Egyszerű C++ program szerkezete

A konzol alkalmazás projektjébe egyetlen cpp kiterjesztésű (alapértelmezésben `main.cpp`) forrásállomány tartozik. Az állomány elején a kódrészben használt utasításokhoz szükséges könyvtárakat nevezhetjük meg az `#include` kezdetű sorok segítségével. Ezek között a leggyakrabbak az `#include <iostream>` (szabványos beolvasáshoz és kiíráshoz), az `#include <fstream>` (szöveges állományok kezeléséhez), az `#include <string>` (karakterláncok kezeléséhez) és az `#include <cmath>` (matematikai függvények használatához). Az `#include` kezdetű sorokat mindig a `using namespace std` utasítás kövesse!

Ezután következzen az úgynevezett `main()` függvény, amelyben programkódunkat helyezzük el. A végrehajtható utasításokból és deklarációkból álló programkód többnyire hármas tagozódású: beolvasás, számítás és kiírás részekből áll, de nem ritka, hogy a beolvasás vagy a kiírás összekeveredik a számítás résszel. A végrehajtható utasítások strukturált szerkezetben helyezkednek el, amelyet tabulátorok elhelyezésével emelünk ki. A deklaráció megnevezi a programban használt változókat, megadnja azok típusát. Azokat a változókat, amelyek a feladat tervében szerepelnek, a programkód elején deklaráljuk, a többi változót pedig abban a blokkban, ahol szükség van rájuk.

A kód végén mindig egy úgynevezett kilépő utasításnak (`return 0`) kell állnia. Az itt szereplő nulla érték a programot futtató környezethez, az operációs rendszerhez jut el, és azt jelzi, hogy a program rendben lefutott. A hibás működés miatti leálláskor ettől eltérő hibakód keletkezik. A kódban magunk is elhelyezhetünk olyan kilépő utasítást, amely hibakóddal áll le. Erre az egyszerű programokban egyformán használható a `return 1` és az `exit(1)`, de az összetettebb programok esetén az utóbbi az általánosabb megoldás, ehhez azonban szükség van az `#include <cstdlib>`-re.

A konzolos alkalmazások végrehajtása során egy úgynevezett konzol-ablak nyílik a képernyőn, amelyben a billentyűzet keresztül bevitt jeleket, illetve a képernyőre kiírt értékeket olvashatjuk. A leállás után ez az ablak megszűnik. Ha egy fejlesztő eszköz keretében futtatunk egy programot, akkor annak befejeződése után többnyire csak egy külön <enter> leütésre szűnik meg ez a konzol-ablak. Ha azonban programot önállóan futtatjuk, akkor a program leállásakor ez az ablak azonnal megszűnik, így nincs lehetőségünk a konzolablakba írt eredmények elolvasására. Ennek kivédésére a leállási pontok elé várakozási utasítást helyezhetünk el. A `char ch; cin >> ch;` ugyan nem túl elegáns, de operációs rendszerektől független várakozó utasítás, amely feloldásához előbb tetszőleges karaktert, majd az <enter> billentyűt kell leütni.

Deklaráció

A változók deklarációja a típus nevének és utána legalább egy szóközt követően a változó nevének megadásából áll. A deklarációk végrehajtásakor megfelelő méretű hely foglalódik le a memóriában, ahol bizonyos esetekben (a változó típusától függően) a változó értéke tárolódik, máskor viszont csak az érték eléréséhez szükséges segédinformáció. Több azonos típusú változó egyszerre is deklarálható ha a típus név után a változóneveket vesszővel elválasztva felsoroljuk. Egy változó deklarációja összevonható a kezdeti értékadásával.

```
típusnév változó;
típusnév változó1, változó2, változó3;
típusnév változó = érték;
típusnév változó = kifejezés;
```

Alapvető típusok

A C++ nyelvben leggyakrabban használt alaptípusok az alábbiak:

Egész illetve természetes számok típusa

típusnév: **int**
értékek: `83, -1320`
műveletek: `+ - * / % == != < <= > >=`
megjegyzés: az osztás eredménye egész

Valós számok típusa

típusnév: **double**
értékek: `83.0, -1320.345`
műveletek: `+ - * / == != < <= > >=`
megjegyzés: az osztás eredménye valós

Logikai típus

típusnév: **bool**
értékek: `false, true`
műveletek: `&& || ! == !=`

Karakterek típusa

típusnév: **char**
értékek: `'a', '3', '\''`
műveletek: `== !=`

Karakterlánc típusa

típusnév: **string**
értékek: `"valami", " "`
Karakterlánc-konstans sortörése:

```
"Ez itt egy nagyon hosszú \
másik sorba is átnyúló szöveg"
```

műveletek: `size(length) + [] c_str() substr() == !=`
`string str` esetén:

`str.size()`	az `str` hossza (karaktreinek száma)
`str[i]`	az `str` `i`-edik karaktere
`str + "vége"`	két karakterlánc összefűzése
`str.c_str()`	`str` átalakítása C stílusú lánccá
`str.substr(i,j)`	`str` részének kivágása
`atoi(str.c_str())`	számmá konvertálása

megjegyzés: Alkalmazásához szükséges az `#include <string>` direktríva.
A string az `std` névtér eleme.

Elemi programok

Értékadás

A változó := kifejezés értékadást a C++ nyelvben `változó = kifejezés` alakban írjuk.

Szimultán értékadások

A szimultán értékadásokat közönséges (rendszerint egy sorba írt) értékadások szekvenciájaként (lásd szekvencia) lehet leírni.

Értékadás értéke

A C++ nyelv értékadásának, mint utasításnak, értéke is van, mégpedig az értékül adott kifejezés értéke. Más szóval egy értékadás egyszerre utasítás is, és kifejezés is. Ezért lehet például olyan értékadásokat írni, mint

```
változó1 = változó2 = kifejezés;
```

amely mindkét változónak a kifejezés értékét adja. Ezt olyan szimultán értékadások kódolásánál használjuk, ahol minden változó ugyanazt az értéket kapja meg.

Gyakori hiba, hogy a `változó == kifejezés` egyenlőség-vizsgálat helyett `változó = kifejezés` formát írunk. Sajnos – mivel az értékadásnak is van értéke, és egy tetszőleges érték könnyen konvertálódik logikai értékké – ezt a fordítóprogram nem jelzi hibaként. Ha például az `i==1` kifejezés helyett rosszul az `i = 1` értékadást használjuk, akkor ennek az értékadásnak az értéke `1`, amit a C++ `true` logikai értéknek tekint. Ez a logikai kifejezésnek szánt értékadás mindig igaz lesz, miközben az `i` változó új értéket kap.

Speciális értékadások

```
++i, i++        i = i+1
--i, i--        i = i-1
```

Vigyázat! Kifejezésként használva az i++ az i kezdeti értékét a ++i az i megnövelt érékét képviseli.

Programszerkezetek

Szekvencia

Az utasítások végrehajtási sorrendje (szekvenciája) az utasítások egymás utáni elhelyezésével fejezhető ki. Mivel az utasítások végét általában pontosvessző jelzi, ezért a pontosvessző egyfajta szekvencia-határolónak (szekvencia pontnak) számít.

Több utasítás szekvenciája befoglalható egy úgynevezett utasításblokkba, amelyet a fordító egyetlen – bár összetett – utasításként kezel. Az utasításblokk elejét a nyitó kapcsos zárójel, végét a csukó kapcsos zárójel jelzi. Mivel az utasításblokk egy utasítás, utasításblokkok szekvenciájáról is beszélhetünk, ahol a szekvencia-határolók a csukó kapcsos zárójelek.

Összetett utasításnak, azaz szekvenciának tekinthető az alábbi értékadás is. Ez először megnöveli a k értékét, utána ezt a megnövelt értéket adja `j`-nek, majd `i`-nek.

```
i = j = ++k;
```

Elágazás

Az elágazás az a program, amely egy feltétel aktuális logikai értékétől függően más-más részprogramot hajt végre. Többféle írásmódot is szoktak alkalmazni:

```
if(feltétel)
{
      ág_1
}
else
{
      ág_2
}
```

```
if(feltétel){
      ág_1
}
else{
      ág_2
}
```

```
if(feltétel){
      ág_1
}else{
      ág_2
}
```

Egyszerűbb esetben

```
if(feltétel)    ut_1;
else            ut_2;
```

A ki nem írt „else” ág üres utasítású ágat jelent:

```
 if(feltétel)ut_1;
```

Sokágú elágazás

A sokágú elágazást egymásba ágyazott „if-else” elágazásokkal kódolhatjuk. Tudni kell azonban, hogy így – az absztrakt elágazással ellentétben, amelyik nem-determinisztikus és ha egyik feltétel sem teljesül, akkor abortál – ez determinisztikus és ha egyik feltétel sem teljesül, akkor üres programként működő kódot kapunk. Ez azonban nem baj, mert ha az absztrakt elágazás helyes volt, akkor annak valamelyik feltétele biztosan teljesül, és bármelyik determinisztikus változat is helyes. A kódban használhatunk sok ág után egy külön „else” ágat is, amely az absztrakt változathoz képest egy extra lehetőséget biztosít.

```
if(feltétel_1){
      ág_1
}
else if(feltétel_2){
      ág_2
}
else if(feltétel_3)
 …
else{
      ág_n+1
}
```

Speciális sokágú elágazás

```
switch (kifejezés) {
      case konstans1 :  utasítássorozat1  ;  break;
      case konstans2 :  utasítássorozat2  ;  break;
      case konstans3 :
      case konstans4 :  utasítássorozat34 ;  break;
      default        :  utasítássorozat5  ;
}
```

Ciklus

Ciklusról akkor beszélünk, amikor egy programot (ciklusmag) mindannyiszor újra és újra végrehajtunk, valahányszor egy megadott feltétel (ciklusfeltétel) aktuális logikai értéke igaz.

```
while(feltétel){
      mag
}
```

Ciklust tartalmazó nevezetes összetett utasítások

A **for** utasítás a

```
prog_1;
while(felt){
    mag;
    prog_2;
}
```

kód egyszerűbb írására használható. Formája:

```
for(prog_1;felt;prog_2){
    mag
}
```

Általában az úgynevezett számlálós ciklusokhoz, illetve az iteratív szerkezetű adatok feldolgozásához használjuk.

```
for(i=0;i<n;++i){
    mag
}
```

Az `i` az úgynevezett ciklus számláló.

Ha a ciklusmag egyetlen utasításból áll, akkor használható az alábbi forma:

```
for(i=0;i<n;++i) utasítás;
```

A **do-while** utasítás a

```
mag;
while ( felt ){
    mag
}
```

kód egyszerűbb írására használható. Formája:

```
do{
    mag
}while ( felt )
```

Egyrészt adatbevitel ellenőrzésénél használjuk:

```
do{
    adatbevitel
    if(kritérium) üzenet
}while ( kritérium )
```

másrészt amikor a programot alkalmassá tesszük arra, hogy tetszőleges sokszor futtathassuk:

```
do{
    program
    cout << "Folytatja? (I/N)";
    char ch; cin >> ch;
}while (ch!='N' && ch!='n');
```

Szabványos be- és kimenet

Az alábbi kódrészlet a szabványos beolvasás, illetve kiírás legfontosabb utasításait mutatja be.

```
#include <iostream>

cin   >> változó1   >> változó2;
cout  << kifejezés1 << kifejezés2;
cout  << endl;
```

Szám ellenőrzött beolvasása

Sokszor van olyan kódra szükségünk, amellyel egy egész számot kell beolvasnunk.

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

Ez a beolvasás „elszáll”, ha nem számformájú karaktereket adunk meg. Ezt a hibát a `cin.fail()` függvényhívással kérdezhetjük le: ha ez igazat ad vissza, akkor nem sikerült az olvasás. Jó tudni, hogy ezután csak akkor lehet újra olvasni, ha egyrészt töröljük a hibaesetet (`cin.clear()`), másrészt egy speciális utasítással kiürítjük a billentyűzet-puffert (`getline(cin,tmp,'\n')`, ebben a `tmp` egy `string` típusú változó).

Az alábbi programrészlet egy pozitív egész számot olvas be. Mindaddig újra és újra próbálkozik a beolvasással, amíg nem kap helyes adatot:

```
int n;
bool hiba = false;
do{
   cout << "Adjon meg egy pozitív egész számot ";
   cin >> n;
   hiba = cin.fail();
   if(hiba) cin.clear();
   string tmp = "";
   getline(cin,tmp);
   hiba = hiba || tmp.size()!=0 || n<0;
   if(hiba) cout << "Hibás adat!" << endl;
}while(hiba);
```

Szerkesztett formátumú beolvasás, kiírás

A formátumjelző flageket a `cin.setf()`, `cout.setf()`, `cin.unsetf()`, `cout.unsetf()` függvények paramétereként kell felsorolni a | jellel elválasztva.. A `setf()` beállítja, az `unsetf()` kikapcsolja őket, a kettő között érvényben vannak. Némelyikük alapértelmezett módon be van kapcsolva. Minden flag elé az `ios::` minősítést kell írni.

Formátumjelző flagek:

`scientific,`	lebegőpontos alak
`fixed,`	fixpontos alak
`right, left,`	jobbra ill. balra tömörítés,
`dec, hex, oct,`	szám megjelenítésének számrendszere
`showpoint, showpos`	előjel látszódjon-e

`skipws`	elválasztó jelek átlépése olvasáskor

A manipulátorok csak az adott adat mozgatásra vonatkoznak. Őket a << operátorral kell a standard kimenetre küldeni.

Manipulátorok:

`setw(int w), width(int w)`	mezőszélesség megadása
`setprecision(int p), precision(int p)`	számábrázolás pontossága
`setfill(char c) ....`	kitöltő karakter definiálása
`endl`	sorvége

Karakter beolvasása

A >> operátorral egyetlen karaktert is be lehet olvasni, feltéve, hogy az nem elválasztó jel. Ha elválasztó jeleket is be szeretnénk olvasni, akkor vagy kikapcsoljuk a whitespace-eket átugró mechanizmust: `cin.unsetf(ios::skipws)`, vagy a `get()` függvénnyel olvasunk.

```
char ch;
cin.get(ch);
```

A `get()` függvény párja a `put()`, amivel egyetlen karaktert lehet kiírni.

Sor beolvasása

Szóközzel elválasztott vagy üres karakterláncot nem tudunk a >> operátorral beolvasni. Ilyenkor a beolvasásnak másik formáját kell használni:

```
getline(cin,str);
```

Ügyeljünk azonban arra, hogy ha vegyesen használjuk a kétféle olvasási módot, akkor a `getline()` alkalmazása előtt ürítsük ki a billentyűzet puffert.

Összetett szerkezetű típusok

A típusszerkezetekkel egyszerűbb típusokból építhetünk fel összetett szerkezetű típusokat.

Tömb

A tömb több azonos típusú elem tárolására használt adatszerkezet, amelyben megengedett az adott indexű elemre való hivatkozás.
A C++ nyelven többféleképpen is használhatunk tömböket. Ezek némelyike túl mutat a tömb használatán, mert olyan műveleteket is megenged (például tömbhöz új elem hozzáfűzése vagy utolsó elemének elhagyása), amelyeket egy tömbre nem szokás megengedni.

Fordítási időben rögzített méretű, automatikus helyfoglalású egydimenziós tömb (vektor)

```
int t[100];            // 100 elemű tömb
int n;                 // tényleges méret: n<=100
cin >> n;
for(int i=0;i<n;++i){  // tömb feltöltése
   cin >> t[i];
}
```

Futási időben rögzített méretű, automatikus helyfoglalású egydimenziós tömb (vektor)

```
int n;
cin >> n;
int t[n];              // n elemű tömb
for(int i=0;i<n;++i){  // tömb feltöltése
   cin >> t[i];
}
```

Futási időben rögzített méretű, dinamikus helyfoglalású egydimenziós tömb (vektor)

```
int* t;                // véges méretű tömb deklarálása
int n;                 // tényleges méret
cin >> n;
t = new int[n];        // tömb lefoglalása futás közben
for(int i=0;i<n;++i){  // tömb feltöltése
   cin >> t[i];
}
delete[] t;            // tömb felszabadítása
```

Változtatható méretű egydimenziós vector<>

```
int n;
cin >> n;
vector<int> t(n);      // tömb deklarálása a méretével együtt
t.size()               // tömb elemeinek száma
vector<int> t;         // véges méretű tömb deklarálása
t.resize(n);           // tömb méretének megváltoztatása
for(int i=0; i<(int)t.size();++i) // tömb feltöltése
   cin >> t[i];
}
t.push_back(új);       // tömb végére új elemet illeszt
```

Fordítási időben rögzített méretű, automatikus helyfoglalású kétdimenziós tömb (mátrix)

```
Elem t[10][20];        // lefoglalt mátrix
int n,m;               // tényleges méret
cin >> n >> m;
for (int i=0;i<n;++i) // t feltöltése
   for (int j=0;j<m;++j)
      cin>>t[i][j];
```

Futási időben rögzített méretű, automatikus helyfoglalású kétdimenziós tömb (mátrix)

```
int n,m;               // méret
cin >> n >> m;
Elem t[n][m];          // lefoglalt mátrix
for (int i=0;i<n;++i) //  t feltöltése
   for (int j=0;j<m;++j)
      cin>>t[i][j];
```

Futási időben rögzített méretű, dinamikus helyfoglalású kétdimenziós tömb (mátrix)

```
Elem** t;              // véges méretű mátrix
int n,m;               // tényleges méret
cin >> n >> m;
t=new Elem*[n];
```

```
for (int i=0;i<n;++i){ // t feltöltése
   t[i]=new Elem[m];
   for (int j=0;j<m;++j) cin>>t[i][j];
}
for (int i=0;i<n;++i) delete[] t[i];
delete[] t;            // t felszabadítása
```

Változtatható méretű kétdimenziós vector<vector<> >

```
t.size()                      // mátrix sorainak száma
t[0]                          // mátrix első sora
t[0].size()                   // első sor elemeinek száma
t[0][0]                       // első sor első eleme
vector<vector<int> > t;       // mátrix deklarálása
t.resize(n);           // mátrix sorai számának megváltoztatása
t[0].resize(m)         // sor méretének megváltoztatása
for(int i=0;(int)i<t.size();++i) // téglalap mátrix feltöltése
   for(int j=0;(int)j<t.size();++j)
      cin >> t[i][j];
```

Struktúra

A rekord típusnak megfeleltetett szerkezet:

```
struct Nev {
   tipus1 mezo1;
   tipus2 mezo2;
   ...
};

Nev d;
d.mezo1 = …;
… = d.mezo1;
```

A `d.mezo1` kifejezés (a `mezo1` a szelektor függvény) állhat értékadás jobb- illetve baloldalán. Az egyik esetben felhasználjuk `d.mezo1` kifejezés által képviselt értéket, a másik esetben megváltoztathatjuk azt.

Szöveges állományok

A szöveges állományok olyan karakteres állományok, amelyekben bizonyos nem látható karakterek a szöveg tördeléséért felelősek. Ezeket az állományokat kétféleképpen használhatjuk: olvasásra vagy írásra. Mindkét művelet sorban, egymás után történik. Nemcsak karakterenként, hanem nagyobb szintaktikai egységenként is, például soronként, szavanként, stb. olvashatunk illetve írhatunk.

Szöveges állomány deklarálása, nyitása, ellenőrzése, zárása

Az

```
ifstream f;
ofstream f;
```

definiál egy úgynevezett bementi illetve kimeneti adatfolyam objektumot (`f` az objektum neve), amelyek segítségével a szöveges állomány adatait olvashatjuk vagy hozzájuk írhatunk. Szükségünk van még az alábbi sorra is:

```
#include <fstream>
```

Az

```
f.open(fnev.c_str());
```

megnyitja az adatfolyamot. Az adatfolyam forrásának, azaz a szöveges állománynak a nevét archaikus karakterlánc formában kell megadnunk. Az archaikus karakterláncokról elég annyit tudni, hogy egy `string` típusú változóban (legyen a neve: `str`) tárolt sztringnek az `str.c_str()` adja meg az archaikus alakját. Az adatfolyam definíciója és megnyitása összevonható egy lépésbe:

```
ifstream f(fnev.c_str());
ofstream f(fnev.c_str());
```

Az adatfolyam az

```
f.close();
```

utasítással zárható le, amelyre azonban automatikusan sor kerül akkor, amikor a vezérlés eléri az adatfolyam deklarációját tartalmazó blokk végét.

Egy adatfolyam megnyitásakor (akár közvetlenül, akár az open() utasítással történik) különféle hibák fordulhatnak elő. A leggyakoribb az, hogy a megnevezett szöveges állományt nem találjuk meg, mert elfelejtettük létrehozni, vagy nem abban a könyvtárban van, ahol keressük. A hibára a `fail()` függvény lekérdezésével kérdezhetünk rá. Hiba esetén értesíthetjük a felhasználót a szabványos kimentre küldött üzenettel, és egy alkalmas hibakóddal – ami nem nulla – leállítjuk a program futását: `exit(1)`.

```
ifstream f(fnev.c_str());
if (f.fail())
{
    cout << "Hiányzik az állomány" << endl;
    char ch; cin>>ch;
    exit(1);
}
```

Ha egy adatfolyamot egy sikertelen kísérlet után újra meg akarunk nyitni, akkor előbb az `f.clear()` utasítást ki kell adnunk.

```
ifstream f;
string fnev;
bool hiba;
```

```
do
{
    cout << "Az állomány neve: "; cin >> fnev;
    f.open(fnev.c_str());
    if(hiba = f.fail()){
        cout << "Fájl nyitási hiba!\n";
        f.clear();
    }
}while(hiba)
```

Olvasás, írás szöveges állománnyal

Az olvasásra, írásra majdnem ugyanazok a szabályok érvényesek, mint a szabványos bemenetre illetve kimenetre. Lényeges különbség az, hogy itt egy olvasás a soron következő szintaktikai egységre, azaz elválasztó jelekkel (szóköz, tabulátor, sorvége) határolt részre vonatkozik.

Általános formája beolvasásnál:

```
f >> valtozo;
```

Kiírásnál:

```
f << kifejezes;
```

Karakterek beolvasása szöveges állományból

Az

```
ifstream f;
char df;
f >> df;
```

olvasás alapértelmezés szerint átlépi az elválasztó jeleket. Ha az összes karaktert akarjuk egymás után beolvasni, akkor az olvasás megkezdése előtt ki kell kapcsolni ezt a tulajdonságot. Ehhez az

```
#include <iomanip>
ifstream f;
f.unsetf(ios::skipws)
```

vagy a

```
f.get(df);
```

kódot kell alkalmazni.

A kiírás egyformán működik a `f<<df` illetve az `f.put(df)` utasítással.

Sorok beolvasása szöveges állományból

A `getline()` függvénnyel lehetőségünk van egy teljes sort, vagy egy megadott elválasztó jelig tartó karakterláncot beolvasni egy szöveges állományból. Az alábbi két megoldás egyenértékű: mindkettő egy teljes sort olvas, hiszen a sorok végét a \n sorvége karakter jelzi.

```
ifstream f( … );
string sor;

getline(f, sor);
getline(f, sor, '\n' );
```

Egyéb nyelvi elemek

Program leállítása 1-es hibakóddal

Az `exit(1)` utasítás hatására programunk azonnal befejeződik, és az 1-es értéket (hibakódot) adja vissza a futtatási környezetnek (operációs rendszernek).

Várakozó utasítás

Elegáns általános várakozó utasítás nincs a C++-ban. A `char ch; cin >> ch;` egy kevésbé elegáns, de operációs rendszerektől független várakozó utasítás, amely feloldásához nem elég egy <enter> leütni, az előtt meg kell adni egy tetszőleges karaktert:

```
char ch; cin >> ch;
```

Megjegyzések kódba illesztése

Írhatunk egy- illetve többsoros megjegyzéseket a programkódba, amiket a fordító figyelmen kívül hagy.

Egy soros megjegyzés esetén:

```
// megjegyzés
```

Több soros megjegyzés esetén:

```
/*
     megjegyzés
*/
```

Sztringek kezelése

Egy sztringre számos hasznos előredefiniált függvény használható.

A find() függvénycsalád (sok változata van) egy sztringben keres karaktert vagy rész-sztringet, annak sztringbeli pozícióját adja vissza, ha nem talál, akkor a string::npos extremális értéket. Lehet a sztring adott pozíciójától kezdődően keresni az első vagy az utolsó előfordulást.

A sztringet megváltoztató függvények között találjuk a karaktert vagy rész-sztringet adott pozícióra beszúró (insert), adott pozícióról törlő (erase), adott pozíción helyettesítő (replace) műveleteket.

Hasznos lehetőséget rejt a sztringek feldolgozásában a stringstream-ek (`#include <sstream>`) alkalmazása.

Egy input-sztringfolyamba helyezett sztringet úgy tudunk feldolgozni, mintha azt egy szöveges állományból olvastuk volna:

```
string str = "Alma a fa alatt";
```

```
istringstream is;
is.str(str);
string word;
while(is >> word) { cout << word << endl;}
```

Az output-sztringfolyam segítségével tetszőleges sztring állítható össze úgy, hogy közben a sztringbe fűzött adatelemek konverziójára is sor kerül:

```
ostringstream os;
os << "A " << 3.2 << " egy valós szám ";
str = os.str();
cout << str << endl;
```