begin()`, egy iterátort ad vissza a tároló első elemére
`end()`, a tároló végét jelző iterátor
`rbegin()`, iterátor a fordított sorrendnek megfelelő első elemre
`rend()`, iterátor a fordított sorrendnek megfelelő tároló végére
`c1.swap(c2)`, kicseréli a két tároló tartalmát

7.3. Sajátos tárolóműveletek

7.3.1. A vector tároló

A dinamikus tömböt a megszokottól eltérően is létrehozhatjuk:

Művelet	Hatás
`vector<T> c(n)`	Létrehoz egy `n` elemű dinamikus tömböt. Az elemeket a `T` típus alapértelmezett konstruktorával hozza létre.
`vector<T>c(n, elem)`	Létrehoz egy `n` elemű dinamikus tömböt. Az elemeket az `elem` másolatával hozza létre.

A dinamikus tömb rendelkezik méretre vonatkozó specifikus műveletekkel is:

Művelet	Hatás
`capacity()`	Visszatéríti a dinamikus tömb kapacitását elemszámban kifejezve. Gyakorlatilag azt adja meg, hogy hány elem számára van hely a tárolóban újabb helyfoglalás nélkül.
`reserve(int)`	Növeli a tároló méretét a paraméternek megfelelően.
`resize(int)`	Újraméretezi a dinamikus tömböt. Hívás után a tömb az új méretnek megfelelő elemet fog tartalmazni, minden elem implicit kezdőértéket kap.

Tipikus hibát képez a következő kódrészlet:

```
vector<int> v; //kapacitás: 0, elemszám: 0
v.reserve(10);¨//kapacitás: 10, elemszám: 0
int i;
for(i=0; i<v.capacity(); ++i)
   v[ i ] = i;
for(i=0; i<v.size(); ++i)
   cout<<v[ i ]<<endl;
```

A fenti kódrészlet semmit nem fog megjeleníteni a kimeneten, ugyanis az index operátor nem módosítja az elemszámot. Ha a v[i]= i utasítást kicserélnénk v.push_back(i) -re, az eredmény az elvárásoknak megfelelő lenne.

Értékadó műveletek

Művelet	Hatás
`c1=c2`	a `c1` tárolóba bemásolja a `c2` tároló összes elemét
`c.assign(n,elem)`	a `c` tárolóba behelyezi az `elem` `n` darab másolatát
`c.assign(beg, end)`	a `c` tárolóba behelyezi a `beg` és `end` iterátorokkal megadott tartományt

Hozzáférés a tároló elemeihez

Művelet	Hatás
`c.at(idx)`	Az `idx`-edik elemet téríti vissza (kivételt dob helytelen index esetén)
`c[idx]`	Az `idx`-edik elemet téríti vissza (nem végez semmiféle ellenőrzést)
`c.front()`	Az első elemet téríti vissza (nem végez semmiféle ellenőrzést)
`c.back()`	Az utolsó elemet téríti vissza (nem végez semmiféle ellenőrzést)

Beszúrás és törlés

Művelet	Hatás
`c.insert(pos,elem)`	Beszúrja a megadott pozícióra a megadott elemet
`c.insert(pos,n,elem)`	A megadott pozícióra `n` darab `elem` értékű elemet szúr be
`c.insert(pos,beg,end)`	A megadott pozícióra beszúrja a `[beg,end)` taromány elemeit
`c.push_back(elem)`	A végéhez fűzi az `elem` másolatát
`c.pop_back()`	Törli az utolsó elemet
`c.erase(pos)`	Törli a `pos` pozíciójú elemet
`c.erase(beg,end)`	Törli a megadott tartományból az összes elemet
`c.clear()`	Törli a tároló összes elemét

7.3.2. A string tároló

A `string` (karakterlánc) karakterek sorozata. Egy karakterlánc bármilyen karakterek sorozata lehet. A karakter egy grafikus szimbólum, a karakterkészlet pedig egy megfeleltetés a szimbólumok és a számértékek között. Többféle karakterkészlet létezik, a legismertebbek az ASCII és az UNICODE készletek. Az ASCII 8 bites kódokat tartalmaz, az UNICODE pedig 16 biteseket.

A C++ karakterlánc elméletileg tetszőleges típust képes karakterként használni. Azonban csak azokat a típusokat érdemes füzérként felfűzni, amelyeknek nincs saját felhasználói másoló művelete (értékadó operátor), mert csak ezeket lehet füzérként is megfelelő hatékonysággal kezelni.

Egy karaktertípus jellemzőit a hozzá tartozó `char_traits` osztály írja le. A standard könyvtár `string` tárolója a `basic_string` sablonosztálynak `char` típusra való példányosítása. Ezt szemlélteti az alábbi kódrészlet:

```
template <class Ch,

 class Tr= char_traits<Ch>,

 class A=allocator<Ch>

>

class basic_string{
...
};

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

A `basic_string` osztály három sablonparaméterrel rendelkezik. A `Ch` sablonparaméter a felfűzött elem típusa, a `Tr` paraméter a `Ch` karaktertípus jellemzőit leíró osztály, a harmadik paraméter a `Ch` típusnak megfelelő memóriafoglaló. A memóriafoglalókról részletes leírás található a [1] 758. oldal és a [2] 727. oldal könyvekben.

A string és a vector összehasonlítása

A `string` és a `vector` tárolók hasonlóan viselkednek. Mindkettő dinamikus tömbként van implementálva. A stringet felfoghatjuk olyan vectorként, amely karaktereket tartalmaz. Mindkét tárolóra használhatjuk a standard könyvtár algoritmusait. Az alapvető különbség a két tároló között a cél, amelynek érdekében létrehozták:

A `vector` tároló a tárolt elemek kezelésére biztosít hatékony műveleteket.

A `string` tároló műveletei nem a tárolt karakterekre összpontosítanak, hanem ezen elemeket mint egy egység kezelik, így ebben az esetben az egyik fő cél `string` objektumok gyors másolása. A gyakorlatban a string típus hivatkozásszámláló osztályként van implementálva.

Konstruktorok

Művelet	Hatás
`string s`	Üres karakterlánc
`string s(str)`	Az `s` karakterlánc, az `str` másolata
`string s(str, stridx)`	Az `s` karakterlánc az `str` másolta `stridx`-től kezdődően
`string s(str, stridx,strlen)`	Az `s` karakterlánc az `str` másolta `stridx`-től kezdődően, legfeljebb strlen hosszú
`string s(cstr)`	Az `s` karakterlánc a `cstr` C típusú füzérrel inicializálva
`string s(chars, chars_len)`	Az `s` karakterlánc a `cstr` C típusú füzér első `chars_len` karakterével inicializálva
`string s(num, c)`	Az `s` karakterlánc `num` darab `c` karakterrel inicializálva
`string s(beg, end)`	Az `s` karakterlánc a `[beg,end)` tartománnyal inicializálva

Kapcsolat a C típusú füzérekkel

A C++ nyelvben a karakterfüzér literálok nem `string` típusuak, hanem `const char *`. (Szabványos C-ben `char *`) Ennek ellenére az új string objektumok szoros kapcsolatban állnak a közönséges C sztringekkel (C-sztring a továbbiakban). Általában a string műveltenél (összehasonlítás, hozzáfűzés, beszúrás) használhatunk C-sztringeket is, mert automatikus konverzió van a `const char *` és a `string` típusok között, de ez a konverzió csak egyirányú *const char* $* \rightarrow$ *string*. A fordított átalakítást a `string` osztály `c_str()` metódusa biztosítja. A `string` osztály nem tulajdonít különleges jelentőséget a `'\0'` karakternek, ennek következtében ez a karakter úgyanúgy előfordulhat a `string` objektumban, mint bármely más karakter.

Háromféleképpen nyerhetjük ki a karakter típusú tömböt egy string objektumból:

1. `data()`

Visszatéríti az objektum által tartalmazott karaktertömböt, amely nem tartalmazza a lezáró `'\0'` karaktert.

2. `c_str()`

Visszatéríti az objektum által tartalmazott karaktertömböt, amely valódi C-sztring, `'\0'` karakterrel lezárva.

3. `copy()`

A `string` tartalmát bemásolja egy paraméterként átadott karaktertömbbe, amely nem tartalmazza a lezáró `'\0'` karaktert.

Vigyázzunk a `data()` és a `c_str()` használatára, amelyek egy karaktertömb mutatót térítenek vissza. Ennek a karaktertömbnek viszont a string objektum a tulajdonosa, ezért a visszaadott mutató egy konstans karaktertömb mutató, amelyen keresztül nem változtatható meg a karaktertömb tartalma. Ha módosítani szeretnénk a kinyert karaktertömböt, akkor ezt át kell másolni egy másik nem konstans karaktertömbbe. A következő kódrészlet ezt szemlélteti:

```
char buffer[ 100 ];
string s(”alma”);
strcpy(buffer, s.c_str());
//buffer feldolgozása
```

Méret és kapacitás

Egy string objektumnak háromféle mérete van. Vizsgáljuk meg, hogy melyiknek mi a pontos szerepe.

1. `size()` és `length()`

Visszatéríti a füzér aktuális karaktereinek számát.

2. `max_size()`

Visszatéríti a füzérben maximálisan tárolható karakterek számát. Ez általában az index típusának megfelelő maximális érték.

3. `capacity()`

Új helyfoglalás nélkül hány karakter tárolható a string objektumban.

Mivel egy új helyfoglalás nemcsak időigényes hanem érvényteleníti a string objektum aktuális iterátorait is, ezért ajánlott a `string` létrehozása után a helyfoglalást is úgy elvégezni, hogy lehetőleg ne legyen szükség újabb helyfoglalásra. A helyfoglalást a `reserve( méret)` tagfüggvénnyel végezhetjük.

Hozzáférés a füzér elemeihez

A string objektum karakterei írhatóak és olvashatóak is. A hozzáférés az index operátorral `[ ]` illetve az `at()` tagfüggvénnyel valósítható meg. Az első karakter a `0` indexű, az utolsó indexe pedig `length()-1`. Amíg az index operátor nem ellenőrzi az index érvényességét, addig az `at()` függvény ellenőrzést végez és érvénytelen index esetén `out_of_range` kivételt dob.

Összehasonlítások

Az összehasonlítási műveletek a megszokottak (`<, <=, <=, >, ==,!=`), az operandusok pedig `string` illetve C-sztring típusúak lehetnek. Az összehasonlítás eredménye egy logikai érték. A `string` osztály biztosít egy `compare` példány szintű tagfüggvényt is, amely a C `strcmp` függvényhez hasonlóan működik. Tekintsük erre a következő példát:

```
string s1(”alma”);
string s2(”almafa”);
string s3 (”almafa teteje”);

s1.compare(s2); // Negatív (<0) érték
s3.compare(s2);// Pozitív (>0) érték
s1.compare(”alma”);// Nulla (0)
```

Módosító műveletek

Értékadás

Ha egy füzérnek új értéket akarunk adni, ezt az értékadó operátorral, illetve az `assign` metódussal tehetjük meg. Az értékadó operátor jobb oldalán lehet egy másik `string` objektum, egy C-sztring vagy akár egy karakter is. Az `assign` használata akkor ajánlott, ha valamely string részfüzérét akarjuk értékül adni, ebben

az esetben egynél több paramétert kell átadni az értékadás elvégzése érdekében. Tekinstük a következő példákat:

```
string s1(”almafa”)
string s2;

s2 = s1;
s2 =”narancsfa”;
s2 =’c’;

s2.assign(s1);//ekvivalens s2=s1
s2.assign(s1,4,2);//”fa” részfüzér
s2.assign(”valami”);
```

Beszúrás

A beszúrás egyik sajátos esete a hozáfűzés. Ezt is háromféleképpen végezhetjük a += operátorral, az `append()` illetve a `push_back()` tagfüggvényekkel. Tekintsük erre a következő példát:

```
string s;
string s1(”almafa”);

s += s1;
s += ”narancsfa”;
s += ’c’;

s.append(s1);
s.append(s1, 3, 2);
s.append(”alma”);

s.push_back(’s’);
```

Az általános célú beszúró függvény az `insert()`, amelynek az `append()`-hez hasonlóan több túlterhelt formája is létezik. A leggyakrabban használt alaknak két paramétere van, az első paraméter a beszúrás pozícióját határozza meg, a második paraméter pedig a beszúrandó füzért, amely megadható `string` vagy C-sztring típusként is.

```
string s(”alfa”);
s.insert(2,”ma”); //s--> ”almafa”
```

Törlés

Az összes karakter törlését háromféleképpen végezhetjük:

```
string s;

s =’””;
s.clear();
s.erase();
```

Részfüzér törlése esetén meg kell adnunk a kezdőpozíciót és esetleg a törlendő karakterek számát. Amennyiben a második paraméter hiányzik, a füzér végéig fog törölni.

```
string s(”almafa”);
s.erase(4,2);//s-->”alma”
```

Részfüzérek és konkatenáció

A konkatenációt a + operátorral végezzük, a részfüzérek lekérdezését pedig a `substr()` tagfüggvénnyel. Ez utóbbi hívásakor meg kell adnunk a kezdőpozíciót és esetleg a részfüzér karaktereinek számát. Amennyiben a második paraméter hiányzik, a füzér végéig fog tartani a részfüzér.

```
string s(”almafa”);
cout<<s.substr(4)<<endl; //”fa”
```

Kimenő/Bemenő műveletek

Karakterfüzérekre is értelmezve vannak a megszokott extractor/inserter(>>,<<) műveletek. Ezek a műveletek a C-sztringek beolvasásához hasonlóan működnek. Az extractor művelettel nem lehet fehér karaktert tartalmazó füzért beolvasni, de létezik egy `getline()` függvény, amely végjellel lezárt sorok beolvasására alkalmas. Ez a függvény sorvég jelig, illetve fájlvég jelig olvas. Alapértelmezetten az újsor jelig olvas, de megadható egy sajátos sor elválasztó karakter is. Tekintsük a következő példákat:

```
string s;

while(getline(cin, s)){
...
}

vagy

while(getline(cin, s, ':')){
...
}
```

Keresés

A kereső függvények tagfüggvényként vannak értelmezve és a következőképpen jellemezhetők:

- Minden kereső függvény az első előfordulási pozíciót téríti vissza, amely egy `string::size_type` típusú érték.
- A függvény első argumentuma a keresett érték.
- A második argumentum opcionális és a keresés kezdőpozícióját határozza meg.
- A harmadik opcionális argumentumon keresztül megadható, hogy hány karaktert keressen a keresett értékből.

A keresőfüggvényeket a következő táblázat foglalja össze:

Művelet	Hatás
`find(str)`	Az `str` első előfordulási pozíciója
`rfind(str)`	Az `str` utolsó előfordulási pozíciója
`find_first_of("ab")`	Az 'a' vagy 'b' karakter első előfordulási pozíciója
`find_last_of("ab")`	Az 'a' vagy 'b' karakter utolsó előfordulási pozíciója
`find_first_not_of("ab")`	Az első 'a' és 'b'-től különböző karakter
`find_last_not_of("ab")`	Az utolsó 'a' és 'b'-től különböző karakter

Példák:

```
string s(”Havazik havazik hópelyhecskék hullnak”);

s.find(”zik”);//visszatérített érték: 4
s.find(”zik”, 10);//visszatérített érték: 12
s.rfind(”zik”);//visszatérített érték: 12

s.find_first_of(”ai”);//visszatérített érték: 1
s.find_last_of(”ai”);//visszatérített érték: 35

s.find_first_not_of(”ai”);//visszatérített érték: 0
s.find_last_not_of(”ai”);//visszatérített érték: 36
s.find(”xy”);//visszatérített érték: npos
```

Az npos érték

Ha a keresés sikertelen a visszatérített érték `string :: npos`. Tekintsük a következő példát:

```
string s;
string::size_type idx;//Vigyázat csak ez a típus használható

idx=s.find(”fa”);
if(idx == string::npos){
...
}
```

Példa

Olvassunk be a szabványos bemenetről sorokat. Bontsuk szavakra a bemenet minden egyes sorát tudva, hogy a szavak a következő szimbólumokkal vannak elválasztva: ’ ’, ’\t’, ’,’, ’.’, ’;’

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

main(){
   string line;
   string::size_type begIdx, endIdx;
   const string delims(" \t,.;");
```

```
    while(getline(cin, line)){
       begIdx = line.find_first_not_of(delims);
       while(begIdx != string::npos){
          endIdx = line.find_first_of(delims, begIdx );
          //ha a szo vege egyben a sor vege is
          if(endIdx == string::npos)
             endIdx = line.length();
          cout<<line.substr(begIdx, endIdx-begIdx+1)<<endl;
          begIdx = line.find_first_not_of(delims, endIdx);
       }
    }
    return 0;
}
```

7.4. Feladatok

1. Mit csinál a következő programrészlet?

```
vector<int> u(10, 100);
vector<int> v;
copy(u.begin(), u.end(),v.begin());
```

2. Javítsuk ki az előző programrészletet, úgy, hogy másolást végezzen a két dinamikus tömb között. Legalább két javítási lehetőség létezik. Implementáljuk mindkettőt, tárgyaljuk a megvalósítások előnyeit és hátrányait.

3. Készítsen szóelőfordulási statisztikát a szabványos bemenetől beolvasott szövegről. A szöveget szavanként olvassuk be. A statisztika elkészítéséhez használjon asszociatív tömböt (`map<string, int>`).

4. A szabványos bemenetről beolvasott szövegről készítsen egy tárgymutatót (index). A tárgymutató tartalmazza az összes, a szövegben előforduló, legalább 3 betűből álló szót és az előfordulási sorok sorszámát. A tárgymutató tárolásához használjunk asszociatív tömböt(`map< string, set<int> >`).

Például:

"cpp" — 2007/10/23 — 19:43 — page 128 — #128


```
   1. sor: ez egy példaszöveg, amelyből indexet készítünk
   2. sor: a példaszöveg több sorból áll, minden sor egy vagy
több mondatot is tartalmaz
   3. sor: ez a példaszöveg vége

   amelyből        1
   áll             2
   egy             1, 2
   indexet         1
   készítünk       1
   minden          2
   példaszöveg     1, 2, 3
   sor             2
   sorból          2
   több            2
   vagy            2
   vége            3
   tartalmaz       2
```

8. FEJEZET

ITERÁTOROK

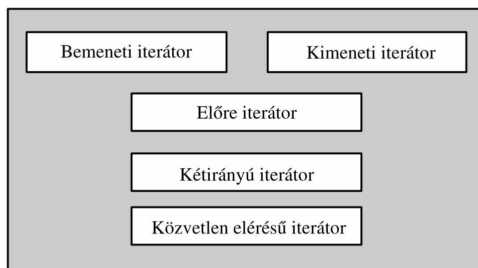
8.1. Bevezetés

Minden tároló meghatározza a saját bejárótípusát, gyakorlatilag a tárolók belső osztályok segítségével valósítják meg az iterátorokat. Létezik egy `iterator` nevű fejállomány, amely a különleges bejárókhoz, mint a fordított irányú bejáró (reverse iterator), tartalmaz definíciókat.

Az iterátor egy absztrakt fogalom. Alapvetően egy pozíciót határoz meg egy adathalmazban. Annak függvényében, hogy milyen műveleteket lehet a meghatározott pozícióval végezni, osztályozhatjuk a bejárókat különféle erősségű bejárókategóriákba. Vannak „igénytelen" algoritmusok, amelyek megelégszenek nagyon egyszerű bejárókkal, más algoritmusok pedig különleges képességű bejárókat igényelnek.

Az iterátorok hasonlítanak a mutatókhoz, akár általánosított vagy intelligens mutatóknak is nevezhetjük, hiszen tetszőleges belső ábrázolású tárolónak biztosítják a bejárhatóságát. Amíg a mutató kaphat NULL értéket, addig az iterátornak nem lehet NULL értéke. Iterátorból csak kétféle van: érvényes és érvénytelen.

- érvényes: `it != c.end()` és használható a `*it` operátorral (`c` egy tetszőleges tároló)
- érvénytelen:
  - nincs kezdőértéke
  - értéke `c.end()`-el egyenlő
  - törlés (részleges vagy teljes) vagy beszúrás következtében érvénytelenné vált



8.1. ábra. Iterátor kategóriák

Iterátor kategória	Képesség	Szolgáltató
Bemeneti iterátor	Előre olvas	istream
Kimeneti iterátor	Előre ír	ostream, inserter
Előre iterátor	Előre olvas és ír	
Kétirányú iterátor	Előre/hátra olvas és ír	list, set, multiset, map, multimap
Közvetlen eléresű iterátor	Bármilyen sorrendben olvas és ír	vector, deque, string, array

8.1. táblázat. Iterátorosztályok képességei

8.2. Iterátor kategóriák

A bejárók a mutatókhoz hasonlóan lehetnek konstans és nem konstans bejárók. Konstans bejárón keresztül nem lehet a mutatott objektumot megváltoztatni. Aszerint, hogy milyen műveleteket képesek hatékonyan végezni, a bejárókat öt osztályba soroljuk, amint ezt a 8.1. ábra is mutatja. Minden osztálynak megvannak a sajátos műveletei. Nemcsak a bejáró típusa, hanem a mutatott elem típusa is befolyásolja a végezhető műveleteket.

Csak közvetlen elérésű bejárókat növelhetünk, illetve csökkenthetünk tetszőleges egész értékkel. Két bejáró közötti távolság, a kimeneti bejáró kivételével, mindig kiszámítható. A 8.1. táblázatban összefoglaltuk a bejárókategóriákat, megadva a tárolókat is, amelyek biztosítják ezeket.

Kifejezés	Hatás
*iter	Az aktuális elem értéke (csak olvasás)
iter->tag	Hozzáférés az aktuális pozícióban levő elemhez
++iter	Következő elem, új pozíció visszatérítése
iter++	Következő elem, régi pozíció visszatérítése
iter1==iter2	Egyenlőség vizsgálata
iter1!=iter2	Különbözőség vizsgálata
TYPE(iter)	másoló konstruktor

8.2. táblázat. Bemeneti bejárók műveletei

8.2.1. Bemeneti iterátorok

Ezek az iterátorok csak szekvenciális feldolgozást tesznek lehetővé. A feldolgozás során minden elem csak egyszer érhető el olvasásra (Single Pass Iterator). Ezek az iterátorok tipikusan a bemenet feldogozására vannak szakosítva. Ugyanaz a bemeneti érték kétszer nem olvasható be. Az 8.2. táblázatban foglaltuk össze a bemeneti bejárók alapműveleteit a megfelelő magyarázatokkal együtt.

Algoritmus, amely ilyen erősségű bejárót igényel:

- `find`
- `for_each()`

8.2.2. Kimeneti iterátorok

Ezek az iterátorok a bemeneti iterátorokkal ellentétes műveleteket végeznek. A bemeneti bejárókhoz hasonlóan csak szekvenciális feldolgozást tesznek lehetővé. A feldolgozás során minden elem csak egyszer érhető el írásra (Single Pass Iterator). Úgy kell elképzelni a kimenetet, mint egy „fekete-lyuk" sorozatot, a $*iter = ertek$ művelet pedig beír egy értéket a soron következő fekete-lyukba. A szabványos kimenet írása ilyen erősségű iterátorral kezelhető.

Algoritmus, amely ilyen erősségű bejárót igényel:

- `copy`, a céltartomány iterátora

Kifejezés	Hatás
*iter=érték	Az aktuális pozíció írása
++iter	Következő elem, új pozíció visszatérítése
iter++	Következő elem, régi pozíció visszatérítése
TYPE(iter)	másoló konstruktor

8.3. táblázat. Kimeneti bejárók műveletei

8.2.3. Előre iterátorok

Ez a kategória a bemeneti és a kimeneti iterátorok tulajdonságait egyesíti. A bemeneti bejárók minden tulajdonsága érvényes rá, a kimeneti iterátorok tulajdonságainak pedig nagyrésze. Minden elem többször is elérhető az iterátoron keresztül (Multi Pass Iterator). Az alapvető különbség a kimeneti és az előre iterátor között az, hogy amíg a kimeneti bejárók esetében nem kell ellenőrizni a pozíció érvényességét, addig az előre iterátor esetében ez szükséges. Tekinstünk konkrét példákat erre:

1. példa

```
while(true){
   *pos = kifejezés;
   ++pos;
}
```

2. példa

```
while(pos != coll.end()){
   *pos = kifejezés;
   ++pos;
}
```

A `pos` egy iterátort jelent. Az 1. példa tökéletes kimeneti bejárókra, de hibás előre iterátorokra. A 2. példa pedig előre bejáróra helyes és lehetetlen kimeneti bejáróra. Kimeneti bejáróra nem értelmezett az összehasonlítás művelet, sőt a `coll.end()` sem értelmezhető.

Algoritmus, amely ilyen erősségű bejárót igényel: `replace`

```
template <class FwdIt, class T>
```

Kifejezés	Hatás
*iter	Hozzáférés az aktuális elemhez
iter->tag	Hozzáférés az aktuális pozícióban levő elemhez
++iter	Következő elem, új pozíció visszatérítése
iter++	Következő elem, régi pozíció visszatérítése
iter1==iter2	Egyenlőség vizsgálata
iter1!=iter2	Különbözőség vizsgálata
iter1 = iter2	Értékadás
TYPE()	Implicit konstruktor
TYPE(iter)	másoló konstruktor

8.4. táblázat. Előre bejárók műveletei

```
    void replace(FwdIt first, FwdIt last, const T& oldvalue, const
T& newvalue){
       for(;first != last; ++first)
          if(*first == oldvalue)
             *first = newvalue;
    }
```

8.2.4. Kétirányú iterátorok

Olyan előre bejáró, amely lehetővé teszi a pozíció visszafele léptetését. Így gyakorlatilag az előre bejáró műveleteit kiegészíti az `iter--` és a `--iter` műveletekkel.

Algoritmus, amely ilyen erősségű bejárót igényel: reverse_copy

```
template <class BiIt, class OutIt>
void reverse_copy(BiIt first, BiIt last, OutIt res){
   while(last != first){
      --last;
      *res = *last;
      ++res;
   }
   return res;
}
```

Kifejezés	Hatás
iter[n]	Az n indexű elem elérése
iter+=n	n pozícióval előre (ha n negatív, akkor hátra)
iter-=n	n pozícióval hátra (ha n negatív, akkor előre)
iter+n	n-el előbbre levő elem visszatérítése
n+iter	n-el előbbre levő elem visszatérítése
iter-n	n-el hátrább levő elem visszatérítése
iter1-iter2	A két iterátor távolsága
iter1<iter2	ellenőrzi, hogy iter1 előbbre van-e, mint iter2
iter1>iter2	ellenőrzi, hogy iter1 hátrább van-e, mint iter2
iter1<=iter2	
iter1>=iter2	

8.5. táblázat. Közvetlen elérésű bejárók műveletei

8.2.5. Közvetlen elérésű iterátorok

A közvetlen elérésű iterátor olyan kétirányú iterátor, amely a kétirányú mozgás mellett közvetlen elérést is biztosít. Ilyen erősségű iterátort biztosítanak a `vector`, a `deque`, a `string` (`wstring`) és a közönséges C tömb is. Ezek az iterátorok a pointeraritmetikában használatos műveleteket biztosítják. Az 8.5. táblázat tartalmazza azokat a műveleteket, amelyeket pluszban biztosít a kétirányú bejáróhoz képest.

Algoritmusok, amelyek ilyen erősségű bejárót igényelnek: `sort()`, `stable_sort()`, `make_heap`, `prev_permutation`, `next_permutation`

8.3. Iterátor adapterek

8.3.1. Vissza iterátorok

A vissza iterátorok olyan iterátor adapterek, amelyek megfordítják az inkrementálás és a dekrementálás szerepét. Ha az algoritmusoknak vissza-iterátorokat adunk át, akkor fordított sorrendben dolgozzák fel az iterátorok által megadott tartományt. Példaként tekinsük a következő programot:

```
#include <iostream>
#include <list>
#include <algorithm>

using namespace std;
void doublevalue (int& elem)
{
   cout<<elem<<" ";elem *= 2; cout<<elem<<endl;
}
int main()
{
   list<int> coll;
   for (int i=1; i<=10; ++i) {
      coll.push_back(i);
   }

   for_each (coll.rbegin(), coll.rend(), doublevalue);
   return 0;
}
```

8.3.2. Beszúró iterátorok

A beszúró iterátorok olyan iterátor adapterek, amelyek az értékadó műveletet beszúró műveletté alakítják. Gyakorlatilag a „felülíró üzemmódot" „beszúró üzemmóddá" alakítják. A beszúró iterátor a kimeneti iterátor kategória egy sajátos formája. Tekintsünk egy algoritmust, amely kimeneti iterátort igényel és nézzük meg, hogyan változik meg az algoritmus viselkedése, ha a kimeneti iterátor helyett beszúró iterátort használunk.

```
1. template <class InIt, class OutIt>
2. OutIt copy(InIt first1, InIt last1, OutIt first2){
3.    while(first1 != last1){
4.       *first2 = *first1;
5.       ++first1;
6.       ++first2;
7.    }
8.    return first2;
9. }
```

Név	Osztály	Hívott függvény	Létrehozás
Hátra beszúró	back_insert_iterator	push_back(érték)	back_inserter(cont)
Előre beszúró	front_insert_iterator	push_front(érték)	front_inserter(cont)
Beszúró	insert_iterator	insert(poz, érték)	inserter(cont,poz)

8.6. táblázat. Beszúró iterátorok

A fenti kódrészlet 4. sora tartalmaz egy értékadást, amelyben a kimeneti tartomány valamely pozíciója egy új értéket kap. Ha az algoritmus hívásakor a céltartomány esetében beszúró iterátort adunk meg, akkor ezt az értékadást fogja beszúrássá alakítani. Felmerül a kérdés, hogy milyen értelme van ennek az átalakításnak. Gondoljunk csak arra, amikor kijelentettük, hogy az algoritmusok sebességre vannak optimalizálva, ezért semmilyen ellenőrzést nem végeznek a megadott tartományokkal kapcsolatosan, ez teljesen a hívó felelőssége. Ha viszont a céltartományt nem felülírásos, hanem beszúrásos üzemmódban kezeljük, akkor ennek mérete automatikusan fog változni a beszúrások következtében, nem okozva semmiféle futásidejű hibát.

A C++ standard könyvtár háromféle beszúró iterátort biztosít: hátra beszúró, előre beszúró és beszúró. Ez a három iterátor csak beszúrási pozícióban különbözik és mindeniket azzal a konténerrel kell inicializálni, amelybe a beszúrásokat végzi. Az 8.6. táblázat összefoglalja a standard könyvtár által biztosított beszúró iterátorokat.

Természetesen a konténereknek biztosítaniuk kell a megfelelő tagfüggvényt, amelyet a beszúró iterátor meghív. Ennek következtében a hátra beszúrók csak `vector`, `deque`, `list` és `string` tárolókra használhatók, előre beszúró iterátorok pedig `deque` és `list` tárolóra. Tekintsünk egy példát az iterátor használatára:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main(){
```

```
    vector<int> v;
    vector<int>::iterator it1;
    int x[] ={ 10, 20, 30 };
    copy(&x[0], &x[3], back_inserter(v));
    for(it1= v.begin(); it1 != v.end(); ++it1)
       cout << *it1 <<" ";
    cout << endl;
    return 0;
}
```

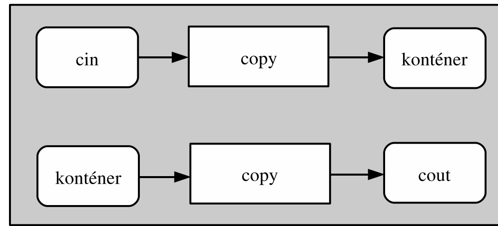
Az beszúró iterátorokat lehet használni úgy is, hogy létrehozzuk az iterátor típusú változót és rajta keresztül szúrjuk be az elemeket. Ez a kevésbé kényelmes használata a beszúró iterátoroknak.

```
vector<int> v;
back_insert_iterator< vector<int> > it(v);
*it = 1; it++;
*it = 2; it++;
```

Az előre beszúrókat a hátra beszúrókhoz hasonlóan használjuk, vigyázva arra, hogy csak a `deque` és a `list` tárolóval használható.

Egy általános beszúrót két paraméterrel inicializálunk: a konténer és a konténeren belüli pozíció. Az asszociatív tárolók esetében a pozíciót természetesen nem fogja figyelembe venni, hiszen ezen tárolók esetében a pozíciót a beszúrandó érték határozza meg. Mivel minden tároló biztosít `insert` műveletet, ezért ez a beszúró iterátor mindenikre használható. Tekintsünk egy példát ennek használatára:

```
#include <iostream>
#include <set>
#include <list>
#include <algorithm>

using namespace std;
int main(){
   list< int > l;
   set < int > s;
   int i;
   for(i=0; i<10; i++)
```



8.2. ábra. Adatfolyamok és algoritmusok

```
      l.push_front(i);
   copy(l.begin(), l.end(), inserter(s, s.end()));
   set< int > :: iterator it;
   for(it = s.begin(); it != s.end(); ++it)
      cout << *it<<" ";
   cout << endl;
   return 0;
}
```

8.3.3. Adatfolyam (stream) iterátorok

Az adatfolyam adapterek segítségével az algoritmusok a bemenetüket illetve a kimenetüket adatfolyamokhoz köthetik, ahogyan az 8.2. ábra is szemlélteti.

A működési alapelvük megegyezik a beszúró iterátorokéval, az értékadást adatfolyam műveletté alakítják. Létezik kimenő adatfolyam iterátor és bemenő adatfolyam iterátor.

Kimenő adatfolyam iterátorok

Ez az iterátor az értékadást a kimenő adatfolyamba való beszúrássá alakítja. Az 8.7. táblázat a kimenő adatfolyam iterátorokkal végezhető műveleteket tartalmazza.

Példa a használatra:

```
int x[ ] ={1, 2, 3, 4};
vector v(&x[0], &x[4]);
copy(v.begin(), v.end(), ostream_iterator<int>(cout,”,”));
```

Kifejezés	Hatás
ostream_iterator<T>(ostream)	Iterátor létrehozása
ostream_iterator<T>(ostream, delim)	Létrehozás, a bemenet elválasztó egysége: delim
*iter = érték	Az érték kiírása az iterátor éltal meghatározott pozícióba
++iter	Visszatéríti az iterátort
iter++	Visszatéríti az iterátort

8.7. táblázat. Kimenő adatfolyam iterátorok műveletei

Bemenő adatfolyam iterátorok

Az ilyen típusú iterátorok segítségével az algoritmusok bemenetüket közvetlenül adatfolyamokból olvashatják. Létrehozáskor az iterátort az adatfolyammal kell inicializálni. Az algoritmusok tartományokon dolgoznak, amelyeket iterátorok segítségével kell megadni. Következtetésképpen szükség van egy adatfolyam-vég iterátorra (end of stream iterator) is, amelyet implicit konstruktorral adunk meg. Ha a beolvasás sikertelen, az iterátor értéke adatfolyam-vég iterátor lesz. Az adatfolyam iterátornak akkor van érvényes értéke, ha különbözik az adatfolyam-vég iterátortól.

A bemenő adatfolyam iterátor konstruktora megnyitja az adatfolyamot és kiolvassa az első értéket. Az 8.8. táblázat tartalmazza a bemenő adatfolyam iterátorok műveleteit.

Tekintsünk két példát a bemenő iterátorok használatára. Az első példában egy bemenő iterátor segítségével fogjuk a bemenetet olvasni, a másodikban pedig a `copy` algoritmus segíségével.

Példa 1

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

int main(){
   istream_iterator<int> it(cin);
   istream_iterator<int> itend;
```

Kifejezés	Hatás
istream_iterator<T>()	Létrehoz egy adatfolyam-vég iterátort
istream_iterator<T>(istream)	Létrehoz egy adatfolyam iterátort a megadott adatfolyamra
*iter	Visszatéríti az aktuális értéket
iter->tag	Visszatéríti az aktuális elem (ha ez összetett) valamely tagját
++iter	Olvassa a következő elemet és visszaadja ennek pozícióját
iter++	Olvassa a következő elemet és visszaadja az aktuális pozíciót
iter1==iter2	Egyenlőségvizsgálat
iter1!=iter2	Különbözőség vizsgálat

8.8. táblázat. Bemenő adatfolyam iterátorok műveletei

```
   while(it != itend){
      cout<<"Olvasott ertek: "<<*it<<endl;
      it++;
   }
   return 0;
}
```

Példa 2

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

int main(){
   vector<int> v;
   copy(istream_iterator<int>(cin),
      istream_iterator<int>(),back_inserter(v));
   copy(v.begin(), v.end(),
      ostream_iterator<int>(cout," "));
   return 0;
}
```

8.4. Iterátorok jellemzői

Minden iterátor kategóriának a C++ standard könyvtár egy iterátor tagot biztosít:

```
struct output_iterator_tag{};
struct input_iterator_tag{};
struct forward_iterator_tag:public input_iterator_tag{};
struct bidirectional_iterator:public forward_iterator_tag{};
struct random_access_iterator:public bidirectional_iterator_tag{};
```

Megfigyelhető, hogy az iterátor típusok származtatási viszonyban vannak. Ennek következtében minden kétirányú bejáró egyben előre bejáró is, és minden közvetlen hozzzáférésű bejáró egyben kétirányú bejáró is.

Ha generikus kódot írunk, nem elég ismernünk csak az iterátor kategóriáját, sokszor egyéb információra is szükségünk lehet. Példának okáért szükség lehet az iterátor által hivatkozott elem típusára is. Ezért a C++ standard könyvtár egy sablon iterátor jellemzők struktúrát biztosít. Ez a struktúra tartalmazza az iterátorra vonatkozó összes releváns információt és az iterátorosztályok interfészként használják:

```
template< class T>
struct iterator_traits{
   typedef typename T::value_type         value_type;
   typedef typename T::difference_type    difference_type;
   typedef typename T::iterator_category  iterator_category;
   typedef typename T::pointer            pointer;
   typedef typename T::reference          reference;
};
```

A fenti sablonban a T az iterátor típusát jelenti. Ha például a tömb osztálynak szeretnénk iterátort megadni, amely a közönséges mutató típusnak megfelelően fog működni, akkor ezt a következően tehetnénk:

```
template <class T>
class CArray{
   T * data;
   int size;
public:
```

```
    ...
    typedef T* iterator;

    typedef T                            value_type;
    typedef ptrdiff_t                    difference_type;
    typedef random_access_iterator_tag   iterator_category;
    typedef T*                           pointer
    typedef T&                           reference;

    iterator begin(){ return data; }
    iterator end()  { return data+size; }
    ...
};
```

8.5. Saját iterátorok készítése: Az egyszeresen láncolt lista

Készítsünk egy egyszeresen láncolt lista tárolót, amely az STL tárolóihoz hasonlóan működik. A következő kódrészlet egy lehetséges implementáció részleteit mutatja be.

Tervezési problémák:

1. Hogyan ábrázoljuk a listaelem(`Node`) és a lista(`List`) kapcsolatát?
2. Hogyan ábrázoljuk a lista és a listán belüli pozíció(`iterator`) kapcsolatát?
3. Hogyan tud együttműködni a lista tárolónk a standard könyvtár algoritmusaival?

Válaszok:

1. A felhasználónak nem kell ismernie a lista belső struktúráját. Ennek következtében a listaelem típus teljesen rejtett kell legyen, hiszen a felhasználó a lista osztály nyilvános metódusai segítségével fogja a listát kezelni. A listaelem tulajdonképpen egy adataggregátum, ezért struktúra segítségével ábrázoljuk, amelynek egyetlen művelete a konstruktor. A listaelemet a lista osztály privát belső

típusaként ábrázoljuk, ezzel biztosítottuk az elérhetetlenségét a lista osztályon kívüli kód számára.

2. A listán belüli pozíciót az `iterator` osztállyal ábrázoljuk. Ezt az osztályt is belső osztályként definiáljuk, de ezt már a lista osztály nyilvános részében, hiszen osztályon kívüli kód számára is elérhetőnek kell lennie. Például:

```
List<int> l;
List<int>::iterator it;
```

Mivel a listaiterátor egy előre iterátor, ezért az előre iterátor típusból származtatjuk. (`std::iterator<std::forward_iterator_tag, T>`)

3. Az STL algoritmusai bizonyos típusok jelenlétét feltételezik. Ezeket a típusokat (`pointer`, `reference`, `difference`, `value_type`, `iterator_category`) a listaiterátor osztály nyilvános részében helyezzük el. A `difference` típust listaiterátorra nem adjuk meg.

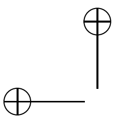
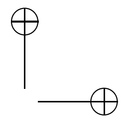
```
#include <iterator>

template <class T>
class List
{

   struct Node
   {
      Node(const T& x,Node* y = 0):m_data(x),m_next(y){}
      T m_data;
      Node* m_next;
   };

   Node* m_head;

public:

   class iterator
   : public std::iterator<std::forward_iterator_tag, T>
   {
      Node* m_rep;
   public:
      typedef T   value_type;
      typedef std::forward_iterator_tag iterator_category;
      typedef T * pointer;
```

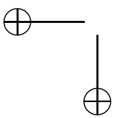
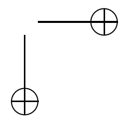
```
        typedef T & reference;

        iterator(Node* x=0):m_rep(x){}
        iterator(const iterator& x):m_rep(x.m_rep) {}
        iterator& operator=(const iterator& x)
        {
           m_rep=x.m_rep; return *this;
        }
        iterator& operator++()
        {
           m_rep = m_rep->m_next; return *this;
        }
        iterator operator++(int)
        {
           iterator tmp(*this); m_rep =m_rep->m_next;
           return tmp;
        }
        reference operator*() const { return m_rep->m_data; }
        pointer operator->() const { return m_rep; }
        bool operator==(const iterator& x) const
        {
           return m_rep == x.m_rep;
        }
        bool operator!=(const iterator& x) const
        {
           return m_rep != x.m_rep;
        }
     };
   ...

};
```

8.6. Feladatok

1. Egészítse ki a saját dinamikus tömb osztályát egy bejárást megvalósító osztállyal. Belső osztályos megvalósítást kell adni, amelyet ugyanúgy használhatunk, mint a szabványos könyvtár `vector` tárolójának bejáróját.

2. Készítse el a láncolt lista adatstruktúrát iterátorral együtt, amely a standard könyvtár `list` tárolójához hasonlóan működik.

9. FEJEZET

SABLONOK

9.1. Bevezetés

A sablonok közvetlenül támogatják az általánosított (generikus) programozást, azaz a típusoknak paraméterként való használatát. Sablonok segítségével az adatstruktúrák megadhatóak típusfüggetlen módon. Egy típusfüggetlen módon megadott adatstruktúrát úgy tekinthetünk, mint egy olyan általános komponens, amelyből bármilyen sajátos típusú komponens előállítható. Gyakorlatilag nevezhetjük komponens-gyárnak is, amelyet csak egyszer kell megírni és utána különböző típusokra példányosítható.

A C++ standard könyvtár szinte teljes egészében sablonokból álló könyvtár. A sablonok nemcsak a típusfüggetlen adatrstruktúrák megadására teremtenek lehetőséget, hanem lehetővé teszik a viselkedésmód paraméterezhetőségét is, illetve a kód optimalizálását.

9.2. Függvénysablonok

9.2.1. Függvénysablonok definíciója

Egy függvénysablon egy függvénycsaládot jelent. A függvénycsalád formailag egy függvényhez hasonlít, amelyben bizonyos elemek nincsenek meghatározva. Ezek a nem meghatározott elemek lesznek a függvénysablon paramétrei, és gyakorlatilag ezek különböztetik meg a függvényeket a függvénysablonektől.

Tekintsük a következő függvénysablont:

```
template <typename T>
const T& max(const T& a, const T& b){
   return a < b ? b : a;
}
```

A max függvény egy függvénycsaládot határoz meg, amely két érték maximumát téríti vissza. A sablonparamétert T-vel jelöltük és a typename kulcsszóval vezettük be. A típusparaméter azonosítója bármi lehet, de T a megegyezés szerinti jelölés. A típusparaméter egy tetszőleges típust jelent, amelyet a hívó specifikál a függvénysablon használatakor. Természetesen a gyakorlatban csak azokra a típusokra lehetséges a függvénysablon használata, amelyekre értelmezett a "<" operátor, hiszen ezt az operátort használtuk az összehasonlításra.

Történelmi okok miatt, a typename kulcsszó helyett a class kulcsszó is használható típusnév megadására. A sablonok bevezetésekor a C++ nyelvben csak a class kulcsszót lehetett használni típusnév megadására. Ezért a fenti függvénysablon megadható a következőképpen is:

```
template <class T>
const T& max(const T& a, const T& b){
   return a < b ? b : a;
}
```

9.2.2. Függvénysablonok használata

Tételezzük fel, hogy a függvénysablonunkat egy max.h nevű fájlban helyeztük el. A következő program a függvénysablon használatát szemlélteti.

```
#include <iostream>
#include <string>
#include "max.h"
using namespace std;

int main(){
   string s1("alma");
   string s2("szilva");
   cout<<"max(100, 30)="<<::max(100, 30)<<endl;
   cout<<"max("<<s1<<","<<s2<<")="<<::max(s1,s2)<<endl;
   return 0;
}
```

A fenti kódrészletben a `max` függvényt kétszer hívtuk. Először `string` típusra, majd egész típusú paraméterekre. Mindkét hívás esetén a `max` függvényt a hatókör (`::`) operátorral prefixeltük. Ez garantálja, hogy a mi függvényünk fog meghívódni a globális névtérből. Azért van szükség a `::max(...)` alakú hívásra, mert a szabványos könyvtárban is van egy max függvény, amely a szabványos névtérhez tartozik (std::max). Ha nem használtunk volna hatókör operátort a függvénynév előtt, akkor ez kétértelműséghez vezetett volna, amely fordítási hibában nyílvánul meg.

Ha csak a sablont fordítjuk, a fordító nem generál semmiféle kódot. Kódgeneráláshoz szükség van a sablon használatára is. A sablonban megadott kód úgy használódik, hogy először kigenerálódik a hívásnak megfelelő, konkrét típusokat tartalmazó kód, majd ezt fogja a fordító lefordítani. Amennyiben többféle típusú paraméterre hívjuk ugyanazt a függvénysablont, több konkrét függvény fog kigenerálódni. A mi programunkban pontosan két konkrét függvény jön létre, az egyiknek string típusú paraméterei vannak, a másiknak int típusú paraméterei:

```
const string& max(const string& a, const string& b){...}
const int& max(const int& a, const int& b){...}
```

A sablonok gyakorlatilag kétszer fordítódnak le:

1. Példányosítás nélkül, a sablon fordításakor csak szintaktikai helyességvizsgálat történik.
2. Példányosításkor a sablonból kigenerált konkrét kód kerül ellenőrzésre.

Függvények esetében a sablonparamétereknek nem lehet implicit értéke.

9.2.3. Argumentumok levezetése

Függvénysablonok hívásakor a sablonparamétereket a függvény aktuális paramétereiből fogja kikövetkeztetni a fordító. Ha például a `max` függvénynek két egész számot adunk át, akkor a `T=int` behelyettesítés történik a függvénysablonban. A `max(2, 2.4)` hívás viszont hibaüzenethez vezet, mert nem történik automatikus típuskonverzió. A `max(2, 2.4)` függvényhívást kétféleképpen alakíthatjuk át úgy, hogy elfogadja a fordító:

– Explicit típusátalakítást végzünk azért, hogy a paramétrek típusa találjon `max(static_cast<double>(2), 2.4)`

– Híváskor nem hagyatkozunk a típusparaméter levezetésére, hanem megadjuk azt `max<double>(2, 2.4)`

Vannak esetek, amikor a sablonparaméterek nem vezethetőek le a függvény aktuális paramétereiből. Ilyen esetben híváskor kötelező megadni az aktuális paraméterek mellett a sablonparamétereket is. Tekintsünk erre egy példát:

```
template<typename RT, typename T1, typename T2>
RT max(const T1& a, const T2& b){
   ...
}
```

A fenti példában a `T1` és a `T2` azon két paraméter típusa, amelyekből ki kell választanunk a maximumot. Az `RT` viszont a visszatérített érték típusa lesz. Tekintsük a következő függvényhívást:

```
max(3, 4);
```

Ebben az esetben $T1 = T2 = int$, viszont nincs, ahonnan kikövetkeztetnie a fordítónak az `RT` típusparamétert. Ilyen estben kötelező híváskor megadni a típusparamétereket is: `max<int,int,int>(3,4)`.

9.2.4. Függvénysablonok túlterhelése

Közönséges függvényekhez hasonlóan a függvénysablonok is túlterhelhetők. A túlterhelés azonos nevű, különböző paraméterezésű függvényeket jelent. Gyakorlatilag egy plusz terhet ró a fordítóra, amelynek az aktuális paraméterekből ki kell következtetnie, hogy pontosan melyik függvényt kell meghívni a több ugyanolyan nevű függvényhalmazból. Túlterhelés esetében még sablonparaméterek nélkül is kétértelműség alakulhat ki, ami fordítási hibához vezethet. A függvénysablonok még összetettebbé teszik ezt a döntési problémát.

Tekintsünk egy példát olyan függvény túlterhelésre, amelyben sablonfüggvény is szerepel.

```
1. #include <iostream>
2. using namespace std;
3.
```

```
 4. const int& max(const int& a, const int& b){
 5.    return a <b ? b : a;
 6. }
 7.
 8. template <typename T>
 9. const T& max(const T& a, const T& b){
10.   return a<b ? b : a;
11.}
12.
12. template <typename T>
13. const T& max(const T& a, const T& b, const T& c){
14.    return max(max(a,b), c);
15. }
16.
17. int main(){
18.    cout<<::max(1,2,3)<<endl;
19.    cout<<::max(1.2, 2.3)<<endl;
20.    cout<<::max('a','b')<<endl;
21.    cout<<::max<double>(2,3)<<endl;
22.    cout<<::max(7,'A')<<endl;
23.    return 0;
24. }
```

A 18. sorban a háromparaméteres függvénysablon hívódik meg.

A 19. sorban a kétparaméteres függvénysablon hívódik, a `<double>` sablonparaméter az argumentumok típusából vezetődik le.

A 20. sorban a kétparaméteres függvénysablon hívódik, a `<char>` sablonparaméter az argumentumok típusából vezetődik le.

A 21. sorban a kétparaméteres függvénysablon hívódik, nincs szükség a sablonparaméter levezetésére, hiszen megadtuk azt.

A 22. sorban az első nem függvénysablon fog meghívódni, amelynek két egész típusú argumentuma van.

Az egyetlen nem egyértelműnek tűnő hívás az utolsó (22. sor). Ebben az esetben a két függvényargumentum különböző típusú. A char viszont automatikusan átalakul egész típussá. A fordítónak két lehetőség közül kell választania: (i) meghívhatja a `max(const int&, const int&)` függvényt vagy (ii) meghívhatja a `max<T>(const T&, const T&)` függvényt `T=int` típusparaméterre. A fordító a nem függvénysablont fogja választani.

Lehetséges üres sablonparaméter lista megadása is. Ez azt jelenti, hogy csak függvénysablon hívható, de ezek a sablonparaméterek az argumentumokból vezethetők le.

```
max<>(3, 4);
```

9.3. Osztálysablonok

9.3.1. Osztálysablonok deklarációja

A függvényekhez hasonlóan az osztályok esetében is használhatunk típusparamétereket, ezeket osztálysablonoknak nevezzük. A standard könyvtár minden egyes tárolója sablonosztállyal van megvalósítva. Következtetésképpen a standard könyvtár tárolói típusfüggetlenek. A szabványos osztályoktól eltérően, ahol az osztalydeklarációt egy fejállományban helyeztük el, a definíciót pedig egy .cpp kiterjesztésű fájlban, a osztálysablonokat egyetlen fejállományban fogjuk elhelyezni. Ellenkező esetben összeszerkesztési (linker) hibát fogunk kapni.

Most pedig elkészítjük az egyik legegyszerűbb adatstruktúrát osztálysablon segítségével. Az egyszerűség kedvéért a vermet a `vector<T>` tároló segítségével fogjuk megadni. Így nem kell foglalkoznunk a memóriakezeléssel, másoló konstruktorral illetve értékadó operátorral.

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

template <typename T>
class Stack{
private:
   vector<T> elements;
public:
   void push(const T&);
   T pop();
   T top() const;
   bool empty() const{
      return elements.empty();
   }
};
```

```
template <typename T>
void Stack<T>::push(const T& a){
   elements.push_back(a);
}

template <typename T>
T Stack<T>::pop(){
   if(elements.empty())
      throw out_of_range("Empty stack");
   T e = elements.back();
   elements.pop_back();
   return e;
}

template <typename T>
T Stack<T>::top() const{
   if(elements.empty())
   throw out_of_range("Empty stack");
   return elements.back();
}
```

9.3.2. Osztálysablonok használata

A következő példaprogram a osztálysablon használatát szemlélteti. Amíg a függvénysablonok esetében nem volt kötelező megadni a sablonparamétereket, addig az osztálysablonok példányainak létrehozásakor mindig kötelező megadni. Ez tökéletesen logikus, hiszen ebben az esetben nincs, ahonnan kikövetkeztetnie a sablonparamétereket a fordítónak.

```
#include "Stack.h"
#include <iostream>
using namespace std;

int main(){
   Stack<int> v;
   int i;
   for(i=0; i<10; ++i)
      v.push(i);
   while(! v.empty())
```

```
        cout<<v.pop()<<endl;
    cout<<endl;
    return 0;
}
```

Csak azokat a függvényeket fogja kigenerálni a fordító, amelyeket ténylegesen meg is hívunk. Tehát vigyázzunk, hogy osztálysablon tesztelésekor ennek minden egyes függvényét hívjuk meg. Ajánlatos legalább egy primitív és egy felhasználói típusra tesztelni az osztálysablont.

Típusnév bevezetésével kényelmesebbé tehetjük az osztálysablonok használatát. Például:

```
typedef Stack<int> IntStack;
typedef Stack<string> stringStack;
Stack<float*> floatPtrStack;
Stack< Stack<int> > intStackStack;
```

Figyelem: Ha két sablonparamétert egymásba ágyazunk, a két bezáró > jel közé tegyünk szünetet, különben fordítási hibát kapunk.

9.3.3. Sablonspecializáció

Egy osztálysablon specializálható bizonyos sablonparaméterekre. A specializáció célja, hogy optimalizáljuk az osztálysablon megvalósítását egy adott típusra. Ha az osztálysablon rendellenesen viselkedik egy adott típusra, ez specializációval kikerülhető. Az osztálysablon specializációja minden tagfüggvény specializációját jelenti. (Lehetőség van külön a tagfüggvények specializációjára is, ha viszont élünk a lehetőséggel, továbbá nem lesz specializálható az osztály.)

A specializáció a következő szintaxissal valósítható meg:

```
template<>
class Stack< string >{
...
};
```

A fenti kódrészlet a `Stack` osztály specializációját adja meg `string` típusra. A tagfüggvények megadását a következőképpen kell végezni:

```
void Stack<string>::push(const string & elem){
   elements.push_back(elem);
}
```

Figyeljük meg, hogy a függvény előtt nem szerepel a `template<typename T>` sor, amelynek az a szerepe, hogy jelezze a függvény osztálysablonhoz való tartozását. A fenti függvény már közönséges függvényhez hasonlít. A következőkben megadjuk a teljes kódot, amely a Stack osztályt string típusra specializálja.

```
#include <string>
#include <stdexcept>
#include "Stack.h"
#include <deque>
using namespace std;

template<>
class Stack< string >{
private:
   deque<string> elements;
public:
   void push(const string&);
   string pop();
   string top() const;
   bool empty() const{
      return elements.empty();
   }
};

void Stack<string> :: push(const string& elem){
   elements.push_back(elem);
}

string Stack<string> :: pop(){
   if(elements.empty())
      throw out_of_range("Empty stack");
   string elem = elements.back();
   elements.pop_back();
   return elem;
}
```

```
string Stack<string> :: top() const{
   if(elements.empty())
      throw out_of_range("Empty stack");
   return elements.back();
}
```

9.3.4. Részleges sablonspecializáció

Az osztálysablonok részlegesen is specializálhatók. Ez akkor lehtséges, ha az osztálynak több sablonparamétere van és ezek közül csak néhányat specializálunk. Például tekintsük a következő két sablonparaméterrel rendelkező osztályt:

```
template< typename T1, typename T2>
class MyClass{
   ...
};
```

A következő kódrészlet a második sablonparamétert specializálja `int` típusra:

```
template< typename T>
class MyClass<T, int>{
   ...
};
```

9.3.5. Implicit értékű sablonparaméterek

Amíg a függvénysablonok esetében a sablonparamétereknek nem lehet implicit értékük, addig osztályok esetében ez lehetséges. Ezeket az értékeket implicit sablonértékeknek nevezzük. Az előző `Stack` osztálysablonban az elemek tárolására a `vector` konténert használtuk. Mi lenne, ha a felhasználóra bíznánk a tároló megválasztását? Természetesen a kevésbé igényes felhasználóknak lehetővé tennénk, hogy ne kelljen megadniuk ezt a másodlagos paramétert. Ezt a problémát az implicit sablonparaméterek oldják meg. A következő `GStack.h` fejállományban elhelyezett kód ezt szemlélteti:

```
#include <vector>
#include <stdexcept>
```

```
using namespace std;

template <typename T, typename CONT = vector<T> >
class GStack{
private:
   CONT elements;
public:
   void push(const T& elem);
   T pop();
   T top() const;
   bool empty() const{
      return elements.empty();
   }
};

template <typename T, typename CONT>
void GStack<T,CONT> :: push(const T& elem){
   elements.push_back(elem);
}

template <typename T, typename CONT>
T GStack<T,CONT> :: pop(){
   if(elements.empty())
      throw out_of_range("Empty Stack");
   T elem = elements.back();
   elements.pop_back();
   return elem;
}

template <typename T, typename CONT>
T GStack<T,CONT> :: top() const {
   if(elements.empty())
      throw out_of_range("Empty Stack");
   return elements.back();
}
```

Tekintsük a következő példát használatra:

```
#include "GStack.h"
#include <deque>
#include <iostream>
using namespace std;
```

```
int main(){
   GStack<int> s1;
   GStack<double, deque<double> > s2;
   int i;
   for(i=0; i<10; ++i){
      s1.push(i);
      s2.push(i);
   }
   while(! s1.empty())
      cout<<s1.pop()<<" ";
   cout<<endl;
   while(! s2.empty())
      cout<<s2.pop()<<" ";
   cout<<endl;
   return 0;
}
```

9.4. Közönséges típusú sablonparaméterek

A sablonparaméterek lehetnek közönséges értékek is. Az eddigi sablonjaink segítségével nyitott kód megadására van lehetőségünk. Az általános kód nyitva hagyta a típus megadásának a lehetőségét. A közönséges értékek sablonparaméterként való használatakor nem egy típus marad nyitott, hanem egy érték.

9.4.1. Közönséges érték, mint sablonparaméter osztálysablonban

Felhasználva a közönséges paraméter lehetőséget definiálhatunk olyan `Stack` osztálysablont, amely rögzített méretű tömbben tárolja az elemeket. A tömb méretét egy közönséges értékű sablonparaméteren keresztül adjuk meg. A verem ezen megadási módjával elkerülhetőek a memóriakezeléssel járó problémák.

```
template <typename T, int MAXSIZE=100>
class Stack{
private:
   T elements[ MAXSIZE];
```

```
    int numElements;
public:
...
};
```

9.4.2. Közönséges érték, mint sablonparaméter függvénysablonban

Most pedig lássunk egy példát függvénysablonban használt közönséges értékű sablonparaméterre. Ebben a példában az első sablonparaméter típust meghatározó paraméter, a második sablonparaméter pedig egy közönséges értékparaméter.

```
template <typename T, int VAL>
T addValue(const T&){ return x+VAL; }
```

Tételezzük fel, hogy `src` és `dest` két tároló, és a `dest` tároló legalább ugyanannyi elemet tartalmaz, mint az `src` tároló, akkor a következő kódrészlet az `src` elemeit fogja megnövelve átheleyezni a `dest` tárolóba:

```
    transform(src.begin(), src.end(), dest.begin(),
addValue<int,5>);
```

Megjegyzés: Nem használhatunk valós típusokat és objektumokat sablonparaméterként.

9.5. Statikus és dinamikus polimorfizmus

A polimorfizmus sokféleséget, sokoldalúságot jelent. Objektumorientált programozásban ugyanazon kódrészletnek a különféle futásidejű viselkedését jelenti. Ezért is illettük a dinamikus jelzővel, mert futásidőben derül ki a pontos viselkedésmód. A C++ nyelvben a polimorfizmus örökléssel és virtuális függvényekkel valósítható meg. Ez a fajta polimorfizmus közös ősosztály meglétét igényli a polimorfikus viselkedés megnyilvánulásához.

A C++ nyelv a dinamikus polimorfizmus mellett, a sablonok segítségével, lehetőséget teremt egy másik fajta polimorfizmusra, a statikus polimorfizmusra. Ebben az esetben bizonyos műveletek a sablonpéldányosítás során megadott típusnak megfelelően fognak viselkedni. Azért nevezzük statikus polimorfizmusnak, mert feloldása fordítási időben történik.

Dinamikus polimorfizmus:

- elegánsan kezeli a heterogén tárolókat
- a végrehajtható kód mérete kisebb.
- korlátos mert csak egy közös ősosztályból származtatott osztályok esetében érvényesül
- dinamikus mert futásidőben nyilvánul meg, futásidejű kötést igényel

Statikus polimorfizmus:

- korlátlan mert azon típusok, amelyekre alkalmazható, nincsenek semmiféle közös ősosztályhoz kötve
- statikus mert a kötés fordításidőben történik, fordításidejű kötést igényel
- primitív típusú tárolók is ugyanolyan egyszerűen megadhatók, mint az objektumokat tartalmazó tárolók. Dinamikus polimorfizmus esetében a primitív típusokhoz csomagoló osztályokat kell készítenünk ahhoz, hogy ezeket behelyezhessük tárolókba. (Ld. Java)
- a végrehajtható kód gyorsabb, hiszen nincsen szükség futásidejű csatolásokra.
- a statikus polimorfizmus típusbiztosabb, hiszen már fordítási időben a fordító rendelkezésére áll a kötéshez szükséges minden információ.

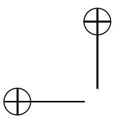
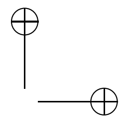
9.6. Feladatok

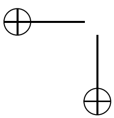
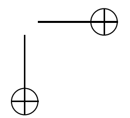
1. Készítsen egy `rendez` függvénysablont. Implementálja a gyorsrendezést, lehetőleg ennek iteratív változatát.

2. Készítsen egy `keres` függvénysablont. Implementálja a szekvenciális keresést. A függvénysablonnak működnie kell minden tárolóval.

3. Készítsen egy `keres` függvénysablont. Implementálja a bináris keresést. A függvénysablonnak működnie kell a C-stílusú tömbbel és a vector tárolóval.

4. Készítsen egy hasító tábla sablonosztályt. Tanulmányozza a standar könyvtár `hash_map` tárolóját.

10. FEJEZET

ADATFOLYAMOK

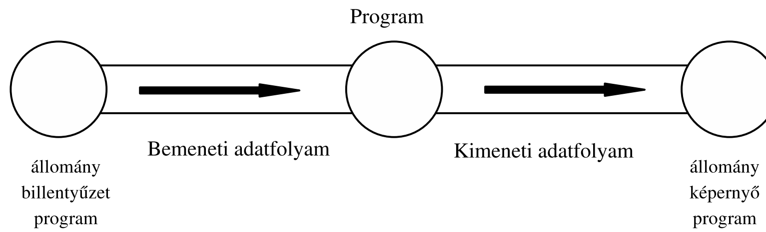
Minden programozási nyelv tanulásakor először a kimeneti/bemeneti műveleteket tanuljuk meg, hiszen ezek nélkül használhatatlanok lennének programjaink. A C++ nyelv tanulásakor is megmutattuk, hogy milyen operátorokkal lehet írni a kimenetre, illetve olvasni a bemenetet. A C++ nyelvben a Ki/Be műveletek a leghaladóbb C++ fogalmak (osztályok, származtatás, függvények túlterhelése, virtuális függvények, többszörös örökítés és sablonok) segítségével vannak megvalósítva, ezért odáztuk el ezeknek a részletes ismertetését. Tehát, ahhoz, hogy megértsük a Ki/Be műveleteket, nagyon jól kell ismernünk a C++ nyelvet.

10.1. Bevezetés

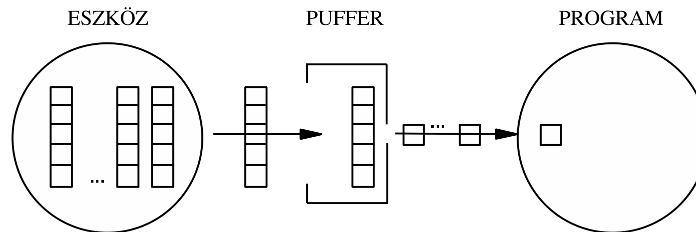
A C++ nyelvben a kimeneti és bemeneti műveletek adatfolyamokkal vannak megvalósítva. Egy adatfolyam gyakorlatilag egy bájtsorozat, amelyből bemenet esetében a program olvas és kimenet esetében pedig ír. Szöveges információk esetében a bájtsorozatot karaktersorozatnak tekinthetjük. Más típusú adatok esetében, a bájtok csoportosíthatóak az adott típusnak megfelelően. Egy bemeneti adatfolyamnak két végpontja van, az egyik végpontja az adatforrás, amely az adatokat szolgáltatja, a másik végpontja az adatnyelő, a mi esetünkben a program, amely az adatokat beolvassa ("lenyeli”). Ezt az 10.1. ábra szemlélteti.

Az adatfolyam tulajdonképpen egy csatorna, amely segítségével az adatok eljutnak az adatforrástól az adatnyelőig. Az adatfolyamokat azért vezették be, hogy egységesen tudják kezelni bármely adatforráshoz, illetve adatnyelőhöz rendelt bájtsorozatot. Az 10.1. ábrán is láthatjuk, hogy a bemeneti adatfolyam esetében a csatorna egyik végpontja (forrás) lehet fájl, billentyűzet vagy akár egy másik

“cpp” — 2007/10/23 — 19:43 — page 164 — #164



10.1. ábra. Kimeneti és bemeneti adatfolyamok



10.2. ábra. A puffer szerepe egy bemeneti adatfolyam esetében

program is. A kimeneti adatfolyam esetében maga a program lesz az adatforrás és az adatfolyam másik végpontja lehet fájl, képernyő vagy akár egy másik program.

A csatorna irányítottságát mindig a program felől tekintjük. Ha például egy program kimenetét egy másik program bemenetével kötjük össze, akkor ugyanaz a csatorna az elsőnek kimeneti, míg a másodiknak bemeneti csatorna lesz.

Összefoglalva, az adatfolyamok lehetővé tették, hogy a C++ nyelvből a bemeneti és a kimeneti műveleteket a fizikai eszköztől függetlenül tudjuk végezni, vagyis attól függetlenül, hogy fájlból vagy billentyűzetről olvassuk, programjaink ugyanazon műveletek segítségével olvasnak. A kimenetet végző műveletekről is ugyanezeket állíthatjuk.

A bemeneti/kimeneti műveletek gyorsítására puffert használnak. A puffer egy köztes memóriaterület, ahol a program és az eszköz között, átvitelre szánt adatok tárolódnak. A merevlemez eszköz esetében az adatok legalább 512 bájtos blokkokban íródnak/olvasódnak. Ezzel ellentétben programjaink általában bájtonként dolgozzák fel az adatokat. A puffer biztosítja az optimális sebességű átvitelt eszközök és programok között. Az 10.2. ábra ezt a folyamatot szemlélteti.

Kimeneti eszközhöz rendelt puffer esetében értelmezett a "puffer ürítése" művelet, amelynek hatására a puffer tartalma fizikailag is kiíródik az eszközre.

10.2. Adatfolyam osztályok

A két legfontosabb adatfolyam osztály a kimeneti `ostream` és a bemeneti `istream` adatfolyamok osztálya. Mindkét osztály egy-egy osztálysablonból előállított osztály. Az `ostream` osztály a `basic_ostream` osztály `char` típusra való példányosítása. Hasonlóan az `istream` a `basic_istream` `char` típusra való példányosítása. Az alapvető adatfolyam osztályok hierarchiáját az 10.3. ábra szemlélteti. A hierarchiában szereplő osztályok szerepe a következő:

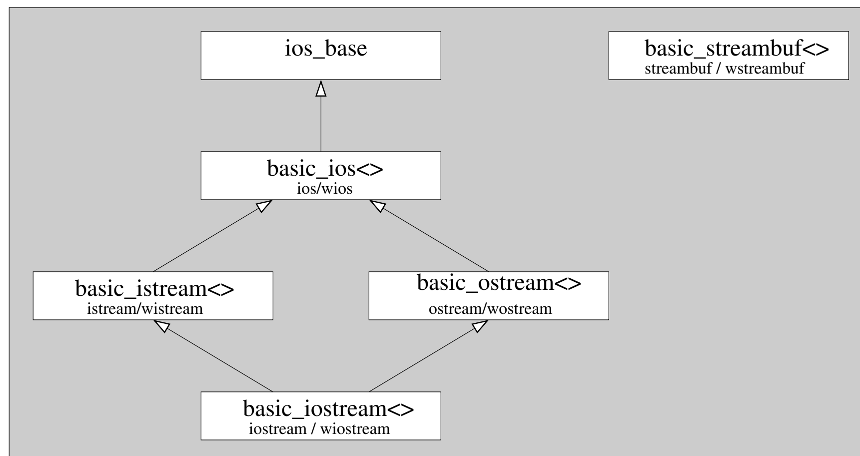
- `ios_base`: Az összes adatfolyamra érvényes tulajdonságokat tartalmazza, függetlenül a karaktertípustól és az ennek megfelelő karakterjellemzőktől. Főképp adatfolyam állapotára vonatkozó és formázásához szükséges adatokat és műveleteket tartalmaz.

- `basic_ios<>`: Azokat a közös adatokat és műveleteket tartalmazza, amelyek függenek a karakter típusától és a karakterjellemzőktől. Tartalmazza az adatfolyamhoz tartozó puffer definícióját is.

- `basic_istream<>` és `basic_ostream<>`: Alapvető bemeneti és kimeneti osztályok, amelyek a karaktertípussal és a karakterjellemzőkkel paraméterezhetők.

- `basic_iostream<>`: Olyan osztályok megadására teremt lehetőséget, amelyek írást és olvasást is biztosítanak, azaz kétirńyú adatfolyamok.

- `basic_streambuf<>`: A puffer típusának megadását szolgáló osztály.



10.3. ábra. Alapvető adatfolyam osztályok

A standard könyvtár tartalmazza a szabványos Ki/Be csatornáknak megfelelő bemeneti/kimeneti adatfolyam objektumokat. Ezek az objektumok `cin, cout, cerr, clog`. Az első három rendre a szabványos C `stdin`, `stdout`, `stderr` megfelelője. A `clog` objektumnak nincs megfelelője szabványos C nyelvben, viszont ennek a csatornának és a `cerr` csatornának megegyezik az adatnyelője, azzal a különbséggel, hogy amíg a `cerr` nem pufferelt, addig a `clog` pufferelt. A 10.4. ábra ezeket a csatornaobjektumokat szemlélteti:

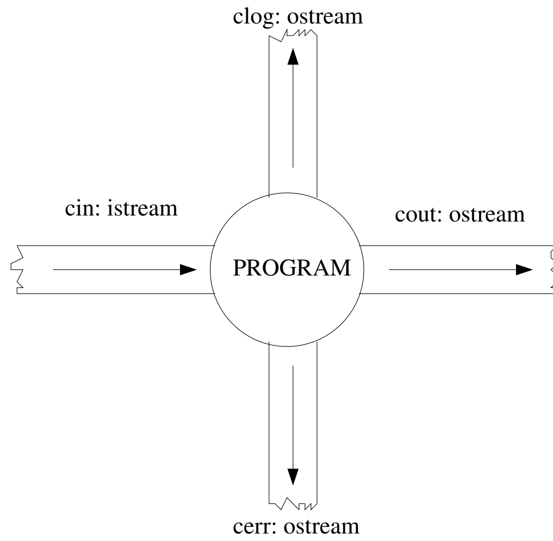
10.2.1. Adatfolyamok állapota

Az adatfolyam állapota a Ki/Be műveletek sikerességének jelzésére szolgál. Sikertelen művelet esetében a sikertelenség okát is tartalmazza.

Állapotjellemző állandók

Az állapotjelzésre az `ios_base` osztály tartalmazza az 10.1. táblázatban `iostate` típusú kapcsolókat.

Ha a `goodbit` be van kapcsolva, azt jelenti, hogy a többi kapcsoló nincs bekapcsolva. A `failbit` bekapcsolt állapota csak az I/O műveletek sikerességére



10.4. ábra. Adatfolyam objektumok

Állandó	Jelentés
`goodbit`	Minden rendben.
`eofbit`	Adatfolyamvég
`failbit`	Hiba. Sikertelen Ki/Be művelet.
`badbit`	Végzetes hiba. Használhatatlan adatfolyam.

10.1. táblázat. *`iostate`* típusú kapcsolók

Tagfüggvény	Jelentés
`good()`	`true` - ha a `goodbit` be van kapcsolva
`eof()`	`true` - ha az `eofbit` be van kapcsolva
`fail()`	`true` - ha a `failbit` vagy a `badbit` be van kapcsolva
`bad()`	`true` - ha a `badbit` be van kapcsolva
`rdstate()`	az állapotjelzőket téríti vissza
`clear()`	törli az összes állapotjelző bitet
`clear(state)`	törli az állapotjelzőket és beállítja a state-re
`setstate(state)`	beállítja a paraméterben megadott állapotjelzőket

10.2. táblázat. Adatfolyam állpotát lekérdező/beállító tagfüggvények

utal. Ettől eltekintve az adatfolyam nem sérült és használható állapotban van. Ez a kapcsoló akkor kerül bekapcsolt állapotba, ha például egy numerikus érték beolvasását kérjük és az adatfolyam következő karaktere egy betű.

A `badbit` bekapcsolt állapota komoly hibát jelez. Ilyenkor az adatfolyam használhatatlanná vált, megsérült.

Adatfolyam állapotát lekérdező tagfüggvények

Az adatfolyam állapotát az 10.2. táblázatban összefoglalt tagfüggvényekkel lehet lekérdezni, illetve beállítani.

Az 10.2. táblázat első négy függvénye logikai típust térít vissza. Vegyük észre, hogy a `fail()` függvény nemcsak a `failbit` állapotát, hanem a `badbit` állapotát is jelzi. Ezt kényelmi szempontok miatt valósították meg így. Bármilyen jellegű hiba is adódott, elégséges egyetlen függvényt, a `fail()` függvényt meghívni.

A `clear()` tagfüggvény törli az öoszes hibajelző bitet. Ha a paraméteres, túlterhelt változatát hívjuk meg a függvénynek, az adatfolyamnak beállítja a paraméterül kapott állapotot.

Ha az adatfolyam `failbit` hibajelző bitje be van kapcsolva, az adatfolyammal semmiféle műveletet nem lehet végezni egészen addig, amíg ki nem kapcsoljuk ezt a hibajelző bitet. Tehát egy sikertelen beolvasás egészen addig leállítja az olvasási folyamatot, amíg ki nem mozdítjuk ebből az állapotból. Tekintsünk egy tipikus hibát, amely a `failbit` beállítását erdeményezi:

```
list <int> a
while(!cin.eof()){
   int x;
   cin>>x;
   a.push_back(x);
}
```

A fenti kódrészlettel az a baj, hogy amikor a bemenet utolsó egységét beolvassuk, az `eofbit` még nem állítódik be. Ez csak akkor következik be, ha az utolsó utáni nem létező elemet akarjuk olvasni. Mivel a `cin>>in` eredményét nem ellenőriztük, ha ez sikertelen volt, az `x` változó előző beolvasásból származó értékét fogjuk a listába behelyezni. Ez azt eredményezi, hogy a bemenet utolsó egysége kétszer kerül be a listába.

10.2.2. A `cout` adatfolyam

Minden adatfolyam egy bájtsorozatnak tekinthető. Az `ostream` osztályban 8 bites entitások alkotják az adatfolyamot. A `wostream` osztály már széles karaktereket használ, így 16 bites egységek alkotják az adatfolyamot. Ha egy egész számot akarunk a szabványos kimenetre (`cout`) írni, akkor a megfelő 4 bájton ábrázolt egész szám szöveges formáját kell kiírnunk. Vagyis a 123456 egész szám esetében ez 6 bájt kiírását jelenti a kimenetre és nem a bináris ábrázolásnak megfelelő 4 bájtnak. Következtetésképpen az `ostream` osztály egyik legfontosabb feladata a numerikus típusok karaktersorozatokká alakítása.

A kimenetre az `inserter (<<)` operátorral bármilyen primitív típusú adatot írhatunk. Ezt az `<<` operátor túlterhelésével oldja meg a C++ nyelv, vagyis a következő operátorfüggvény a C++ nyelv összes primitív típusára túl van terhelve. Az operátor használható C-sztringekre és pointerekre is (`char*, void*`).

`ostream& operator<<(T);` ahol $T \in char,\ short,\ int,\ float,\ double$

Az operátorfüggvény egy `ostream&` típust térít vissza. Ennek következményeként az operátor láncoltan is használható. Tekintsük példaként a következő kódrészlet végrehajtását:

Feladat: Készítsünk programot, amely két vagy több fájlnevet kap argumentumként és az első fájl végéhez fűzi az összes többi tartalmát.

```
int a=10;
double b=12.3;
cout<<"a="<<a<<", b="<<b;
```

1. `cout<<"a="; ---> cout.operator<<("a")`, visszatérít egy referenciát a `cout` objektumra
2. `cout<<a; --> cout.operator<<(a)`, visszatérít egy referenciát a `cout` objektumra
3. `cout<<", b="; ---> cout.operator<<(", b=")`, visszatérít egy referenciát a `cout` objektumra
4. `cout<<b; --> cout.operator<<(b)`, visszatérít egy referenciát a `cout` objektumra

A kimeneti adatfolyamokat bájtonként is lehet írni. Ez a `put( char)` tagfüggvénnyel valósítható meg.

10.2.3. Felhasználói típusok írása

Az inserter operátor egy bináris operátor, amelyet a következőképpen használunk:

```
stream<<objektum
```

A C++ nyelv szabályai szerint ez kétféleképpen értelmezhető:

```
1. stream.operator<<(objektum)
2. operator<<(stream, objektum)
```

A primitív típusok az 1. lehetőséggel élnek. A második lehetőséggel a beépített és felhasználói típusok élnek, hiszen az adatfolyam osztályok zártak, nem bővíthetők. A második lehetőség egy globális függvény megadását jelenti. Mivel a kiíráshoz szükséges az objektum privát tagjainak az elérése is, ez is kétféleképpen oldható meg:

– adattag lekérdező függvényekkel

– az inserter függvényt barát függvényként fogjuk megadni az objektum osztályára nézve

Most pedig példákat adunk a kétféle megadási módra.
1. eset

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

class Complex{
private:
   double re, im;
public:
   Complex(double _re=0, double _im = 0): re(_re), im(_im){}
   double Re() const { return re; }
   double Im() const { return im; }
};

ostream& operator<<(ostream& os, const Complex& z){
   os<<z.Re()<<"+"<<z.Im()<<"i";
   return os;
}

int main(){
   Complex z1, z2(1,2);
   cout<<z1<<endl<<z2<<endl;
   return 0;
}
```

2. eset

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

class Complex{
private:
   double re, im;
public:
   Complex(double _re=0, double _im = 0): re(_re), im(_im){}
   friend ostream& operator<<(ostream& os, const Complex& z);
};
```

```
ostream& operator<<(ostream& os, const Complex& z){
   os<<z.re<<"+"<<z.im<<"i";
   return os;
}

int main(){
   Complex z1, z2(1,2);
   cout<<z1<<endl<<z2<<endl;
   return 0;
}
```

Amíg az első megadási módnak az az előnye, hogy az osztály módosítása nélkül megadhatjuk a példányok kiíratását végző függvényt, addig a második megadási mód gyorsabb kiíratást eredményez, mert barát függvényként hozzáférése van az osztály privát adattagjaihoz.

10.2.4. A `cin` adatfolyam

A `cin` adatfolyam is a többi adatfolyamhoz hasonlóan egy bájtsorozat. A szabványos bemenetet normális esetben a billentyűzettel állítjuk elő. Lehetőségünk van különböző típusú adatok beolvasására a szabványos bemenetről, ez pedig úgy valósul meg, hogy az istream osztály biztosítja a karaktersorozatok bármilyen primitív típussá alakítását.

A `cin` adatfolyamot a következőképpen használjuk:

```
cin >> valtozo;
```

A változónak van egy lefoglalt memóriaterülete, amelyben eltárolásra kerül a beolvasott érték. Ha például egész változóba helyezzük el az `123456` értéket, akkor ez a hatszámjegyű szám az egészeknek megfelelő négy bájtos bináris ábrázolásmódban kerül eltárolásra. Ezt az `istream` osztály végzi.

A fenti beolvasási mód használható bármilyen primitív típusú változóra. Ez az extractor(>>) operátor túlterhelésével van megvalósítva. Az operátor használható C-sztringekre és pointerekre is (`char*, void*`).

`istream& operator>>(T);` ahol $T \in char, short, int, float, double$

Az operátorfüggvény egy `istream&` típust térít vissza. Ennek következményeként az operátor láncoltan is használható, az inserter operátorhoz hasonlóan.

A bemeneti adatfolyamokat bájtonként is lehet olvasni. Ez a `get( char&)` tagfüggvénnyel valósítható meg.

10.2.5. Felhasználói típusok olvasása

Az extractor operátorokat az inserter operátorokhoz hasonlóan implementáljuk. Objektum olvasásakor itt is két lehetőségünk van: egyszerű függvényként vagy barátként implementáljuk az operátorfüggvényt. A következőkben példát adunk mindkét esetre:

1. eset

```
istream& operator>>(istream& is, Complex& z){
   double a, b;
   is>>a>>b;
   z=Complex(a,b);
   return is;
}

int main(){
   Complex z1, z2;
   cin>>z1>>z2;
   cout<<z1<<endl<<z2<<endl;
   return 0;
}
```

2. eset

```
class Complex{
...
   friend istream& operator>>(istream& is, Complex& z);
};
...
istream& operator>>(istream& is, Complex& z){
   is>>z.re>>z.im;
   return is;
```

Manipulátor	Osztály	Jelentés
`flush`	`basic_ostream`	Puffer ürítése
`endl`	`basic_ostream`	Újsor $'\backslash n'$ kiírása a pufferbe (+pufferürítés)
`end`	`basic_ostream`	C-sztring lezáró karakter kiírása a pufferbe($'\backslash 0'$)
`ws`	`basic_istream`	olvas elhanyagolva a fehér karaktereket

10.3. táblázat. Az *<istream>* és *<ostream>* fejállományokban definiált manipulátorok

```
}
...
int main(){
   Complex z1, z2;
   cin>>z1>>z2;
   cout<<z1<<endl<<z2<<endl;
   return 0;
}
```

10.3. Manipulátorok

A manipulátorok olyan függvények, amelyek módosítják az adatfolyam állapotát. A legfontosabb manipulátorokat az 10.3. táblázat tartalmazza.

A formázás is manipulátorok segítségével történik, amelyet a következő részben tárgyalunk. Most pedig nézzük meg, hogy hogyan működnek a manipulátorok. A manipulátorok az inserter és extractor operátoroknak átadott argumentumok. Például az `ostream` osztály manipulátorai a következő operátortúlterheléssel vannak megvalósítva.:

```
ostream& ostream::operator<<(ostream& (*op) (ostream&)){
   return (*op)(*this);
}
```

Az operátorfüggvénynek átadott `op` argumentum egy olyan függvénymutató, amelynek egy `ostream&` típusú paramétre van és egy `ostream&` típust térít vissza. Például az `endl` manipulátor a következőképpen van megvalósítva:

```
ostream& endl (ostream& os){
   os.put('\n');
   os.flush();
   return os;
}
```

A manipulátorokat egyszerűen beszúrjuk az adatfolyamba:

```
cout<<endl;
```

A C++ standard könyvtár nemcsak argumentum nélkül hívható manipulátorokat definiál, hanem argumentumokkal megadhatókat is. Az ilyen manipulátorok megvalósítása C++ implementáció-függő. Az argumentumokkal hívható manipulátorok esetében az `<iomanip>` fejállományt kell használni.

10.4. Formázás

Ebben a részben a formázási kapcsolókat fogjuk bemutatni. A formázási kapcsolókkal a kiírásnál használt pontosságot (tizedesek száma), a kitöltő karaktert, számrendszert és egyéb hasznos dolgot lehet beállítani.

A formázási kapcsolók lekérdezéséhez, illetve beállításához tagfüggvények állnak rendelkezésünkre. Ezeket a formátum beállító/lekérdező tagfüggvényeket a 10.4. táblázatban foglaltuk össze.

Bizonyos kapcsolók csoportot alkotnak. Például ilyen csoportot alkotnak a számrendszer alapjára vonatkozó kapcsolók. A formázási kapcsolók az `ios_base` osztályban vannak definiálva. A továbbiakban ezeket ismertetjük:

```
skipws// üres helyek átlépése a bemeneten

left //mezőigazítás, feltöltés az érték után
right//feltöltés az érték előtt
internal// feltöltés előjel és érték között

boolalpha//a true és false megjelenítése

dec//számrendszer alapja, 10 (decimális)
```

Tagfüggvény	Jelentés
`setf(kapcsolók)`	Csak a paraméterként megadott kapcsolókat állítja, visszatérítve a beállítás előttieket
`setf(kapcsolók,maszk)`	Csak a paraméterként megadott és a maszkkal azonosított kapcsolókat állítja, visszatérítve a beállítás előttieket
`unsetf(kapcsolók)`	Törli a kapcsolókat
`flags()`	Visszatéríti a kapcsolókat
`flags(kapcsolók)`	Beállítja a kapcsolókat, visszatérítve a beállítás előttieket
`copyfmt(stream)`	A megadott adatfolyam összes formázási kapcsolóját másolja

10.4. táblázat. Formázási tagfüggvények

`hex`//16 (hexadecimális)
`oct`//8(oktális)

`scientific`//lebegőpontos jelölés: d.ddddddEdd
`fixed`//dddd.dd

`showbase`//kiíráskor oktálisok elé 0, hexadecimálisok elé 0x
`showpoint`//a záró nullák kiíratása
`showpos`//+ a pozitív egészek elé
`uppercase`//'E', 'X' alkalmazása ('e', 'x' helyett)

`adjustfield`//mezőigazítással kapcsolatos jelző
`basefield`//egész számrendszer alapjával kapcsolatos jelző
`floatfield`//lebegőpontos kimenettel kapcsolatos jelző

`unitbuf`// minden kimenet után a puffer ürítése

Példák formázási kapcsolók használatára

1. Numerikus adatok kiíratása hexadecimális számrendszerben

```
cout.unsetf(ios::dec);
cout.setf(ios::hex);
cout<<256<<endl; // Kimenet: 100
cout.unsetf(ios::hex);
cout.setf(ios::dec);
```

Nagyon sok formázási kapcsolóhoz létezik manipulátor is, amelyek használata jóval egyszerűbb a kapcsolókénál. A számrendszerekre három manipulátor van definiálva. Például:

```
int v = 256;
cout<<v<<"***"<<hex<<v<<"***"<<oct<<v<<endl;

Eredmény: 256***100***400
```

A következőkben bemutatjuk a különböző formázási feladatokat ellátó manipulátorokat.

10.4.1. Logikai értékek formázása

Alapértelmezetten, a logikai értékek kiíratásakor numerikus kimenetet kapunk, `1` igaz esetében és `0` hamis esetében. Ha szöveges `true/false` értékeket szertnénk, akkor a `boolalpha` kapcsolót kell használnunk. Alapértelmezetten ez a kapcsoló nincs bekapcsolva, ezért a logikai értékek kiíratása `1/0` értékekkel történik. A kapcsoló használatát két manipulátor segíti:

`ios::boolalpha` :bekapcsolja a boolalpha kapcsolót
`ios::nboolalpha`: törli a boolalpha kapcsolót

```
cout<<boolalpha<<(1==1)<<endl; // kimenet: true
cout<<nboolalpha<<(1==1)<<endl;//kimenet: 1
```

10.4.2. Mező hossza, kitöltési karakter és igazítás

`setw(val)`: a mező hossza `val` értéket kap
`setfill(c)`: beállítja a kitöltő karaktert `c` -re
`left`: balra igazítás
`right`: jobbra igazítas
`internal`: balra igazítja az előjelet és jobbra igazítja az értéket

Például a -42 értéket, `6` helyre, a háromféle igazítás az 10.5. táblázatban megadott módon végzi:

Igazítás	Eredmény
left	-42
right	-42
internal	- 42

10.5. táblázat. Igazítás-példák

A következő kódrészlet az igazításokat szemlélteti:

```
#include <iostream>
#include <iomanip>
using namespace std;
main(){
   int x = -42;
   cout<<setw(6)<<left <<x<<endl;
   cout<<setw(6)<<right <<x<<endl;
   cout<<setw(6)<<internal<<x<<endl;
   return 0;
}
```

10.4.3. A plusz előjel és nagybetűk

A plusz előjel megjelentését a `showpos` és a `noshowpos` manipulátorok szabályozzák. Az előző kikényszeríti, míg a második nem engedélyezi a plusz előjel megjelenítését.

Az `uppercase` és `nouppercase` manipulátorok a numerikus értékek esetében nagybetűs, illetve kisbetűs megjelenítést kényszerítenek ki. Például az `uppercase` esetében a hexadecimális számok nagybetűkkel lesznek írva (`A-F`), a valós számok "scientific" megjelenítése esetében 1.2E-7 lesz 1.2e-7 helyett.

10.4.4. Számrendszerek

Egész számok írása és olvasása három számrendszerben történhet: 10, 16, 8. Alapértelmezetten a 10-es számrendszer használatos. Ezeknek megfelelő manipulátorok: `dec, hex, oct`. Tekintsük a következő példát:

```
cin>>hex>>v1; //Hexa beolvasás
```

Manipulátor	Jelentés
`showpoint`	Tizedes pont használata
`noshowpoint`	Nem használ tizedes pontot
`setprecision(val)`	Pontosság értéke `val` lesz
`fixed`	Tizedespontos megjelenítés
`scientific`	Mantissza/karakterisztika megjelenítés

10.6. táblázat. Lebegőpontos számok megjelenítése

```
cin>>v2;//Ez is hexa beolvasás
cout<<v1<<" "<<v2<<endl;//Decimális kiíratás
```

Ha a fenti kódrészlet bemenete: `ff 12` a kimenete: `255 18` lesz.

10.4.5. Lebegőpontos számok

Lebegőpontos számok megjelenítését az 10.6. táblázatban foglalt manipulátorok végzik.

A lebegőpontos számok megjelenítésére használt manipulátorokra tekintsük a következő példát:

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

main(){
   float f1 = 421;
   float f2 = 0.123456789;
   cout<<setprecision(2)<<fixed<<f1<<endl;
   cout<<setprecision(2)<<fixed<<f2<<endl;
   cout<<setprecision(6)<<fixed<<f1<<endl;
   cout<<setprecision(6)<<fixed<<f2<<endl;
   cout<<setprecision(2)<<scientific<<f1<<endl;
   cout<<setprecision(2)<<scientific<<f2<<endl;
   cout<<setprecision(6)<<scientific<<f1<<endl;
   cout<<setprecision(6)<<scientific<<f2<<endl;
   return 0;
}
```

Kimenet:

```
421.00
0.12
421.000000
0.123457
4.21e+02
1.23e-01
4.210000e+02
1.234568e-01
```

10.5. Fájlfolyamok

Fájlokhoz kapcsolt adatfolyamokat az `ifstream`, `ofstream` illetve az `fstream` osztályok segítségével hozhatunk létre. Ezek alapvetően egybájtos adatfolyamok. Széles karaktereket tartalmazó fájlokhoz a `wifstream`, `wofstream` illetve `wfstream` használata ajánlott. Amíg az `ifsteam` és az `ofstream` csak bemeneti illetve csak kimeneti fájlfolyam típus, addig az `fstream` egy kétirányú fájlfolyam kezelését biztosító osztály.

Egy fájlt olvasásra az `ifstream` osztály egy példányának létrehozásával nyithatunk meg. Például:

```
ifstream infile(”in.txt”);
```

Hasonlóképpen az írásra megnyitás egy `ofstream` példány létrehozását igényli. A megnyitás sikerességét a következőképpen ellenőrizhetjük:

```
if(! infile){ ... }
```

A fájlfolyam megszűnésekor automatikusan záródik a hozzá tartozó fájl is. Ezt az adatfolyam osztály destruktora végzi. Ha az fájlt az adatfolyam hatókörének megszűnése előtt le akarjuk zárni, akkor ezt a `close()` tagfüggvénnyel tehetjük meg.

```
{
   ofstream outfile(”out.txt”);
   ...
   outfile.close();
   //...
}
```

Az eddigi ismereteinkkel már elkészíthetünk egy másolást végző programot, amely egy megadott bemenetet bájtonként átmásol egy megadott kimenetre.

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

int main(int argc, char * argv[]){
   if(argc!=3){
      cerr<<"Helytelen használat"<<endl;
      exit(1);
   }
   ifstream infile(argv[ 1 ]);
   if(!infile) {
      cerr<<"Allomany nyitasi hiba:"<<argv[ 1 ]<<endl;
      exit(1);
   }
   ofstream outfile(argv[ 2 ]);
   if(!outfile) {
      cerr<<"Allomany nyitasi hiba:"<<argv[ 2 ]<<endl;
      exit(1);
   }
   char ch;
   while(infile.get(ch))
      outfile.put(ch);
   if(!infile.eof()){
      cerr<<"Hibas masolas"<<endl;
      return 1;
   }
   return 0;
}
```

`Feladat:` Készítsünk egy függvényt, amely kiszámítja egy valós számokat tartalmazó adatfolyam összegét. Használjuk a függvényt a szabványos bemenetre, illetve egy fájlfolyamra is.

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

double sumStream(istream& is){
   double v, s = 0;
```

Kapcsoló	Jelentés	C nyelvi megfelelő
`ios::in`	Olvasás(létező fájl)	"r"
`ios::out`	Létrehozás(törlés), írás	"w"
`ios::out|ios::trunc`	Létrehozás(törlés), írás	"w"
`ios::out|ios::app`	Hozzáfűzés (létrehozás)	"a"
`ios::in|ios::out`	Írás/Olvasás (létező fájl)	"r+"
`ios::in|ios::out|ios::trunc`	Törlés, olvasás, írás(létrehozás)	"w+"

10.7. táblázat. Fájl-megnyitási módok

```
    while(is >> v) s += v;
    if(!is.eof())
       throw ios::failure("Input error");
    return s;
}

int main(){
    cout<<"Kerem a szamokat: ";
    cout<<"Osszeg: "<<sumStream( cin)<<endl;
    ifstream ifs("in.txt");
    if(ifs)
       cout<<"Osszeg: "<<sumStream( ifs)<<endl;
    else
       cout<<"Hiba az in.txt megnyitasanal"<<endl;
    return 0;
}
```

10.5.1. Fájl-megnyitási módok

Az eddigi példákban nem használtunk megnyitási módokat explicit módon. Ha adatfolyam létrehozásánál nem adjuk meg a megnyitási módot, akkor az alapértelmezettet fogja használni, amely `ifstream` esetén olvasásra való megnyitást (`ios::in`) jelent, `ofstream` esetében írásra való megnyitást (`ios::out`) és `fstream` esetében pedig írásra és olvasásra való megnyitást(`ios::in|ios::out`). A megnyitási módokat az `ios_base` osztály definiálja (az `ios` pedig örökli) és az 10.7. táblázatban foglaltuk össze.

Feladat: Készítsünk programot, amely két vagy több fájlnevet kap argumentumként és az első fájl végéhez fűzi az összes többi tartalmát.

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

using namespace std;

int main(int argc, char * argv[]){
   int i;
   if(argc < 3){
      cout<<"Hasznalat fuz file1 file2 ..."<<endl;
      exit(1);
   }
   ofstream outf(argv[ 1 ] , ios::out|ios::app);
   for(i=2; i<argc; ++i){
      ifstream f(argv[ i ]);
      if(f){
         char c;
         while(f.get(c))
            outf.put(c);
      }
      else
         cout<<"Megnyitasi hiba:"<<argv[ i ]<<endl;
   }
}
```

10.5.2. Közvetlen elérésű fájlok

Az eddigi példákban az fájlokat vagy csak írtuk, vagy csak olvastuk, következtetésképpen vagy bemeneti vagy pedig kimeneti fájlfolyamok voltak. Ugyanazt az fájlt megnyithatjuk úgy, hogy írás és olvasási műveleteket is végezzünk. Mivel az fájl bármely pozíciójáról olvashatunk, illetve írhatunk, szükségünk van adott pozíció beállítását, illetve lekérdezését végző műveletekre. A pozícióval kapcsolatos függvényeket az 10.8. táblázat foglalja össze:

A `tellg()` és `tellp()` függvények `pos_type` típust térítenek vissza, ezért használatuk a következőképpen történik:

Osztály	Tagfüggvény	Jelentés
basic_istream<>	`tellg()`	Visszatéríti az olvasási pozíciót
basic_istream<>	`seekg(pos)`	Beállítja az olvasási pozicíót (abszolút)
basic_istream<>	`seekg(offset,pos)`	Beállítja az olvasási pozicíót (relatív)
basic_ostream<>	`tellp()`	Visszatéríti az írási pozíciót
basic_ostream<>	`seekp(pos)`	Beállítja az írási pozicíót (abszolút)
basic_ostream<>	`seekp(offset,pos)`	Beállítja az írási pozicíót (relatív)

10.8. táblázat. Adatfolyamok pozíciókezelő függvényei

```
ios::pos_type pos = file.tellg();
```

vagy

```
streampos pos = file.tellg();
```

Ha a relatív pozíciókat használjuk, segítségünkre lehet az alábbi három állandó:

`ios::beg` - a pozíció az fájl elejéhez van viszonyítva
`ios::cur` - a pozíció az aktuálishoz viszonyítva
`ios::end` - a pozíció az fájl végéhez viszonyítva

Példák:

```
file.seekg(0, ios::beg); //fájl eleje
file.seekg(10,ios::cur); //10 karakterrel előbbre
file.seekg(-10,ios::end); //10 karakterrel a vége előtt
```

10.5.3. Fájlok bináris feldolgozása

Fájlok bináris feldolgozását az `istream` osztály `read` illetve az `ostream` osztály `write` tagfüggvényével végezhetjük.

```
istream& read(char * buffer, streamsize count);
ostream& write(const char * buffer, streamsize count);
```

Példaként írjuk be egy bináris fájlba az 1-10 közötti egész számokat, majd olvassuk vissza és írassuk ki a szabványos kimenetre.

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

int main(){
   ofstream ofs("out.dat");
   int i;
   for(i=1; i<=10; ++i)
      ofs.write((char *) &i, sizeof(int));
   ofs.close();
   ifstream ifs("out.dat");
   while(ifs.read((char *) &i, sizeof(int))){
      cout<<i<<endl;
   }
   return 0;
}
```

Végezetül pedig tekintsük egy telefonkönyv megvalósítását bináris adatfolyamok segítségével. A telefonkönyv bejegyzéseket tartalmaz. Egy bejegyzést a `PhoneRecord` osztállyal fogunk megvalósítani. Ez alapvetően egy adataggregátum, amelyhez inserter és extractor operátorokat készítünk a kényelmes kezelhetőség érdekében. A telefonkönyvet egy `PhoneBook` osztállyal valósítjuk meg. A `PhoneBook` osztály alapvető műveletei a következők:

- adott sorszámú bejegyzés lekérdezése
- bejegyzés hozzáfűzése
- adott sorszámú bejegyzés frissítése

PhoneRecord.h

```
#ifndef _PHONERECORD
#define _PHONERECORD

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

```
class PhoneBook;
class PhoneRecord{
private:
   char name[30];
   char tel[15];
public:
   PhoneRecord(char * pname="", char * ptel=""){
      strcpy(name, pname);
      strcpy(tel, ptel);
   }
   const char * getName() const { return name; }
   const char * getTel () const { return tel; }
   friend istream& operator>>(istream& is, PhoneRecord& rec);
   friend ostream& operator<<(ostream& os, PhoneRecord& rec);
   friend class PhoneBook;
};
#endif
```

PhoneRecord.cpp

```
#include "PhoneRecord.h"

istream& operator>>(istream& is, PhoneRecord& rec){
   is>>rec.name>>rec.tel;
   return is;
}
ostream& operator<<(ostream& os, PhoneRecord& rec){
   os<<rec.name<<" "<<rec.tel;
   return os;
}
```

PhoneBook.h

```
#ifndef _PHONEBOOK_H
#define _PHONEBOOK_H

#include <iostream>
#include <fstream>
#include <cstring>
#include "PhoneRecord.h"
using namespace std;
```

```
class PhoneBook{
private:
   fstream datafile;
public:
   PhoneBook(const char * filename){
      datafile.open(filename, ios::in|ios::out);
      if(!datafile){
         datafile.open(filename, ios::in|ios::out|ios::trunc);
         if(!datafile){
            cout<<"File open error"<<endl;
            exit(1);
         }
      }
   }
   void close(){ datafile.close();}
   void addRecord(PhoneRecord& rec);
   bool getRecord(PhoneRecord& rec, int recnum);
   bool updateRecord(PhoneRecord& rec, int recnum);
};
#endif
```

PhoneBook.cpp

```
#include "PhoneBook.h"
#include "PhoneRecord.h"
#include <iostream>
using namespace std;

void PhoneBook::addRecord(PhoneRecord& rec){
   datafile.seekp(0, ios::end);
   if (! datafile.write((char *)&rec, sizeof(PhoneRecord )))
      cout<<"Write error"<<endl;
}

bool PhoneBook::getRecord(PhoneRecord& rec, int recnum){
   if(!datafile.seekg((recnum-1) *sizeof(PhoneRecord),
                     ios::beg))
      return false;
   if(!datafile.read((char *)&rec, sizeof(PhoneRecord) ))
      return false;
   return true;
}
```

```
bool PhoneBook::updateRecord(PhoneRecord& rec, int recnum ){
   if (!datafile.seekp((recnum-1) *sizeof(PhoneRecord )))
      return false;
   if(!datafile.write((char *)&rec, sizeof(PhoneRecord) ))
      return false;
   return true;
}
```

main.cpp

```
#include "PhoneRecord.h"
#include "PhoneBook.h"
#include <iostream>
using namespace std;

void menu(){
   cout<<endl;
   cout<<"Please select:"<<endl;
   cout<<"1. Add record"<<endl;
   cout<<"2. Get record"<<endl;
   cout<<"3. Update record"<<endl;
   cout<<"4. Exit"<<endl;
}

int main(){
   PhoneBook book("tel.dat");
   PhoneRecord rec;
   int pos;
   do{
      char ch;
      menu();
      cin>>ch;
      switch(ch){
         case '1': cout<<"Record:";cin>>rec;
                   book.addRecord(rec); break;
         case '2':
                   cout<<"Position:";cin>>pos;
                   if(book.getRecord( rec, pos))
                      cout<<"Record: "<<rec<<endl;
                   else
                      cout<<"Get error"<<endl;
                   break;
         case '3':
```

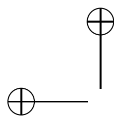
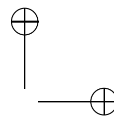
```
                    cout<<"Position:";cin>>pos;
                    cout<<"Record :";cin>>rec;
                    if(!book.updateRecord( rec, pos))
                       cout<<"Update error"<<endl;
                    break;
          case '4': exit(1);
          default:
                    cout<<"Undefined"<<endl;
       }
    }
    while(true);
    return 0;
 }
```

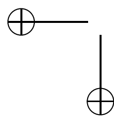
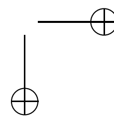
10.6. Feladatok

1. Készítsen programot, amely egy állomány nagybetűsítést végzi. Az állományokat adatfolyamok segítségével kezeljük, magát a nagybetűsítést pedig kötelező a `transform` algoritmussal végezni.

2. Készítsen egy olyan osztályt, amelyben az index operátor segítségével közvetlenül olvashatjuk be a fájl karaktereit. A fájl karaktereinek írására készítsünk egy közönséges függvényt.

3. Általánosítsa az előző feladatot. Készítse el az index operátort úgy, hogy írásra és olvasásra is használhassuk.

1. Melléklet - Definíciók

1. Programozási paradigma- Programozási mód. Alapvetően a program felépítésére használt eszközkészletet jelenti, vagyis milyen egységek képezik a program alkotóelemeit. (Moduláris programozás, Objektumorientált programozás, általánosított programozás, aspektus-orientált programozás stb.)

2. Absztrakt adattípus- Az adattípus leírásának legmagasabb szintje, amelyben az adattípust úgy specifikáljuk, hogy az adatok ábrázolására és a műveletek implementációjára semmilyen előírást nem adunk. Lehetőleg matematikai fogalmakat használva írjuk le az adattípust (halmazok és ezeken értelmezett műveletek).

3. OOP- Objektumorientált programozás-Olyan programozási paradigma, amely a programokat objektumokból építi fel. A program működése tulajdonképpen objektumok kommunikációját jelenti.

4. Osztály- Az osztály egy felhasználói típus, amelynek alapján példányok (objektumok) hozhatók létre. Az osztály alapvetően adat és metódus (művelet) definíciókat tartalmaz.

5. Objektum (példány)- Információt (adatokat) tárol és kérésre műveleteket végez.

6. Üzenet: Objektumhoz továbbított kérés. Válaszként az objektum végrehajtja a kért műveletet.

7. Interfész- Viselkedésmódot definiál. Gyakorlatilag egy művelethalmaz deklarációját jelenti. Ha egy osztály implementál egy adott interfészt,

akkor példányai az interfészben meghatározott viselkedéssel fognak rendelkezni.

8. Egységbezárás- Az adatok és a metódusok osztályban való összezárását jelenti.

9. Információ elrejtése- Az objektum elrejti az adatait és bizonyos műveleteit. Ez azt jelenti, hogy nem tudjuk pontosan, hogy egy objektumban hogyan vannak az adatok ábrázolva, sőt a műveletek implementációit sem ismerjük. Az információk elrejtése az objektum biztonságát szolgálja, amelyeket csak a ellenőrzött műveleteken keresztül érhetünk el.

10. Származtatás (örökítés)- Olyan osztályok között értelmezett viszony, amely segítségével egy általánosabb típusból (ősosztály) egy sajátosabb típust tudunk létrehozni (utódosztály). Az utódosztály adatokat és viselkedésmódot örököl.

11. Metódusok túlterhelése- Egy osztályon belül több azonos nevű, különböző szignatúrájú függvény. A függvényhívás argumentumai meghatározzák, hogy melyik függvény fog meghívódni. Ezt már a fordításidőben eldől (statikus, fordítás idejű kötés).

12. Metódusok felülírása- Egy osztályhierarchián belül, az utódosztály újradefiniálja az ősosztálytól örökölt metódust. (Azonos név, azonos szignatúra) Ha ősosztály típusú mutatón vagy referencián keresztül érjül el az osztályhierarchia példányait és ezen keresztül meghívjuk a felülírt metódust, akkor futási időben dől el, hogy pontosan melyik metódus kerül meghívásra. (dinamikus, futásidejű kötés).

13. Virtuális függvény- Polimorfikus viselkedést megvalósító függvény. A virtual kulcsszó segítségével kell bekapcsolni egy adott művelet többalakúságát.

14. Tiszta virtuális függvény- Virtuális függvénydeklaráció. Az adott szinten nincs megadva a függvény implementációja. Felületet meghatározó osztályokban használjuk és az utódosztályok fogják implementálni.

15. Absztrakt osztály- Olyan osztály, amelynek van legalább egy tiszta virtuális függvénye. Felületet határoz meg.

16. Polimorfizmus- Többalakúság. Ugyannarra a kérelemre a különböző objektumok különbözőképpen reagálnak. Lehetőséget nyújt a heterogén tárolók megvalósítására. A polimorfizmus lehet statikus és dinamikus.

    a) statikus polimorfizmus: metódusok túlterhelése, sablonfüggvények, sablonosztályok

    b) dinamikus polimorfizmus: metódusok felülírása.

17. Konstruktor- Az a művelet, amely inicializálja az objektumot. Egy osztálynak annyiféle konstruktora van, ahányféle képpen inicializálhatóak példányai.

18. Destruktor- A konstruktorral ellentétes művelet, általában a konstruktorban lekötött erőforrásokat szabadítja fel. Az objektum megsemmisítése előtt hajtódik végre és automatikusan hívódik.

19. Osztályszintű (statikus) tagok- A statikus adattagok, olyan adatok, amelyeket az adott osztály példányai közösen használnak (osztott). A statikus műveletek olyan műveletek, amelyek az argumentumaikon illetve az osztály statikus adattagjain 'dolgoznak". Ezek a tagok már példányok létrehozása előtt használhatók.

20. Inline függvények- Olyan függvények, amelyeket a fordító a hívás helyén kifejt, vagyis nem történik függvényhívás, hanem a hívás helyére behelyettesítődik a függvény kódja.

21. Konstans tagfüggvények- Olyan függvények, amelyek nem módosítják az objektum állapotát.

22. Privát örökítés (származtatás)- Hozzáférés szűkítő hatása van. Az ősosztálytól átvett adat és metódustagok privát tagokká alakulnak, ezáltal az utódosztály már nem biztosítja az ősosztály által meghatározott viselkedésmódot.

23. Tároló (Konténer) - Olyan típus, amely objektumok tárolását biztosítja. A tárolási funkció mellett karbantartó műveleteket is biztosít.

24. Bejáró (Iterátor) - Olyan típus, amely poziciót határoz meg egy halmazban (tároló, adatfolyam). Műveletein keresztül biztosítja a tároló bejárását, azaz a tárolt elemek egymás utáni feldolgozását.

25. Algoritmus - általánosan megvalósított függvény, amely minimális követelményt támaszt azon adatokkal szemben, amelyeken végrehajtódik.

26. Függvényobjektum: Függvényként viselkedő objektum. Az az előnye a függvénymutatóhoz képest, hogy mint objektum, állapotot is tárol, nemcsak függvényként viselkedik. Megvalósítás: olyan osztállyal, amelyben értelmezzük a függvényhívás operátort. Ezen kívül az osztály tartalmazhat adattagokat és más segédműveleteket is.

27. Sorozat: Olyan tároló, amelyben minden elemnek van egy rögzített pozíciója, amelyet a beszúrás helye és ideje határoz meg.

28. Asszociatív tároló: Olyan tároló, amelyben az elemek valamilyen rendezettségi kritérium szerint vannak tárolva. A beszúrás helyét nem a beszúrás ideje, hanem a beszúrt elem értéke határozza meg.

29. Sablonfüggvény: Típusparaméterekkel ellátott függvény, amely egy függvénycsaládot határoz meg.

30. Sablonosztály: Típusparaméterekkel ellátott osztály, amely egy típuscsaládot határoz meg.

31. Sablonspecializáció: Egy sablonosztály sajátosítása adott típusra, amelynek célja vagy az adott típusra való optimalizálás, vagy az adott típusra előállt rendellenes viselkedés kikerülése.

32. Adatfolyam: Csatorna, amely segítségével az adatok eljutnak az adatforrástól az adatnyelőig. Az adatforrás végpont lehetőséget biztosít adatok beírására a csatornába, míg az adatnyelő, a másik végpont az adatok kiolvasását biztosítja.

SZAKIRODALOM

[1] Bjarne Stroustrup, A C++ programozási nyelv, Kiskapu, 2001.

[2] Nicolai M. Josuttis, The C++ Standard Library, A Tutorial and Reference, Addison-Wesley, 1999.

[3] Andrei Alexandrescu, Programarea modernă în C++, Teora, 2002.

[4] Scott Meyers, STL biblioteca programatorului, Teora, 2002.

[5] Matthew H. Austern, Generic Programming and the STL, Addison-Wesley, 1999.

[6] David Vandevoorde, Nicolai M. Josuttis, C++ Templates, Addison-Wesley, 2003.

[7] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Programtervezési minták, Kiskapu, 2004.

[8] Krzysztof Czarnecki, Ulrich W. Eisenecker, Generative Programming, Addison-Wesley, 2000.

[9] James O. Coplien, Multi-Paradigm Design for C++, Addison-Wesley, 2003.

[10] Stephen C. Dewhurst, C++ hibaelhárító, Kiskapu, 2003.

[11] Andrei Alexandrescu, Herb Sutter, C++ kódolási szabályok, Kiskapu, 2005.

[12] Andrew Koenig, Barbara E. Moo, Accelerated C++, Addison Wesley, 2000.

[13] Nicolai M. Josuttis, Object-Oriented Programming in C++, Addison Wesley, 2003.

[14] Scott Meyers, Effective C++, Addison Wesley, 2005.

[15] Herb Sutter, Exceptional C++ style, Addison Wesley, 2005.