Fordítás

A futtatható állomány előállítása három lépésből áll: preprocesszálás → fordítás → összeállítás (preprocessor → compiler → linker).

Preprocesszálás

A preprocesszálás végeredménye ritkán jelenik meg külön állományként, a fordítás részeként fut le.

preproctest/preproctest.cc

```
#include "some_header.h"

int main()
{
    return 0;
}
```

preproctest/some_header.h

```
struct Something
{
    int a, b;
};
```

Ha azt akarjuk, hogy a fordító csak preprocesszáljon, akkor a `-E` kapcsolót kell használnunk gcc vagy clang esetén. Ekkor a stdoutra kiírja azt, ami egyébként a fordítónak ment volna. Ha a kimenetet megnézzük, az include egyszerűen csak beilleszti a megadott állományt a forrásba, ezen kívül belekerül pár különös jel, ezekkel tud a fordító a hibakijelzésnél az eredeti források soraira hivatkozni.

preproctest/output.txt

```
$ clang -E preproctest.cc
# 1 "preproctest.cc"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 167 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "preproctest.cc" 2
# 1 "./some_header.h" 1
struct Something
{
 int a, b;
};
# 2 "preproctest.cc" 2

int main()
{
 return 0;
}
```

Fordítás

A fordítás során minden egyes `.cc`/`.cpp` forrásból objektum állomány (`.o`/`.obj`) készül. Az objektum állomány tartalmazza, a változók neveit, a függvények neveit és gépi kódú forrását, valamint hogy ezek a függvények hol hivatkoznak meg változókat és más függvényeket. Itt még nincs eldöntve, hogy a memóriában hol fognak az egyes függvény kódok és globális változók elhelyezkedni.

A külső nevek típusának az ellenőrzése a fordító feladata, de ahhoz, hogy ezt megtehesse, tudnia kell róla.

refer1/refer1.cc

```
void foo(int x);
extern const char *bar;

int main()
{
    for(int i=0; i<16; ++i)
        foo(i);

    return 0;
}
```

refer1/util.cc

```
const char *bar = "BAR";

void foo(int x)
{
    // ...
}
```

Ha azt akarjuk, hogy csak a preprocesszáló és a fordító fusson le, használjuk a `-c` kapcsolót.

refer1/output.txt

```
$ clang -c refer1.cc
$ clang -c util.cc
$ ls *.o
refer1.o util.o
```

A máshol definiált függvényeket és globális változókat deklarálni minden egyes forrásban, ahol használni akarjuk komoly hibaforrás és rengeteg pluszmunka, ezért szokás egy `.h` forrásba beletenni mindazon változókat és függvényeket, amit egy másik modul számára elérhetővé akarunk tenni. Ezt teszi minden egyes rendszerkönyvtár.

refer2/refer2.cc

```
// "Elsosorban a helyi headerek kozott keressen"
#include "util.h"

int main()
{
    bar = "QUUX";
    for(int i=0; i<16; ++i)
        foo(i);

    return 0;
}
```

refer2/util.h

```
struct Record
{
    int a, b;
};

typedef const char *STR;
void foo(int x);
extern STR bar;
```

refer2/util.cc

```
// "Elsosorban a helyi headerek kozott keressen"
#include "util.h"
// <Elsosorban a rendszerheaderek kozott keressen>
#include <stdio.h>

STR bar = "BAR";

void foo(int x)
{
    printf("%d %s\n", x, bar);
}
```

refer2/output.txt

```
```

```
$ clang -c refer2.cc util.cc
$ clang refer2.o util.o -o refer2
$ ./refer2
0 QUUX
1 QUUX
2 QUUX
3 QUUX
4 QUUX
5 QUUX
6 QUUX
7 QUUX
8 QUUX
9 QUUX
10 QUUX
11 QUUX
12 QUUX
13 QUUX
14 QUUX
15 QUUX
```

Előfordulhat, hogy a `util.h`-ban definiált `Record` típust használja több modul is, pl. a függvények paraméterlistájában. Elkerülhetetlen, hogy emiatt közvetve többször is befűződjön a `util.h`. A típust újradeklarálni viszont nem lehet.

refer2/include.cc

```
#include "util.h"
#include "util.h"
```

refer2/include.txt

```
$ clang -c include.cc
In file included from include.cc:2:
./util.h:2:8: error: redefinition of 'Record'
struct Record
       ^
./util.h:2:8: note: previous definition is here
struct Record
       ^
1 error generated.
```

Ezért szokás a fejléceket makrókkal körülbástyázni, hogy ha újra megjelennének egy fordítási egységben, akkor lényegében nem tartalmaznak semmit.

refer2/util_safe.h

```
#ifndef _UTIL_SAFE_H_
#define _UTIL_SAFE_H_

struct Record
{
    int a, b;
};

typedef const char *STR;
void foo(int x);
extern STR bar;

#endif
```

A gyakorlatban minden népszerű fordító támogatja a nem szabványos **`#pragma once`** direktívát, amivel ugyanezt lehet egyetlen plusz sorral megvalósítani. A vizsgán ne használd.

Minden olyan függvényt, amit nem használunk modulon kívüli használatra érdemes úgy deklarálnunk, hogy azt más modulok még véletlenül se érhessék el, hogy egy nagyobb program esetén ne érjenek kellemetlen meglepetések. C-ben ezt a **`static`** kulcsszó használatával lehet elérni, C++-ban lehetőség van **`static`** kulcsszót használni vagy névtelen névteret létrehozni (a névterekről később).

refer3/util.cc

```

int foo;
static int bar;
```

refer3/refer3.cc

```
extern int foo;
extern int bar;

int main()
{
    ++foo;
    ++bar;
    return 0;
}
```

refer3/output.txt

```
$ clang -c refer3.cc util.cc
$ clang refer3.o util.o -o refer3
Undefined symbols for architecture x86_64:
  "_bar", referenced from:
      _main in refer3.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

A példában a `foo` változót megtalálta, de a `bar`t nem.

Összeállítás

A linker feladata, hogy eldöntse, hogy futás során hol fognak elhelyezkedni a memóriában a globális változók és a függvények gépi kódja, és hogy ez utóbbiak a már megállapított címen keressék őket.

Nagyon fontos tudni, hogy a linkerben a C++ már díszített (decorated) neveket használ, ellentétben a C-vel, mert C++-ban már nem elég pusztán a függvény nevét tudni a beazonosításhoz.

link1/util.c

```
int quux();
int corge();

int foo(const char *s)
{
    return 0;
}

void bar(int x)
{
    corge();
    quux();
}
```

link1/link1.cc

```
extern "C" int foo(const char *s);
void bar(int x);

int quux()
{
    return -1;
}

extern "C" int corge()
{
    return -1;
}

int main()
{
    foo("xyz");
    bar(-1);

    return 0;
}
```

link1/output.txt

```
$ clang -c link1.cc util.c
$ clang link1.o util.o -o link1
Undefined symbols for architecture x86_64:
  "bar(int)", referenced from:
      _main in link1.o
  "_quux", referenced from:
      _bar in util.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

A C++ kód nem látta azt a C függvényt, amelyiket nem`extern "C"` kulcsszóval deklaráltunk, illetve a C kód nem látta azt a C++ függvényt, amit nem ezzel a kulcsszóval definiáltunk.

Az`extern "C"` lényegében azt mondja: „erre a deklarációra vagy definícióra ne használj díszített neveket!” Van `extern "C" { /* ... */ }` formája is, amivel egyszerre több deklarációra vagy definícióra lehet ezt a módot beállítani. Elsősorban C rutinkönyvtárak fejléceiben találkozni vele:

```
#ifdef __cplusplus
extern "C" {
#endif

/* ... */

#ifdef __cplusplus
}
#endif
```

Új nyelvi elemek

A `const` kulcsszó

Előnyei:

- Ha átadok egy mutatót egy függvénynek, biztos lehetek, hogy nem fog a tartalma változni, és a fordítót is biztosítom erről: ennek megfelelően optimizálhatja a kódot
- Elegánsabban lehet vele konstanst definiálni, mint a `#define` direktívával: a konstansnak lesz típusa, és lehet címe (a prefix & operátorral)

new/const.cc

```
const int FOO = 5;
const float BAR = 7;

const char *string1 = "shibe";
const char * const CONSTANT_STRING = "wow";

void mutate(int *val);
void use(int val);
void read(const int *val);

int main()
{
    // hiba:
    FOO = 3;
    // hiba:
    BAR = 3;

    int a = 11;

    // OK:
    mutate(&a);
    // hiba:
    mutate(&FOO);

    // OK:
    use(a);
    // OK, mert ekkor masolja:
    use(FOO);

    // OK:
    read(&a);
    // OK:
    read(&FOO);

    // OK:
    string1 = "such";
    // hiba:
    CONSTANT_STRING = "many";

    // hiba:
    string1[0] = 0;
    // hiba:
    CONSTANT_STRING[0] = 0;

    return 0;
}
```

new/const.txt

```
$ clang -c const.cc
const.cc:15:6: error: read-only variable is not assignable
        FOO = 3;
        ~~~ ^
const.cc:17:6: error: read-only variable is not assignable
        BAR = 3;
        ~~~ ^
const.cc:24:2: error: no matching function for call to 'mutate'
        mutate(&FOO);
        ^~~~~~
const.cc:8:6: note: candidate function not viable: 1st argument ('const int *') would lose const qualifier
void mutate(int *val);
     ^
const.cc:39:18: error: read-only variable is not assignable
        CONSTANT_STRING = "many";
        ~~~~~~~~~~~~~~~ ^
const.cc:42:13: error: read-only variable is not assignable
        string1[0] = 0;
        ~~~~~~~~~~ ^
const.cc:44:21: error: read-only variable is not assignable
        CONSTANT_STRING[0] = 0;
        ~~~~~~~~~~~~~~~~~~ ^
6 errors generated.
```

Egyszerűbb függvény deklaráció

C-ben a `int main() int main(...)`-nak felelt meg, a paraméter nélküli függvényeket `int main(void)`-ként kellett deklarálni. C++-ban ez már nem érvényes, a `int main()` ekvivalens a `int main(void)` deklarációval.

Változó deklaráció bárhol

Bárhol lehet változót deklarálni, nem csak a blokkok elején és a for ciklusokban.

new/decl.cc

```
#include <stdio.h>

int main()
{
    printf("foo\n");

    int bar = 3;

    printf("Big whoop: %d\n", bar);

    return 0;
}
```

Függvények túlterhelése

A függvényeket már nem csak a nevük, hanem a paraméterlistájuk is azonosítja, amíg egyértelmű.

new/overload.cc

```
#include <stdio.h>

void print(int n)
{
    printf("Szam: %d\n", n);
}

void print(const char *str)
{
    printf("Szoveg: %s\n", str);
}

struct Point
{
    int x, y;
};

void print(const struct Point *p)
{
    printf("Pont: %d, %d\n", p->x, p->y);
}

int main()
{
    print(5);
    print("foo");

    struct Point p;
    p.x = 3;
    p.y = 7;
    print(&p);

    return 0;
}
```

new/overload.txt

```
Szam: 5
Szoveg: foo
Pont: 3, 7
```

Többek között emiatt van szükség az `extern "C"` módosítóra.

Függvény paraméter alapértelmezett értékek

A függvény utolsó paramétereinek adhatunk alapértelmezett értéket, ezzel azok a paraméterek elhagyhatóvá válnak. Paramétereket elhagyni csak a függvény végéről lehet, nem lehet pl. a harmadikat elhagyni, a negyediket viszont megadni.

new/default.cc

```cpp
#include <stdio.h>

void foo(int a, int b=1, int c=2, int d=3, int e=4)
{
    printf("%d %d %d %d %d\n", a, b, c, d, e);
}


int main()
{
    foo(0);
    foo(1, 0);
    foo(2, 1, 0);
    foo(4, 3, 2, 1, 0);
    return 0;
}
```

new/default.txt

```
0 1 2 3 4
1 0 2 3 4
2 1 0 3 4
4 3 2 1 0
```

Referenciák

Mutatók mellett referenciákat is használhatunk. A háttérben ezek is mutatók, de nem tudjuk a mutató értékét megváltoztatni, csak azt az értéket, amire mutat.

Ha a referenciát const kulcsszóval deklaráljuk, a fordító képes automatikusan ideiglenes változót létrehozni, ha kell (lásd a példát). Ha azonban a visszatérési érték referencia, akkor a biztonságos működéshez csak olyan referenciát adhatunk vissza, ami nincs a függvény élettartamához kötve. Ha mégis megtesszük, a program még le is fordulhat, de nem várt eredményeket produkálhat.

new/ref.cc

```cpp
#include <stdio.h>

void add1(int &dest, int a, int b)
{
    dest = a+b;
}

void add2(int &dest, const int &a, const int &b)
{
    dest = a+b;
}

void add3(int &dest, int &a, int &b)
{
    dest = a+b;
}

int foo = 0;

// OK, a foo akkor is letezik, amikor a fv mar kilepett
int& good()
{
    return foo;
}

// Nem OK, a val mar nem letezik, amikor a fuggveny erteket felhasznalna
// A fordito nem fog hibat jelezni (de figyelmeztethet).
// Meg jol is mukodhet néhány esetben, más esetekben viszont kellemetlen
// meglepeteseket okozhat. Ha szeretnenk, hogy a programunk ne szalljon
// el veletlenszeruen, ne tegyunk ilyet. Meg egyebkent sem :)
int& bad()
{
    int val = 1;
    return val;
}

// Erre is figyelmeztet, de lefordul. Ugyanaz vonatkozik
// ra, mint az elozore. A const kulcsszo nelkul le se fordulna,
// mert a const kulcsszonal a fordito uj valtozot hoz letre,
// es annak a cimet veszi. Viszont a fuggveny visszatertekor
// a fuggveny osszes valtozoja ervenyet veszti, igy ez az
// ideiglenes valtozo is.
const int& lit()
{
    return 5;
}

int main()
{
    int v;

    // OK:
    add1(v, 1, 2);
    // OK, mondhatjuk, hogy letrehoz egy ideiglenes valtozot,
    // es annak veszi a cimet:
    add2(v, 3, 4);
    // Hiba:
    add3(v, 5, 6);

    // A good a foo valtozora ad referenciat, tehat
    // lenyegeben foo = 3
    good() = 3;

    return 0;
}
```

new/ref.txt

```
$ clang -c ref.cc
ref.cc:34:9: warning: reference to stack memory associated with local variable 'val' returned [-Wreturn-stack
        return val;
               ^~~
ref.cc:45:9: warning: returning reference to local temporary object [-Wreturn-stack-address]
        return 5;
               ^
ref.cc:58:2: error: no matching function for call to 'add3'
        add3(v, 5, 6);
        ^~~~
ref.cc:13:6: note: candidate function not viable: expects an l-value for 2nd argument
void add3(int &dest, int &a, int &b)
     ^
2 warnings and 1 error generated.
```

`inline` függvények

A `#define` direktívával létrehozott makró-függvényeknek kellemetlen mellékhatásai lehetnek:

```
#define MUL(a, b) a*b
// ...
#define SIX 1+5
#define NINE 8+1
int val = MUL(SIX, NINE);
// val erteke 42 (nem 54)
```

Zárójelezéssel javíthatunk rajta, a deklaráció nem lett szebb, és még így is problémáink akadhatnak:

```
#define MAX(a, b) ((a)>(b)?(a):(b))
// ...
int a = 6;
int b = 5;
int val = MAX(++a, b);
// a es val erteke mar 8
```

Ezen kívül ha makrókat használunk, nincsenek típusdeklarációk, nem lehet használni a túlterhelést, az alapértelmezett értékeket, és egyebeket, amikről eddig nem volt szó. De szeretnénk, ha nem hajtana végre külön függvényhívást, ha a fordító is úgy gondolja, hogy ez kézenfekvő.

```
inline int max(int a, int b)
{
    return a > b ? a : b;
}
// ...
int a = 6;
int b = 5;
int val = max(++a, b);
// a es val erteke 7
```

A fordító nem veszi figyelembe az `inline` kulcsszót, ha

- a függvénynek címét vesszük, és azzal hívjuk meg (elvileg közvetlen meghívásnál még mindig befűzheti a hívás helyett)
- a függvény rekurzív
- a függvény túl bonyolult vagy nagy
- fordító hóbortból

Implicit `typedef` struktúrákra

A struktúrákra nem kell ezentúl `struct StrukturaNev` formában hivatkozni, simán `StrukturaNev` is megteszi.

```
struct Rect {
    int x, y;
    int width, height;
};

// OK
struct Rect rect1;

// C++-ban mar ez is OK
Rect rect2;
```

Objektum-orientált fejlesztés

Az OOP lényege, hogy objektumokkal dolgozunk. Objektum lehet pl. egy komplex szám, egy dinamikus tömb vagy egy gomb a képernyőn. Az objektumokon műveleteket lehet végezni. Az objektumok négy fontos elvet követnek:

Egységbezárás (encapsulation)
: Az objektum és a műveletei olyan formában jelennek meg a nyelvben, hogy szintaktikailag egységet alkotnak. Ez a gyakorlatban azt jelenti, hogy az osztályoknak (és a struktúráknak) tagfüggvényei lehetnek, amik megvalósítják a műveleteket.

Adatrejtés (information hiding)
: Az objektum belső állapota nem érhető el közvetlenül. Pl. a dinamikus tömb hosszához nem férhet hozzá mezőnhivatkozáson keresztül, csak művelettel kérheti le és állíthatja, így a dinamikus tömb megvalósíthatja az automatikus újraallokálást.

Öröklés (inheritance)
: Az objektumok osztálya van, az osztályok pedig más osztályokat bővíthetnek („örökölhetnek” – nem szeretem ezt a szóhasználatot, helyette rendre a „bővített” szót fogom használni). Pl. a `Bicycle` egy bicikli adatait tárolja el (pl. kerékátmérő és sebességek száma). Viszont biciklitől függően lehet, hogy további adatokat akarunk megadni, pl. városi kerékpár (`CityBike`) esetén a kosár űrtartalmát. Nem kell újra megírnunk azt a függvényt, ami a kerék fordulatszámából (a kerékátmérő ismeretében) a bicikli sebességét kiszámolja, hiszen az ugyanaz a városi kerékpárra, mint bármely más kerékpárra. A

leszármazott osztályok új mezőkkel és függvényekkel bővíthetik az ősosztályt, valamint az arra megjelölt függvényeket akár le is cserélhetik.

Behelyettesíthetőség

Az osztályok bővítései behelyettesíthetővé válnak oda, ahol a bővített osztályba („szülőosztályba”) tartozó objektum szerepelhet. Pl. ha egy függvény `Bicycle*` vagy `Bicycle&` típusú paramétert vár, ott rendre `CityBike*` és `CityBike&` típusú érték is megadható.

Típustámogatás (extra, csak C++-ra és egy pár más nyelvre igaz)

Az osztályok ugyanúgy működhetnek, mint a beépített típusok. Vagyis a `Complex` osztály ugyanúgy működhet, mint a `float` vagy a `double`.

Osztályok C++-ban

oo/point.h

```
#ifndef _POINT_H_
#define _POINT_H_

// a class ugyanaz, mint a struct, csak az alapertelmezett
// lathatosag private
class Point
{
    int x, y;
public:
    // konstruktor
    Point(int initX, int initY);
    // destruktor
    ~Point();

    // inline fuggvenyek a headerbe kerulnek
    inline int getX() { return x; }
    // deklaralhatjuk a fuggvenyen kivul is,
    // ajanlott, az attekintheto fuggvenylistahoz
    inline int getY();

    // nem inline fuggveny ne keruljon a headerbe
    Point offset(int offsetX, int offetY);
};

// igy jeloljuk, hogy a Point getY fuggvenyenek az
// implementacioja kovetkezik
inline int Point::getY() { return y; }

#endif
```

A **`class`** abban különbözik a **`struct`**-tól, hogy az alapértelmezett láthatóság **`private`**. Háromféle láthatósági szint létezik:

- **`private`**: csak az osztály látja
- **`protected`**: a bővített osztályok (gyermekosztályok) is látják
- **`public`**: mindenki látja

oo/point.cc

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

Point::Point(int initX, int initY)
{
    x = initX;
    y = initY;
    // azert, hoy kovetni tudjuk, mikor hivodik meg
    printf("Konstruktor: %d %d\n", x, y);
}

Point::~Point()
{
    // eleg hulye dolog, de a demonstracioert
    // mindent
    printf("Destruktor: %d %d\n", x, y);
}

Point Point::offset(int offsetX, int offsetY)
{
    return Point(x+offsetX, y+offsetY);
}
```

oo/first.cc

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

int main()
{
    // egyik konstruktorhivasi forma
    Point p1(1, 2);
    Point p2 = p1.offset(3, 4);

    // 1 2
    printf("%d %d\n", p1.getX(), p1.getY());
    // 4 6
    printf("%d %d\n", p2.getX(), p2.getY());

    {
        // masik konstruktorhivasi forma
        Point p3 = Point(5, 6);
        Point p4 = p3.offset(7, 8);
        printf("1..\n");
    }

    printf("2..\n");

    return 0;
}
```

oo/first.txt

```
Konstruktor: 1 2
Konstruktor: 4 6
1 2
4 6
Konstruktor: 5 6
Konstruktor: 12 14
1..
Destruktor: 12 14
Destruktor: 5 6
2..
Destruktor: 4 6
Destruktor: 1 2
```

Figyeljük meg, hogy a destruktorokat fordított sorrendben hívja meg, amikor az objektum kikerül a scope-ból.

Nagyon fontos, hogy a konstruktorokat nem hívjuk meg sehol. A fordító nem fog szólni miatta, de ettől helytelen gyakorlat. Inkább hozzunk létre egy függvényt, amit a konstruktor és a konstruktort hívó függvény meghív.

A destruktort sem hívjuk meg kézzel.

A `Point` osztálynak nincs paraméter nélküli, azaz alapértelmezett konstruktora. Ezért az alábbi nem fordul:

oo/nodefault.cc

```
#include "point.h"

int main()
{
    // az alabbi ket deklaracio ugyanazt a konstruktort
    // hivna meg, ha letezne
    Point p1;
    Point p2 = Point();
    // jegyezzuk azt is meg, hogy a default konstruktort
    // nem lehet Point p(); formaban meghivni
    return 0;
}
```

oo/nodefault.txt

```
$ clang -c nodefault.cc
nodefault.cc:7:8: error: no matching constructor for initialization of 'Point'
        Point p1;
              ^
./point.h:11:2: note: candidate constructor not viable: requires 2 arguments, but 0 were provided
        Point(int initX, int initY);
        ^
./point.h:6:7: note: candidate constructor (the implicit copy constructor) not viable: requires 1 argument, b
class Point
      ^
nodefault.cc:8:13: error: no matching constructor for initialization of 'Point'
        Point p2 = Point();
                   ^
./point.h:11:2: note: candidate constructor not viable: requires 2 arguments, but 0 were provided
        Point(int initX, int initY);
        ^
./point.h:6:7: note: candidate constructor (the implicit copy constructor) not viable: requires 1 argument, b
class Point
      ^
2 errors generated.
```

A `new` és a `delete` kulcsszó

A C++ a memóriaallokációt nyelvi szinten tartalmazza. Allokálni a new operátorral lehet, felszabadítani a `delete` illetve a `delete[]` operátorral. Az utóbbit abban az esetben kell alkalmazni, ha tömböt szeretnénk felszabadítani. A C-vel ellentétben a memóriaallokáció mindig típusos:

oo/new.cc

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

char* repeat(char c, int count)
{
    // figyeljuk meg, hogy a tomb meretenek nem csak konstanst, hanem
    // akarmilyen erteket megadhatunk
    char *buf = new char[count+1];
    memset(buf, c, count);
    buf[count] = 0;
    return buf;
}
int main()
{
    char *sleep = repeat('z', 10);
    printf("%s\n", sleep);
    // ne felejtsuk el felszabaditani!
    // ha tombot allokaltunk, akkor a delete[] operatort hasznaljuk
    delete[] sleep;

    Point *p = new Point(2, 2);
    // ha nem tomb, akkor siman delete
    delete p;

    return 0;
}
```

Fontos megjegyezni, hogy az így létrehozott objektumok destruktorait nem a scope-ból való kikerülésükkor, hanem a `delete`/`delete[]` meghívásakor hívja meg automatikusan.

A `malloc` hívással ellentétben ez meghívja a konstruktort is, még akkor is, ha tömböt allokálunk:

oo/counter.h

```
#ifndef _COUNTER_H_
#define _COUNTER_H_

// struct, class, majdnem teljesen mindegy
// demonstrativ cellal ez most legyen struct
struct Counter
{
    // mivel ez struct es nem class,
    // ezek publikusak - ennyi a kulonbseg
    Counter();
    ~Counter();

    int getValue();
private:
    const int value;
};

// a vizsgan azert irjatok class-t, mert
// structot ritkan hasznalunk osztalydeklaraciora

#endif
```

oo/counter.cc

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

static int count = 0;

Counter::Counter():
    // ez az inicializalo lista
    // bizonyos tipusokat, pl. const
    // mezoket csak ilyennel lehet
    value(++count)
{
    // ez nem mukodne:
    // value = ++count;
    printf("Counter::Counter() x %d\n", value);
}

Counter::~Counter()
{
    printf("Counter::~Counter() x %d\n", value);
}

int Counter::getValue()
{
    return value;
}
```

oo/second.cc

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

int main()
{
    printf("c = new Counter[4]:\n");
    // ezt akkor tehetjuk meg, ha az osztalnyak
    // van default (parameter nelkuli) konstruktora
    // a Point osztalyunkra nem mukodne
    Counter *c = new Counter[4];

    printf("delete[] c:\n");
    delete[] c;

    return 0;
}
```

Jól látható, hogy a tömböt visszafelé szabadítja fel:

oo/second.txt

```
```

```
c = new Counter[4]:
Counter::Counter() x 1
Counter::Counter() x 2
Counter::Counter() x 3
Counter::Counter() x 4
delete[] c:
Counter::~Counter() x 4
Counter::~Counter() x 3
Counter::~Counter() x 2
Counter::~Counter() x 1
```

A másoló konstruktor

A Point osztályunknak van egy rejtett konstruktora. A tagfüggvény *szignatúrája* valahogy így néz ki:

```
Point::Point(const Point &point);
```

Ezt a fordító automatikusan generálta: egyszerűen minden mezőt lemásol.

oo/third.cc

```
#include "point.h"

int main()
{
    Point p1(1, 2);
    // copy konstruktort hivja meg:
    Point p2(p1);
    // ez is:
    Point p3 = p1;
    // ez nem:
    p3 = p2;

    return 0;
};
```

A változódeklarációnál leírt = nem ugyanaz, mint az = művelet.

A másoló konstruktort testre is szabhatjuk, és ez néha szükséges is. Az alábbi `StringBuffer` osztály egy karaktertömböt allokál. Ha nem bíráljuk felül a másoló konstruktort, akkor egy az egyben átveszi a mutatót a másik objektumból, és amelyiknek a destruktorát másodjára hívja meg, az már felszabadított területre próbálja meghívni a **`delete`**-et.

oo/stringbuffer.h

```
#ifndef _STRINGBUFFER_H_
#define _STRINGBUFFER_H_

class StringBuffer
{
    char *buffer;
    int length;
public:
    StringBuffer(int length);
    StringBuffer(const StringBuffer &other);
    ~StringBuffer();

    inline char& at(int index) { return buffer[index]; }
    // a vegen a const kulcsszo azt jelzi,
    // hogy ezt a fuggvenyt meg lehet hivni
    // const StringBufferen is
    inline const char* whole() const
    {
        // ilyesmi nem ferne itt bele:
        // buffer = "foo";
        // mert itt most minden mezot
        // constnak lat
        return buffer;
    }

    inline int getLength() const { return length; }
};

#endif
```

oo/stringbuffer.cc

```
#include "stringbuffer.h"
#include <string.h>

StringBuffer::StringBuffer(int length)
{
    buffer = new char[length];
    // length = length-et nem irhatok...
    // az elso a tagvaltozora, a masodik
    // a lokalis valtozora hivatkozik
    this->length = length;
}

StringBuffer::~StringBuffer()
{
    delete[] buffer;
}

StringBuffer::StringBuffer(const StringBuffer &other)
{
    length = other.getLength();
    buffer = new char[length];
    memcpy(buffer, other.whole(), length);
}
```

A `this` mindig az aktuális objektumra mutat. Akkor használjuk, ha az aktuális objektumunkra kell egy mutató (meglepő módon), vagy egy olyan mezőt akarunk elérni, aminek a nevét elfedi egy lokális változó. Vagy azért, mert ilyen hülye szokásunk van.

Statikus deklarációk és névterek

A `static` kulcsszó függvényekben

A `static` kulcsszó C-ben és C++-ban egyaránt rendelkezésre áll függvényeken belül is:

```
void foo()
{
    static int count = 0;
    printf("%d\n", ++count);
}
```

Az ilyen statikus változók inicializálója csak egyszer hívódik meg, és hívásról hívásra megőrzi az értékét, pont mint egy globális változó. A globális változóval ellentétben viszont nem érhető el, csak a függvényen belülről.

A `static` kulcsszó osztályokban

Az osztályhoz kötött statikus mezők és függvények szintén globális változóként működnek, azaz nem kötöttek objektum példányhoz, a láthatóságuk pedig a `public`/`protected`/`private` elérési sémákkal szabályozhatók.

vis/stat.cc

```
#include <stdio.h>

class Counter
{
    static int nextValue;
    const int value;
public:
    inline Counter(): value(nextValue++)
    {
        printf("Counter::Counter() x %d\n", value);
    }
    ~Counter();

    static int getNextValue();
};

int Counter::nextValue = 1;

int Counter::getNextValue()
{
    return nextValue;
}

Counter::~Counter()
{
    printf("Counter::~Counter() x %d\n", value);
}

int main()
{
    Counter c[3];

    printf("Next value: %d\n", Counter::getNextValue());

    return 0;
}
```

vis/stat.txt

```
Counter::Counter() x 1
Counter::Counter() x 2
Counter::Counter() x 3
Next value: 4
Counter::~Counter() x 3
Counter::~Counter() x 2
Counter::~Counter() x 1
```

Figyeljük meg, hogy a statikus mező deklaráció egy `extern` változódeklarációval egyenértékű, vagyis a változót még külön egyszer deklarálni kell, ahol egyébként a kezdőérték is megadható. Megosztott komponenseknél ezt az osztály implementációját tartalmazó forrásba célszerű tenni.

A `friend` kulcsszó

A `friend` kulcsszóval hozzáférést adhatunk az osztály `protected` és `private` mezőihez más függvények és osztályok számára (ez esetben az osztály valamennyi tagfüggvénye hozzáférhet a mezőkhöz). A friend deklarációt az osztályon belül bárhova írhatjuk, minden esetben ugyanazt jelenti.

vis/friend.cc

```cpp
#include <stdio.h>

// van egy Foo nevu osztaly
// de itt meg nem definialhatjuk
class Foo;

class Quux
{
public:
    inline void modifyFoo(Foo &f);

    void action(Foo &f);
};

class Foo
{
    int a;
public:

    friend class Bar;
    friend void Quux::modifyFoo(Foo &f);
    friend int a(const Foo &f);
};

class Bar
{
public:
    int something(const Foo &f);
};

int Bar::something(const Foo &f)
{
    // friend nelkul nem fordulna
    return f.a*f.a;
}
inline void Quux::modifyFoo(Foo &f)
{
    f.a = 5;
}

void Quux::action(Foo &f)
{
    // mivel Foo-nak nincsenek
    // publikus elemei, es nem
    // vagyunk baratok, sokat
    // nem tudok itt kezdeni vele
}

int a(const Foo &f)
{
    return f.a;
}

int main()
{
    return 0;
}
```

Névterek

Más nyelvekben, ahol nincs lehetőség globális változókat vagy függvényeket deklarálni, előfordul, hogy az összetartozó konstansokat, függvényeket vagy globális változókat egy osztályban deklarálják statikus elemekként. Ami szép, mert az összetartozó elemeket egybefogja, és csúnya, mert mindezt osztályokkal teszi. C++-ban erre nincs szükség: használhatjuk a **namespace** kulcsszót.

vis/ns.cc

```cpp
#include <stdio.h>

int bar = 3;

namespace foo
{
    int bar = 4;

    void show(const char *s)
    {
        printf("(%d/%d) %s\n", ::bar, ++bar, s);
    }
};

int main()
{
    foo::bar = 5;
    for(int i=0; i<3; ++i)
        foo::show("Hello World!");

    return 0;
}
```

vis/ns.txt

```
(3/6) Hello World!
(3/7) Hello World!
(3/8) Hello World!
```

Figyeljük meg, hogy a show függvényben (illetve bármely deklarációban a névteren belül) a névtér saját változóit részesíti előnyben. A névtér nélküli globális változókat elérni a négyesponttal tudja. A négyesponttal kezdődő minősített neveknél a névtereket teljesen kézzel adjuk meg. Az alábbi példa – bár nem szép – demonstrálja, hol lehet ez szükséges:

vis/badns.cc

```cpp
#include <cstdio>

namespace foo
{
    int v = 0;

    namespace quux
    {
        int v = 1;
    }
}

namespace bar
{
    int v = 2;

    namespace foo
    {
        int v = 3;
    }

    void show()
    {
        // sima foo::quux::v-vel nem fordulna
        std::printf("%d %d %d\n", foo::v, ::foo::v, ::foo::quux::v);
    }
}

int main()
{
    std::printf("%d %d %d\n", foo::v, bar::v, bar::foo::v);
    bar::show();
    return 0;
}
```

vis/badns.txt

```
0 2 3
3 0 1
```

C++-ban minden standard C könyvtárnak megtalálható a C++ verziója, ami annyiban különbözik, hogy a rendszerhívások belekerülnek a std névtérbe. A header nevét a „h” elhagyásával és a „c” előtag hozzáadásával kapjuk meg.

Globális statikus változók és függvények (vagyis amik csak az adott fordítási egységben láthatók) üres névtérrel is deklarálhatók:

```
namespace
{
    // mas forditasi egysegbol nem lathato,
    // pont mint a static kulcsszoval
    int lokalis = 1;
}
```

Névterekben található változók és függvények semmilyen formában nem oszthatóak meg C modulokkal.

A `using` kulcsszó

A `using` kulcsszó használatával egyszerűsíthetjük a névterekben lévő elemek elérését. A deklaráció mindig a fordítási egységre vonatkozik, vagyis header esetén minden C++ forrásra érvényes lesz, amelyik a headert befűzi, ezért headerbe ilyet írni nem szép dolog. A `using` önmagában egyetlen nevet tesz elérhetővé a névtér megadása nélkül. A `using namespace` a névtér valamennyi elemét elérhetővé teszi a névtérmegjelölés (kvalifikáció) nélkül.

oo/using.cc

```
#include <iostream>

// A using direktivakat es deklaraciokat mindig
// irjuk elore a felreertesek elkerulese vegett.
// Az illusztracio kedveert itt kicsit szet lesznek
// szorva.

using std::cout;
// Innentol a cout kvalifikalatlan nevvel is elerheto
// (vagyis siman cout neven)

void helloWorld()
{
    // endl-hez meg kell a nevtermegjeloles
    cout << "Hello World!" << std::endl;
}

using namespace std;
// std valamennyi tagja elerheto kvalifikalatlanul

void helloNamespaces()
{
    cout << "Hello Namespaces!" << endl;
}

int main()
{
    helloWorld();
    helloNamespaces();
}
```

C++ I/O

Folyamok (streamek)

Ahogyan a C-s szöveges I/O legtöbbet használt része a formátum string, úgy a C++ I/O alapja a stream. A `stdout` C++-ban a `std::cout`, a `stderr` pedig a `std::cerr`.

Az elmaradhatatlan „Hello World!” alkalmazás:

io/hello.cc

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

A `std::endl` azon kívül, hogy kiír egy soremelés karaktert, üríti a buffert is: a karaktereket ugyanis a rendszer nem egyenként szokta a rendszer a képernyőre írni, hanem kötegelve (ez C-ben sem volt másképp, legfeljebb nem tudtatok róla :D).

A `std::cout`-hoz hasonlóan van `std::cin` is:

io/circumference.cc

```
// az alabbi define kell a Visual C++-hoz kapott C/C++ implementaciohoz,
// hogy legyen M_PI-nk
#define _USE_MATH_DEFINES
#include <cmath>
#include <iostream>

int main()
{
    double radius;
    std::cout << "Radius:" << std::endl;
    std::cin >> radius;
    std::cout << "Circumference: " << radius*2.0*M_PI << std::endl;
    return 0;
}
```

A program beolvassa a radius változóba a `stdin`ről egy tört értéket, majd kiszámolja a kör kerületét és kiírja.

Figyeljük meg, hogy a shift operátor nyilai a kimenetnél a kimenet felé mutatnak, a bemenetnél pedig onnan indulnak ki (hogy miért tud a shift operátor ilyen különösen viselkedni, arról később lesz szó).

Írás állományokba

Az `ifstream` és az `ofstream` osztályokkal történik. Szöveges formázott elérés:

io/distance.cc

```
#define _USE_MATH_DEFINES
#include <iostream>
#include <fstream>
#include <cmath>

const double R_EARTH = 6371.009; // km

class GeoPoint
{
    const double latitude;
    const double longitude;
public:
    inline GeoPoint(double latitude, double longitude):
        latitude(latitude),
        longitude(longitude)
    {
    }

    GeoPoint diff(const GeoPoint &other) const;

    inline double getLatitude() const { return latitude; }
    inline double getLongitude() const { return longitude; }
};

GeoPoint GeoPoint::diff(const GeoPoint &other) const
{
    return GeoPoint(latitude-other.latitude, longitude-other.longitude);
}

const double TO_RAD = M_PI/180.0;

double toRad(double degrees)
{
    return degrees * TO_RAD;
}

GeoPoint readGeoPoint(std::ifstream &s)
{
    double lt, ln;
    s >> lt >> ln;
    return GeoPoint(toRad(lt), toRad(ln));
}

double distanceInKm(const GeoPoint &p1, const GeoPoint &p2)
{
    GeoPoint dp = p2.diff(p1);
    double a =
        std::sin(dp.getLatitude()*0.5) * std::sin(dp.getLatitude()/2) +
            std::sin(dp.getLongitude()*0.5) * std::sin(dp.getLongitude()*0.5) *
            std::cos(p1.getLatitude()) * std::cos(p2.getLatitude());
    double c = 2 * std::atan2(std::sqrt(a), std::sqrt(1-a));
    return R_EARTH * c;
}

int main()
{
    std::ifstream in("coords.txt");
```

```
    // ha nem szoveges, akkor: in("coords.bin", std::ios::in | std::ios::binary);
    std::ofstream out("distances.txt", std::ios::out | std::ios::trunc);
    // ios::trunc: lenullazzuk a file hosszat
    if(!in)
    {
        std::cerr << "missing coords.txt" << std::endl;
        return -1;
    }

    if(!out)
    {
        std::cerr << "can't write distances.txt" << std::endl;
        return -1;
    }

    int line = 0;
    do {
        GeoPoint p1 = readGeoPoint(in);

        if(in.eof())
            break;

        GeoPoint p2 = readGeoPoint(in);

        out << "Distance for line " << ++line << ": " << distanceInKm(p1, p2) << " km" << std::endl;
    } while(!in.eof());

    in.close();
    out.close();

    return 0;
}
```

io/coords.txt

```
48.858222 2.2945 40.689167 -74.044444
51.5033 -0.1197 27.175 78.041944
```

io/distances.txt

```
Distance for line 1: 5837.43 km
Distance for line 2: 6882.73 km
```

Egyéb elérési módok (itt most ifstream és ofstream helyett a szülőosztályt, az istreamet és az ostreamet használom):

io/string.cc

```
#include <iostream>

int main()
{
    std::istream &in = std::cin;
    std::ostream &out = std::cout;

    char c;

    // 1 karakter
    in.get(c);
    out.put(c).put(c);

    out << std::endl;

    // binaris kimenet
    int word;
    in.read((char*)&word, sizeof(word));
    out.write((char*)&word, sizeof(word));

    out << std::endl;

    char line[256];
    in.getline(line, sizeof(line));
    out << line << std::endl;

    return 0;
}
```

io/string.txt

```
$ clang -o string -lstdc++ string.cc && echo zwordHello World! | ./string
zz
word
Hello World!
```

Mezők formázása

io/format.cc

```cpp
#define _USE_MATH_DEFINES
#include <cmath>
#include <iostream>
#include <iomanip>

using namespace std;

const float pi = M_PI;

int main()
{
    const int num = 123456;
    cout << "1. " << hex << num << endl;
    cout << "2. " << dec << num << endl;
    cout << "3. " << oct << num << endl;

    cout << "4. |" << pi << "|" << endl;
    cout << "5. |" << setprecision(3) << pi << "| |" << pi << "|" << endl;
    cout << "6. |" << setw(8) << setprecision(3) << pi*10.0f << "| |" << pi << "|" << endl;
    cout << "." << endl;
    // lehet a setw es a setprecision manipulatort hasznalni, ezek ekvivalensek
    // az alabbi hivasokkal:
    cout.precision(5);
    cout.width(12);
    // a width mindenre vonatkozik, nem csak a szamokra
    // de csak egy hivas erejeig
    cout << "." << "|" << endl;
    cout << "7. |" << pi << "|" << endl;
    cout << "8. |" << setw(8) << showpos << pi << "|" << endl;
    cout << "9. |" << setw(8) << noshowpos << pi << "|" << endl;

    cout << showbase;
    cout << "10. " << hex << num << endl;
    cout << "11. " << dec << num << endl;
    cout << "12. " << oct << num << endl;

    // termeszetesen ugyanilyen neven van manipulator is
    cout << noshowbase;

    cout << "13. " << 1.0e15 << endl;
    cout << "14. " << hex << 0xcafe12 << endl;

    cout << uppercase;

    cout << "15. " << 1.0e15 << endl;
    cout << "16. " << hex << 0xcafe12 << endl;

    cout << nouppercase;

    cout << "17. " << true << endl;

    cout << boolalpha;

    cout << "18. " << true << endl;

    cout << noboolalpha;

    cout << "19. |" << left << setw(8) << pi << "|" << endl;
    cout << "20. |" << setw(8) << pi << "|" << endl;
    cout << "21. |" << internal << showpos << setw(8) << pi << "|" << endl;

    return 0;
}
```

io/format.txt

```
```

```
1. 1e240
2. 123456
3. 361100
4. |3.14159|
5. |3.14| |3.14|
6. |    31.4| |3.14|
.
          .|
7. |3.1416|
8. | +3.1416|
9. |  3.1416|
10. 0x1e240
11. 123456
12. 0361100
13. 1e+15
14. cafe12
15. 1E+15
16. CAFE12
17. 1
18. true
19. |3.1416  |
20. |3.1416  |
21. |+ 3.1416|
```

Operátor túlterhelés

Operátorokat definiálhatunk olyan típusokra, amikre egyébként nincsenek. Nem változtathatjuk meg az eredeti működést (vagy legalábbis nagyon nem javaslom), nem változtathatjuk meg a precedenciát, nem hozhatunk létre teljesen új operátorokat (pl. **)

Az operátorok legalább egyik tagja osztály kell, hogy legyen.

opover/complex.cc

```
#include <iostream>

class Complex
{
    double re;
    double im;
public:
    inline Complex(): re(0), im(0) { }
    inline Complex(double real, double imaginary): re(real), im(imaginary) { }

    inline double real() const
    {
        return re;
    }

    inline double imaginary() const
    {
        return im;
    }

    // operatorok fuggveny szignaturajaban kulonosen fontos, hogy ahol csak lehet,
    // hasznaljuk a const kulcsszot

    // az operatorok a teljesseg igenye nelkul szerepelnek

    Complex operator+(const Complex &other) const
    {
        return Complex(re+other.re, im+other.im);
    }

    Complex operator*(const Complex &other) const
    {
        return Complex(re*other.re-im*other.im, re*other.im+im*other.re);
    }

    Complex operator*(double other) const
    {
        std::cout << "complex*double" << std::endl;
        return Complex(re*other, im*other);
    }

    // az operator=-re van egy implicit alapertelmezett implementacio,
    // ami egyszeruen lemasolja a mezoket
    // ez pontosan ugyanannyit tesz, tehat ilyen szempontbol felesleges
    Complex& operator=(const Complex &other)
    {
        if(this == &other)
            return *this;
        re = other.re;
        im = other.im;
```

```
        im = other.im;
        return *this;
    }

};

// barmelyik operatort definialhatjuk kivul is:
// a kulonbseg annyi, hogy az osztalyon belul definialt
// fuggvenyeknel a baloldali operandus mindig a this lesz
Complex operator*(double real, const Complex &complex)
{
    std::cout << "double*complex" << std::endl;
    return Complex(real*complex.real(), real*complex.imaginary());
    // ha hozzaadjuk az alabbi sort az osztalyhoz:
    // friend Complex operator*(double real, const Complex &complex);
    // akkor nezhetne igy is ki:
    // return Complex(real*complex.re, real*complex.im);
}

// itt pedig kulonosen hasznos, hogy nem kell feltetlen tagfuggvenynek lennie:
std::ostream& operator<<(std::ostream &stream, const Complex &complex)
{
    return stream << complex.real() << " + " << complex.imaginary() << "i";
}

int main()
{
    Complex c1(0.866, 0.5); // e^(pi*i/6) - pi/6 rad = 30 fok
    Complex c2(3, 7);
    Complex c3 = c1 + c2;
    Complex c4 = c3*2;
    Complex c5 = 2*c4;
    Complex c6;

    c6 = c5;

    std::cout << c1 << std::endl <<
        c2 << std::endl <<
        c3 << std::endl <<
        c4 << std::endl <<
        c5 << std::endl <<
        c6 << std::endl;

    return 0;
}
```

opover/complex.txt

```
complex*double
double*complex
0.866 + 0.5i
3 + 7i
3.866 + 7.5i
7.732 + 15i
15.464 + 30i
15.464 + 30i
```

Figyeljük meg, milyen sorrendben értékeli ki őket:

opover/order.cc

```cpp
#include <iostream>

class Counter
{
    static int current;
    const int value;
public:
    Counter(): value(current++) { }

    inline const Counter& operator+(const Counter &other) const
    {
        std::cout << "Counter(" << value << ") + Counter(" << other.value << ")" << std::endl;
        return *this;
    }

    inline const Counter& operator=(const Counter &other) const
    {
        std::cout << "Counter(" << value << ") = Counter(" << other.value << ")" << std::endl;
        return *this;
    }

    friend std::ostream& operator<<(std::ostream &stream, const Counter &other);
};

int Counter::current = 0;

std::ostream& operator<<(std::ostream &stream, const Counter &other)
{
    return stream << "Counter(" << other.value << ")";
}

int main()
{
    Counter c1, c2, c3, c4, c5;
    std::cout << c1 << ", " << c2 << "... " << c5 << "\n" << std::endl;
    c1 + c2 + c3 + c4 + c5;
    std::cout << std::endl;
    c1 = c2 = c3 = c4 = c5;

    return 0;
}
```

opover/order.txt

```
Counter(0), Counter(1)... Counter(4)

Counter(0) + Counter(1)
Counter(0) + Counter(2)
Counter(0) + Counter(3)
Counter(0) + Counter(4)

Counter(3) = Counter(4)
Counter(2) = Counter(3)
Counter(1) = Counter(2)
Counter(0) = Counter(1)
```

Kérdés még, hogy hogyan különböztetjük meg a prefix és a posztfix ++ operátort.

```
class A
{
    int val;
public:
    //
    // ...
    //

    // ++a
    A& operator++()
    {
        ++val;
        return *this;
    }

    // a++
    A operator++(int)
    {
        // elmentjuk az eredeti erteket
        A result(*this);
        ++val;
        return result;
    }

    //
    // ...
    //
};
```

A posztfix `++` és `--` operátorhoz meg kell követelnünk egy extra `int` paramétert, amit nem használunk fel.

Típuskonverzió, mint operátor

opover/simplecomplex.cc

```
#include <iostream>

struct SimpleComplex
{
    double re, im;

    // ez sunyi modon adatveszteshez vezet,
    // nem ajanlott ilyen formaban implementalni
    // tessek egy real() fuggvenyt hasznalni,
    // ahol egyertelmu, hogy az imaginarius
    // reszt elvesztettuk

    // illusztracionak viszont megfelelt :)
    operator double() const
    {
        return re;
    }
};

std::ostream& operator<<(std::ostream &stream, const SimpleComplex &c)
{
    return stream << c.re << " + " << c.im << "i";
}

int main()
{
    SimpleComplex c = { 7, 5 };

    double d = c;

    std::cout << c << std::endl;
    std::cout << d << std::endl;

    return 0;
}
```

opover/simplecomplex.txt

```
7 + 5i
7
```

Figyeljük meg, hogy a típuskonverziós operátoroknál hiányzik a visszatérési érték típusa, mert azt az `operator` kulcsszó után adjuk meg.

A függvényhívás operátor az egyetlen, amely tetszőleges számú paramétert vehet fel. A példában nem ez szerepel, de pl. egy 0 paraméteres függvényhívás operátort `double operator()()` szignatúrával tagfüggvényként, vagy `double operator()(Poly3 &p)` kívülről.

opover/poly3.cc

```
#include <iostream>

class Poly3
{
    double p[4];
public:
    inline Poly3(double a, double b, double c, double d)
    {
        p[3] = a;
        p[2] = b;
        p[1] = c;
        p[0] = d;
    }

    double operator()(double x) const
    {
        return p[3]*x*x*x + p[2]*x*x + p[1]*x + p[0];
    }

    double operator[](int index) const
    {
        return p[index];
    }
};

int main()
{
    Poly3 smoothstep(-2, 3, 0, 0);

    for(double d = 0; d <= 1.0; d += 0.125)
        std::cout << "smoothstep(" << d <<") = " << smoothstep(d) << std::endl;

    std::cout << std::endl;

    for(int i=0; i<4; ++i)
    {
        if(i > 0)
            std::cout << " + ";
        std::cout << smoothstep[i] << "*x^" << i;
    }

    std::cout << std::endl;

    return 0;
}
```

opover/poly3.txt

```
smoothstep(0) = 0
smoothstep(0.125) = 0.0429688
smoothstep(0.25) = 0.15625
smoothstep(0.375) = 0.316406
smoothstep(0.5) = 0.5
smoothstep(0.625) = 0.683594
smoothstep(0.75) = 0.84375
smoothstep(0.875) = 0.957031
smoothstep(1) = 1

0*x^0 + 0*x^1 + 3*x^2 + -2*x^3
```

Az következő operátorok nem terhelhetőek túl: `.` `::` `?:` `sizeof`. Minden más igen, beleértve néhány különös megoldást is:

opover/strange.cc

```cpp
#include <iostream>

using namespace std;

struct Pair
{
    inline Pair() { }
    inline Pair(int first, int second):
        first(first), second(second)
    {
    }

    int first, second;
};

class TwoLetters
{
    char letters[2];
    Pair p;

    inline TwoLetters(char first, char second)
    {
        letters[0] = first;
        letters[1] = second;
        buildPair();
    }

    void buildPair()
    {
        p = Pair(letters[0], letters[1]);
    }
public:
    inline TwoLetters(const char *s)
    {
        letters[0] = s[0];
        letters[1] = s[1];
        buildPair();
    }

    // barmilyen osztaly vagy struktura mutatoval visszaterhet
    // de mutatonak kell lennie
    inline const Pair* operator->() const
    {
        return &p;
    }

    // minden tagfuggvenynek van egy implicit parametere,
    // aminek az erteke a this
    // a tipusa pedig ennel az osztalynal TwoLetters* vagy
    // const TwoLetters* - ez esetben az elobbi

    // ez nem operator overload specifikus dolog, csak azzal
    // illusztraltam
    inline Pair* operator->()
    {
        return &p;
    }

    inline TwoLetters operator,(const TwoLetters &other) const
    {
        return TwoLetters(letters[0], other.letters[1]);
    }

    inline char get(int index) const
    {
        return letters[index];
    }
};

ostream& operator<<(ostream &stream, const TwoLetters &l)
{
    return stream << l.get(0) << l.get(1);
}

int main()
{
    TwoLetters ab("ab");
    TwoLetters cd("cd");

    cout << ab->first << " " << ab->second << endl;
    cout << (ab, cd) << endl;

    return 0;
}
```

opover/strange.txt

```
97 98
ad
```

Osztály bővítés C++-ban

Nézzünk először egy viszonylag egyszerű példát: dátum és idő. Az idő napot is megjelöl, a dátum viszont csak napot:

extend/date.h

```
#ifndef _DATE_H_
#define _DATE_H_

#include <iostream>
#include <iomanip>

struct Date
{
    int year, month, day;
};

inline std::ostream& operator<<(std::ostream &stream, const Date &d)
{
    return stream << d.year << "-" << std::setw(2) << std::setfill('0') << d.month <<
        "-" << std::setw(2) << std::setfill('0') << d.day;
}

struct DateTime: public Date
{
    int hour, minute, second;
};

inline std::ostream& operator<<(std::ostream &stream, const DateTime &dt)
{
    const Date &d(dt);
    return stream << d << " " << dt.hour << ":" <<
        std::setw(2) << std::setfill('0') << dt.minute << ":" <<
        std::setw(2) << std::setfill('0') << dt.second;
}

#endif
```

A szülőosztály lehet rejtett, ilyenkor a következőképpen alakul a mezőinek a láthatósága az új osztályban:

Szülőosztály: / Mező:	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	*nem érhető el*	*nem érhető el*	*nem érhető el*

A táblázat alapján elmondhatjuk, hogy a szülőosztály láthatósági megszorítása egy „láthatósági plafont” ad.

Virtuális metódusok

Az új osztály a bővítendő osztály függvényei helyett újakat definiálhat, de ebből a szempontból meg kell különböztetnünk a virtuális és a hagyományos függvényeket:

extend/person.h

```
#ifndef _PERSON_H_
#define _PERSON_H_

#include <string>
#include "date.h"

class Person
{
    std::string name;
    Date birthDate;
public:
    Person(const std::string &name, const Date &birthDate);

    const std::string& getName();
    const Date& getBirthDate();

    void print();
    virtual void printVirtual();
};

#endif
```

extend/person.cc

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

using namespace std;

Person::Person(const string &name, const Date &birthDate):
    name(name), birthDate(birthDate)
{
}

const string& Person::getName()
{
    return name;
}

const Date& Person::getBirthDate()
{
    return birthDate;
}

void Person::print()
{
    cout << name << " (" << birthDate << ")" << endl;
}

void Person::printVirtual()
{
    cout << name << " (" << birthDate << ")" << endl;
}
```

extend/employee.h

```
#ifndef _EMPLOYEE_H_
#define _EMPLOYEE_H_

#include "person.h"

class Employee: public Person
{
    Date hiringDate;
public:
    Employee(const std::string &name, const Date &birthDate, const Date &hiringDate);

    const Date& getHiringDate();

    void print();
    virtual void printVirtual();
};

#endif
```

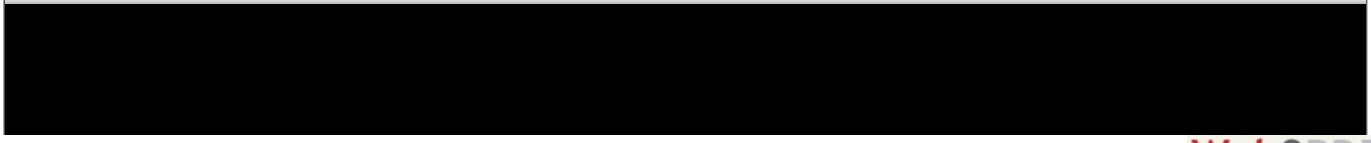
extend/employee.cc

```cpp
#include "employee.h"

#include <iostream>

using namespace std;

Employee::Employee(const std::string &name, const Date &birthDate, const Date &hiringDate):
    Person(name, birthDate),
    hiringDate(hiringDate)
{
}

const Date& Employee::getHiringDate()
{
    return hiringDate;
}

void Employee::print()
{
    Person::print();
    cout << "[" << hiringDate << "]" << endl;
}

void Employee::printVirtual()
{
    Person::print();
    cout << "[" << hiringDate << "]" << endl;
}
```

extend/extend.cc

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

using namespace std;

Date makeDate(int year, int month, int day)
{
    Date result;
    result.year = year;
    result.month = month;
    result.day = day;

    return result;
}

void fooPerson(Person p)
{
}

void fooPersonAt(Person &p)
{
}

void fooPersonPtr(Person *p)
{
}

void fooEmployee(Employee e)
{
}

void fooEmployeeAt(Employee &e)
{
}

void fooEmployeePtr(Employee *e)
{
}

int main()
{
    Employee e("John Doe", makeDate(1987, 8, 7), makeDate(2011, 7, 15));
    Person p("Jane Doe", makeDate(1991, 2, 3));

    cout << "Employee:" << endl;
    e.print();
    cout << "----" << endl;
    e.printVirtual();
    cout << "====" << endl;

    // p Peson tipusu referencia e-re
    Person &pe(e);

    cout << "Person&:" << endl;
    pe.print();
    cout << "----" << endl;
    pe.printVirtual();
    cout << "====" << endl;

    // copy konstruktort hiv,
    // a copy konstruktor const Person& tipusu parametert var,
    // vagyis nem lesz problemaja az Employee tipusu ertekkel
    fooPerson(e);
    // nem hiv copy konstruktort, es "e" Person "reszet" latja
    fooPersonAt(e);
    // nem hiv copy konstruktort, a mutato mogott ugyanazt latja,
    // mint az elozo a referenciaval
    fooPersonPtr(&e);

    // nyilvan nem megy:
    // fooEmployee(p);
    // fooEmployeeAt(p);
    // fooEmployeePtr(&p);

    return 0;
}
```

extend/extend.txt



```
Employee:
John Doe (1987-08-07)
[2011-07-15]
----
John Doe (1987-08-07)
[2011-07-15]
====
Person&:
John Doe (1987-08-07)
----
John Doe (1987-08-07)
[2011-07-15]
====
```

A virtuális függvény azt jelenti, hogy az osztályhoz tartozni fog egy virtuális függvény tábla, amely az osztály virtuális függvényeinek címét tartalmazza. Az objektumpéldány tartalmaz egy mutatót az osztály virtuális függvény táblájára. Normális esetben a tagfüggvényeket úgy hívja meg, hogy annak a címét fordítási időben a linker írja be. Virtuális függvény esetében viszont az objektum egy rejtett mezőjéből kiolvassa a virtuális függvény tábla címét, majd a tábla megfelelő elemét, amely tartalmazni fogja a meghívott függvény címét.

Így amikor `Person&`-ként hivatkoztunk az `Employee` objektumra, a **`print()`** függvény meghívásánál a `Person` osztály **`print()`** függvényét drótotza be még a linker, viszont a **`printVirtual()`** esetén a referencia mögötti objektum osztályától függ, hogy melyik függvényt hívja meg.

Ha valamelyik szülőosztályban egy függvény virtuálisként lett megjelölve, akkor annak minden gyermekosztálybeli megfelelője implicit módon virtuális lesz, akár kitesszük a **`virtual`** kulcsszót, akár nem. A kód megértését segíti, ha kitesszük.

Mivel a virtuális függvény alapvető tulajdonsága, hogy címe van, **`inline`** nem lehet.

Virtuális destruktor

A destruktor is lehet virtuális. Azt mondják ökölszabálynak: ha az osztálynak van legalább egy virtuális metódusa, akkor a destruktornak is virtuálisnak kell lennie. A valóság ennél árnyaltabb:

extend/vd.cc

```cpp
#include <iostream>

using namespace std;

// SD: statikus destruktor
class SDA
{
public:
    inline SDA() { cout << "\tSDA()" << endl; }
    inline ~SDA() { cout << "\t~SDA()" << endl; }
};

class SDB: public SDA
{
public:
    inline SDB() { cout << "\tSDB()" << endl; }
    inline ~SDB() { cout << "\t~SDB()" << endl; }
};

// VD: statikus destruktor
class VDA
{
public:
    inline VDA() { cout << "\tVDA()" << endl; }
    virtual ~VDA();
};

VDA::~VDA()
{
    cout << "\t~VDA()" << endl;
}

class VDB: public VDA
{
public:
    inline VDB() { cout << "\tVDB()" << endl; }
    virtual ~VDB();
};

VDB::~VDB()
{
    cout << "\t~VDB()" << endl;
}

int main()
{
    cout << "SDA *sda = new SDB():" << endl;
    SDA *sda = new SDB();
    cout << "delete sda:" << endl;
    delete sda;

    cout << "SDB *sdb = new SDB():" << endl;
    SDB *sdb = new SDB();
    cout << "delete sdb:" << endl;
    delete sdb;

    cout << "VDB *vdb = new VDB():" << endl;
    VDB *vdb = new VDB();
    cout << "delete vdb:" << endl;
    delete vdb;

    cout << "VDA *vda = new VDB():" << endl;
    VDA *vda = new VDB();
    cout << "delete vda:" << endl;
    delete vda;

    return 0;
}
```

extend/vd.txt

```
```

```
SDA *sda = new SDB():
    SDA()
    SDB()
delete sda:
    ~SDA()
SDB *sdb = new SDB():
    SDA()
    SDB()
delete sdb:
    ~SDB()
    ~SDA()
VDB *vdb = new VDB():
    VDA()
    VDB()
delete vdb:
    ~VDB()
    ~VDA()
VDA *vda = new VDB():
    VDA()
    VDB()
delete vda:
    ~VDB()
    ~VDA()
```

Figyeljük meg, hogy az első esetet leszámítva – helyesen – mindkét destruktor lefutott. Az első esetben a delete művelet során azonban, megnézte, mivel van dolga: `SDA*`, *meghívta* a destruktorát, és felszabadította a területet. Az utolsó esetben megnézte, mivel van dolga: `VDA*`, *meghívta* a destruktorát, és felszabadította a területet. Volt azonban egy fontos különbség: az előbbi esetben kifejezetten az `SDA::~SDA()` destruktort hívta meg, míg az utóbbi esetben a meghívandó függvény címét a virtuális metódustáblából nézte ki, így végül a `VDB::~VDB()` függvény lefuttatására került sor.

Absztrakt osztályok és metódusok

A virtuális függvények lehetnek absztrakt metódusok, ami azt jelenti, hogy az adott osztályban nem tartozik hozzá implementáció.

extend/abstract.cc

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

using namespace std;

class Stringifiable
{
public:
    // absztrakt:
    virtual string stringify() const = 0;
};

class Number: public Stringifiable
{
    const int value;
public:
    inline Number(int value): value(value) { }

    virtual string stringify() const;
};

string Number::stringify() const
{
    stringstream s;
    s << value;
    return s.str();
}

class Text: public Stringifiable
{
    const string value;
public:
    inline Text(const string &value): value(value) { }

    virtual string stringify() const;
};

string Text::stringify() const
{
    return value;
}

void print(const Stringifiable &s)
{
    cout << s.stringify() << endl;
}

int main()
{
    Number num(5);
    Text txt("foo");
#ifdef FAILCOMPILATION
    // absztrakt osztalyt nem lehet
    // peldanyositani

    // ezt a reszt csak a
    // -D FAILCOMPILATION
    // parameter hozzaadasaval
    // veszi figyelembe a fordito

    // se igy:
    Stringifiable s;
    // es persze igy sem:
    new Stringifiable();
#endif

    print(num);
    print(txt);

    return 0;
}
```

extend/abstract.txt

```
5
foo
```

Ha az osztálynak van absztrakt függvénye, nem példányosítható:

extend/abstractInstance.txt

```
$ clang -lstdc++ -D FAILCOMPILATION abstract.cc
abstract.cc:63:16: error: variable type 'Stringifiable' is an abstract class
        Stringifiable s;
                      ^
abstract.cc:11:17: note: unimplemented pure virtual method 'stringify' in 'Stringifiable'
        virtual string stringify() const = 0;
                       ^
abstract.cc:65:6: error: allocating an object of abstract class type 'Stringifiable'
        new Stringifiable();
            ^
2 errors generated.
```

Többszörös öröklés

A `Stingifiable` megfelelő azoknak az osztályoknak, amelyek értelmesen szöveggé alakíthatóak, de mi van azokkal, amik egész számmá is alakíthatóak? A felvázolt `Number` osztálynál egyértelmű, mit kell tenni, de egy komplex típusnál vehetnénk pl. a valós rész egészrészét. De mindkettőről tudjuk, hogy kiválóan átalakíthatóak szöveggé is. Itt jön be a többszörös öröklés:

extend/multi.cc

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

using namespace std;

class Stringifiable
{
public:
    // absztrakt:
    virtual string stringify() const = 0;
};

class Intifiable
{
public:
    virtual int intify() const = 0;
};

class Number: public Stringifiable, public Intifiable
{
    const int value;
public:
    inline Number(int value): value(value) { }

    virtual string stringify() const;
    virtual int intify() const;
};

string Number::stringify() const
{
    stringstream s;
    s << value;
    return s.str();
}

int Number::intify() const
{
    return value;
}

void print(const Stringifiable &s)
{
    cout << s.stringify() << endl;
}

int sum(const Intifiable &left, const Intifiable &right)
{
    return left.intify() + right.intify();
}

int main()
{
    Number num(5);
    Number six(6);

    print(num);
    print(Number(sum(num, six)));

    return 0;
}
```

A többszörös öröklés nem csak absztrakt szülőosztályokra vonatkozik, teljes értékű osztályok is lehetnek szülőosztályok többszörös öröklésnél is.

Virtuális szülőosztályok

Nézzünk egy másik példát, ahol egy klub csoportvezetőit és csoporttagjait tartjuk számon. Azonban a klub vezetőjét leszámítva mindenki tagja a csoportvezetők csoportjának, vagyis egyszerre csoporttagok és csoportvezetők.

extend/wrongHierarchy.h

```
#ifndef _WRONG_HIERARCHY_H_
#define _WRONG_HIERARCHY_H_

#include <string>

class Member
{
    std::string name;
public:
    inline Member(const std::string &name): name(name)
    {
    }

    virtual void setName(const std::string &newName);

    virtual std::string getName() const;
};

class TeamLeader: public Member
{
    int numTeamMembers;
public:
    inline TeamLeader(const std::string &name, int numTeamMembers):
        Member(name),
        numTeamMembers(numTeamMembers)
    {
    }

    inline void setNumTeamMembers(int newNum)
    {
        numTeamMembers = newNum;
    }

    inline int getNumTeamMembers() const
    {
        return numTeamMembers;
    }
};

class TeamMember: public Member
{
    TeamLeader *teamLeader;
public:
    inline TeamMember(const std::string &name):
        Member(name),
        teamLeader(0)
    {
    }

    inline TeamLeader* getTeamLeader() const
    {
        return teamLeader;
    }

    inline void setTeamLeader(TeamLeader *newTeamLeader)
    {
        teamLeader = newTeamLeader;
    }
};

class MiddleTeamLeader: public TeamLeader, public TeamMember
{
public:
    inline MiddleTeamLeader(const std::string &name, int numTeamMembers):
        TeamMember(name),
        TeamLeader(name, numTeamMembers)
    {
    }
};

#endif
```

Első ránézésre jónak néz ki. Próbáljuk ki:

extend/wrongHierarchy.cc

```cpp
#include <iostream>
#include "wrongHierarchy.h"

using namespace std;

void Member::setName(const std::string &newName)
{
    name = newName;
}

std::string Member::getName() const
{
    return name;
}

int main()
{
    MiddleTeamLeader mtl("Jane Doe", 9);
    TeamLeader &tl(mtl);
    TeamMember &tm(mtl);

    tl.setName("Summer Smith");
    cout << tl.getName() << endl;
    cout << mtl.getName() << endl;

    tm.setName("Beth Sanchez");
    cout << tm.getName() << endl;
    cout << mtl.getName() << endl;

    return 0;
}
```

extend/wrongHierarchy.txt

```
$ clang -lstdc++ wrongHierarchy.cc
wrongHierarchy.cc:24:14: error: non-static member 'getName' found in multiple base-class subobjects of type '
    class MiddleTeamLeader -> class TeamLeader -> class Member
    class MiddleTeamLeader -> class TeamMember -> class Member
        cout << mtl.getName() << endl;
                    ^
wrongHierarchy.cc:11:21: note: member found by ambiguous name lookup
std::string Member::getName() const
                    ^
wrongHierarchy.cc:28:14: error: non-static member 'getName' found in multiple base-class subobjects of type '
    class MiddleTeamLeader -> class TeamLeader -> class Member
    class MiddleTeamLeader -> class TeamMember -> class Member
        cout << mtl.getName() << endl;
                    ^
wrongHierarchy.cc:11:21: note: member found by ambiguous name lookup
std::string Member::getName() const
                    ^
2 errors generated.
```

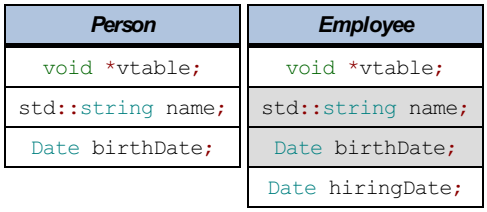
A fordítás sajnos elszállt. Nézzük meg, hogyan néznek ki az egyes osztályok példányai a memóriában, de először csak egy egyszerű példára, a már felvázolt `Date`-re:

Date
`int year;`
`int month;`
`int day;`

DateTime
`int year;`
`int month;`
`int day;`
`int hour;`
`int minute;`
`int second;`

Ha egy `DateTime` típusú objektumra úgy tekintünk, mint egy `Date` típusúra, akkor azt látjuk, amire számítunk: évet, hónapot, napot, egy-egy `int`-ként, egymás után.

Nézzük az egyszerű, virtuális metódusos öröklést a `Person` és az `Employee` osztályokra:



A `vtable` természetesen a virtuális metódustábla mutató, minden osztály példányaiban az osztály (nem az objektum: az pazarlás lenne) saját táblájára mutat.

A többszörös öröklés tényleges megvalósítása fordítófüggő, ezért nem részletezzük. A lényeg, hogy a `TeamLeader` és a `TeamMember` egyenként már tartalmazza a `Member` mezőit, és ezzel a bővítési mechanizmussal nincs esély arra, hogy a `MiddleTeamLeader` szülőosztálybeli függvényei megfelelő objektumot lássanak, miközben nincs két `Member` szülő beágyazva.

A megoldás a virtuális szülőosztály. A segítségével a `TeamLeader` és a `TeamMember` „részei” a `MiddleTeamLeader` osztálynak közös `Member`t látnak.

extend/virtualInheritance.h

```
#ifndef _VIRTUAL_INHERITANCE_H_
#define _VIRTUAL_INHERITANCE_H_

#include <string>

class Member
{
    std::string name;
public:
    inline Member(const std::string &name): name(name)
    {
    }

    virtual void setName(const std::string &newName);

    virtual std::string getName() const;
};

class TeamLeader: public virtual Member
{
    int numTeamMembers;
public:
    inline TeamLeader(const std::string &name, int numTeamMembers):
        Member(name),
        numTeamMembers(numTeamMembers)
    {
    }

    inline void setNumTeamMembers(int newNum)
    {
        numTeamMembers = newNum;
    }

    inline int getNumTeamMembers() const
    {
        return numTeamMembers;
    }
};

class TeamMember: public virtual Member
{
    TeamLeader *teamLeader;
public:
    inline TeamMember(const std::string &name):
        Member(name),
        teamLeader(0)
    {
    }

    inline TeamLeader* getTeamLeader() const
    {
        return teamLeader;
    }

    inline void setTeamLeader(TeamLeader *newTeamLeader)
    {
        teamLeader = newTeamLeader;
    }
};

class MiddleTeamLeader: public TeamLeader, public TeamMember
{
public:
    inline MiddleTeamLeader(const std::string &name, int numTeamMembers):
        Member(name),
        TeamMember(name),
        TeamLeader(name, numTeamMembers)
    {
    }
};

#endif
```

extend/virtualInheritance.cc

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

using namespace std;

void Member::setName(const std::string &newName)
{
    name = newName;
}

std::string Member::getName() const
{
    return name;
}

int main()
{
    MiddleTeamLeader mtl("Jane Doe", 9);
    TeamLeader &tl(mtl);
    TeamMember &tm(mtl);

    tl.setName("Summer Smith");
    cout << tl.getName() << endl;
    cout << mtl.getName() << endl;

    tm.setName("Beth Sanchez");
    cout << tm.getName() << endl;
    cout << mtl.getName() << endl;

    return 0;
}
```

extend/virtualInheritance.txt

```
Summer Smith
Summer Smith
Beth Sanchez
Beth Sanchez
```

Adatmodellezési elvek

Ha a gyakorlatban virtuális ősosztályokat kell használnod, esélyes, hogy valamit nagyon rosszul modelleztél le.

A gyakorlatban fontos érteni a különbséget az *„X egy Y”* és az *„X része Y”* kijelentés között, mert nem lesz mindig egyértelmű. Az első öröklést (a téglalap egy síkidom, van pl. területe), a második objektum kompozíciót jelent (pl. a téglalap egy pontból és egy méretből áll, a téglalap viszont nem pont és nem méret).

Típuskonverziók

Konverziók megvalósítása

Mindenről volt már szó, összefoglalva:

conversion/conversion.cc

```cpp
#include <iostream>
#include <cstdlib>

using namespace std;

class Number
{
    int value;
public:
    inline Number(int value): value(value)
    {
        cout << "\tNumber(int value = " << value << ")" << endl;
    }

    explicit inline Number(const char *num): value(atoi(num))
    {
        cout << "\tNumber(const char* value = \"" << value << "\")" << endl;
    }

    inline operator int() const
    {
        cout << "\tNumber(" << value << ")::operator int() const" << endl;
        return value;
    }

    inline int operator=(int other)
    {
        cout << "\tint Number(" << value << ")::operator=(int other = " << other << ")" << endl;
        value = other;
        return value;
    }
};

void print(Number n)
{
    int i = n;
    cout << "print(" << i << ")" << endl;
}

void printRef(const Number &n)
{
    int i = n;
    cout << "printRef(" << i << ")" << endl;
}

int main()
{
    // az alabbi ketto ekvivalens
    cout << "Constructor calls:" << endl;
    Number n1 = 1;
    Number n2(2);

    // az explicit kulcsszo miatt ez nem:
// Number n3 = "3";
    // de ez igen:
    Number n4("4");

    // szinten explicit miatt ezek nem:
// print("5");
// printRef("5");
    // de ez igen:
    print(Number("5"));
    printRef(Number("5"));

    print(6);
    print(Number(6));
    // ha a referencia nem lenne konstans, ezt nem engedne:
    printRef(7);
    printRef(Number(7));

    cout << "Assignment operator calls:" << endl;
    n1 = n2 = n4 = 8;

    cout << "Cast operator call:" << endl;
    cout << n1 << endl;

    return 0;
}
```

conversion/conversion.txt

```
Constructor calls:
    Number(int value = 1)
    Number(int value = 2)
    Number(const char* value = "4")
    Number(const char* value = "5")
    Number(5)::operator int() const
print(5)
    Number(const char* value = "5")
    Number(5)::operator int() const
printRef(5)
    Number(int value = 6)
    Number(6)::operator int() const
print(6)
    Number(int value = 6)
    Number(6)::operator int() const
print(6)
    Number(int value = 7)
    Number(7)::operator int() const
printRef(7)
    Number(int value = 7)
    Number(7)::operator int() const
printRef(7)
Assignment operator calls:
    int Number(4)::operator=(int other = 8)
    int Number(2)::operator=(int other = 8)
    int Number(1)::operator=(int other = 8)
Cast operator call:
    Number(8)::operator int() const
8
```

Típuskonverziós operátorok

Nézzük a klasszikus C-ben megismert típuskonverziót:

conversion/miscasting.cc

```
#include <iostream>

using namespace std;

struct A
{
    int value;

    inline A(int v): value(v) { }
};

struct B
{
    float value;
};

int main()
{
    A a(3);
    B *b = (B*) &a;

    cout << b->value << endl;

    return 0;
}
```

Egy szó nélkül lefordítja:

conversion/miscasting.txt

```
4.2039e-45
```

Ez nem probléma, ha tudja az ember, hogy mit csinál, de jó lenne, ha meg tudnánk tenni ezt úgy is, hogy figyelmeztessen a rendszer:

conversion/castingerror.cc

```cpp
#include <iostream>

using namespace std;

struct A
{
    int value;

    inline A(int v): value(v) { }
};

struct B
{
    float value;
};

int main()
{
    A a(3);
    B *b = static_cast<B*>(&a);

    cout << b->value << endl;

    return 0;
}
```

conversion/castingerror.txt

```
$ clang -lstdc++ castingerror.cc
castingerror.cc:20:9: error: static_cast from 'A *' to 'B *' is not allowed
        B *b = static_cast<B*>(&a);
               ^~~~~~~~~~~~~~~~~~~
1 error generated.
```

Nem engedi, mert ez két teljesen különböző osztály.

conversion/staticcast.cc

```cpp
#include <iostream>

using namespace std;

struct A
{
    int i;

    inline A(int v): i(v) { }
};

struct B: public A
{
    float f;

    inline B(int i, float f): A(i), f(f) { }
};

int main()
{
    B b(3, 5.2f);
    A *ap = &b;
    B *bp = static_cast<B*>(ap);

    cout << bp->f << endl;

    return 0;
}
```

conversion/staticcast.txt

```
5.2
```

Sajnos a `static_cast` engedi a bővítést is, cserébe viszont gyors, és nem dobja meg a program méretét:

conversion/downcast.cc

```
#include <iostream>

using namespace std;

struct A
{
    int i;

    inline A(int v): i(v) { }
};

struct B: public A
{
    double f;

    inline B(int i, double f): A(i), f(f) { }
};

int main()
{
    A a(3);
    B *bp = static_cast<B*>(&a);

    // allokalatlan teruletrol olvassa ki az erteket,
    // mert bp helyen csak A van
    cout << bp->f << endl;

    return 0;
}
```

conversion/downcast.txt

```
6.95322e-310
```

A megoldás erre az esetre a `dynamic_cast` lenne, amelyhez be kell kapcsolni a fordítóban a futási idejű típusinformációt (RTTI, run-time type information). A példám sajnos nem úgy fut le, mint amit vártam: elvileg null pointert kellene visszaadnia, vagy kivételt dobnia (erről később), ehhez képest egyik sem történik meg: „Segmentation fault” üzenetet kapok.

conversion/dynamiccast.cc

```
#include <iostream>

using namespace std;

struct A
{
    int i;

    inline A(int v): i(v) { }
    // a dynamic_cast hasznalatahoz kell,
    // hogy legyen legalabb egy virtualis
    // fuggveny
    virtual ~A();
};

A::~A()
{
}

struct B: public A
{
    float f;

    inline B(int i, float f): A(i), f(f) { }
};

int main()
{
    A a(3);
    B *bp = dynamic_cast<B*>(&a);
    A *ap = bp;

    // allokalatlan teruletrol olvassa ki az erteket,
    // mert bp helyen csak A van
    cout << bp->f << endl;

    return 0;
}
```

A `dynamic_cast` használata kerülendő, mert lassú, és megdobja a program méretét.

A `const_cast` segítségével a `const` kulcsszótól szabadulhatunk meg. Használjuk módjával, csak nagyon indokolt esetben.

conversion/constcast.cc

```
#include <iostream>
#include <cstring>

using namespace std;

int main()
{
    char backing[256];

    const char *cs = backing;
    char *s = const_cast<char*>(cs);

    strcpy(s, "Hello World!");

    cout << cs << endl;

    return 0;
}
```

Megmaradt a `reinterpret_cast`, ami bármilyen mutatót bármilyen mutatóra vált kérdés és megfontolás nélkül, valamint akár `int` típusú értéket is hajlandó mutatóként értelmezni (ha elég széles a platformon: 64 bites fordítás esetén már nem `int`, hanem `long long` szélességű a mutató).

conversion/reinterpretcast.cc

```
#include <iostream>

using namespace std;

int main()
{
    int num = 0x40490fdb;
    float f = *reinterpret_cast<float*>(&num);

    cout << f << endl;

    return 0;
}
```

conversion/reinterpretcast.txt

```
3.14159
```

Kivételkezelés

A hibakezelésre C-ben nincs igazán jó megoldás. Feláldozhatjuk a függvények visszatérési értékét, és visszatérhetnek hibakóddal, de nagyon megbonyolítja a kódot:

```
int calculate(Value **result)
{
    // f1, f2, f3 valamit kiszamol, ami el is szallhat
    // 0-val valo visszateres a siker
    // ossze akarjuk vonni oket
    *result = new Value();
    Value *merged = *result;
    Value value;
    int error;
    error = f1(&value);
    if(error)
    {
        delete *result;
        *result = 0;
        return error;
    }

    merged->merge(value);

    error = f2(&value);
    if(error)
    {
        delete *result;
        *result = 0;
        return error;
    }

    merged->merge(value);

    error = f3(&value);
    if(error)
    {
        delete *result;
        *result = 0;
        return error;
    }

    merged->merge(value);

    return 0;
}
```

Erre alkották meg a kivételkezelést. Először maradjunk meg csak a hibakódoknál, de most már ne áldozzuk fel a visszatérési értéket.

```
Value* calculate()
{
    Value *result = new Value();

    try
    {
        result->merge(f1());
        result->merge(f2());
        result->merge(f3());
    }
    catch(int error)
    {
        delete result;
        throw error;
    }

    return result;
}
```

Sokkal letisztultabb, átláthatóbb és egyszerűbb. A `throw` segítségével dobhatunk valamilyen értéket, amit a `try catch` ágával kaphatunk el. Nem kell feláldoznunk a függvény visszatérési értékét sem.

Dobni bármilyen típust lehet, minden típusra írhatunk külön `catch` ágat. Az ágak közül mindig a legelső for lefutni, ami illeszkedik az adott értékre, vagyis ha el szeretnénk kapni szülő- és gyermekosztályhoz is tartozó példányt, akkor az utóbbit kell előre írni, egyébként a szülőosztály példányát elkapó ág fog lefutni mindig.

Kérdés még, hogy el lehet-e kapni azt, amiről nem tudjuk, hogy micsoda, vagyis „minden mást.”

trycatch/catchall.cc

```
#include <iostream>

using namespace std;

class A
{
    const char * const message;
```

```
public:
    A(const char *message);

    const char* getMessage() const;
};

A::A(const char *message): message(message)
{
}

const char* A::getMessage() const
{
    return message;
}

typedef void (*SimpleCall)();

void f1()
{
    throw "Hello World!";
}

void f2()
{
    throw 42;
}

void f3()
{
    throw A("foobar");
}

void t(SimpleCall f)
{
    try
    {
        f();
    }
    catch(const char *s)
    {
        cout << "t caught string: " << s << endl;
        throw s;
    }
    catch(int errorCode)
    {
        cout << "t caught error code: " << errorCode << endl;
    }
    catch(...)
    {
        cout << "t caught something else..." << endl;
        // barmi is volt, dobjuk tovabb
        // igy, ertek nelkul csak catch(...) blokkban szerepelhet
        throw;
    }
}

int main()
{
    SimpleCall calls[] = { f1, f2, f3 };
    int numCalls = sizeof(calls)/sizeof(*calls);
    for(int i=0; i<numCalls; ++i)
    {
        SimpleCall f = calls[i];
        try
        {
            t(f);
        }
        catch(const char *s)
        {
            cout << "main caught string: " << s << endl;
        }
        catch(int errorCode)
        {
            cout << "main caught error code: " << errorCode << endl;
        }
        catch(const A &a)
        {
            cout << "main caught A: " << a.getMessage() << endl;
        }
    }
}
```

trycatch/catchall.txt

```
t caught string: Hello World!
main caught string: Hello World!
t caught error code: 42
t caught something else...
main caught A: foobar
```

A dobott típusok megjelölése

A függvény szignatúrában megjelölhetjük, hogy milyen típusú értékeket dobhat a függvény. Ez nem jelenti azt, hogy a belső, jelöletlen függvények nem dobhatnak mást, vagy nagyon csúnya vége lesz:

trycatch/funthrow.cc

```cpp
#include <iostream>

using namespace std;

int f() throw (int)
{
    throw "foo";
}

void unexpectedHandler()
{
    cout << "unexpected" << endl;
}

int main()
{
    set_unexpected(unexpectedHandler);

    try
    {
        f();
    }
    catch(const char *s)
    {
        cout << s << endl;
    }
    catch(int i)
    {
        cout << i << endl;
    }
    catch(...)
    {
        cout << "Other" << endl;
    }

    return 0;
}
```

trycatch/funthrow.txt

```
unexpected
libc++abi.dylib: terminate called throwing an exception
Abort trap: 6
```

A helyzet akkor is ugyanez, ha nem közvetlenül az `f()` függvény dobja a kivételt, hanem valamelyik másik, amelyik nem is deklarálta a dobott típusokat.

Persze tekinthetjük égbekiáltó hibának, ha olyan kivételt dobnak, amire nem vagyunk egyáltalán felkészülve. Ez néhány esettől eltekintve követendő gyakorlatnak is tekinthető. Ha mégis szeretnénk kultúráltan, egy `catch` ágban kezelni, akkor használhatunk egy egyéni `unexpected` függvényt és a szabvány C++ könyvtár `std::bad_exception` osztályát:

trycatch/badthrow.cc

```cpp
#include <iostream>
// bad_exception:
#include <exception>

using namespace std;

int g()
{
    throw "foo";
}

int f() throw (int, bad_exception)
{
    return g();
}

void unexpectedHandler()
{
    cout << "unexpected" << endl;
    // igy lesz belole bad_exception
    throw;
}

int main()
{
    set_unexpected(unexpectedHandler);

    try
    {
        f();
    }
    catch(const char *s)
    {
        cout << s << endl;
    }
    catch(int i)
    {
        cout << i << endl;
    }
    catch(bad_exception e)
    {
        cout << e.what() << endl;
    }
    catch(...)
    {
        cout << "Other" << endl;
    }

    return 0;
}
```

trycatch/badthrow.txt

```
unexpected
std::bad_exception
```

A szabvány C++ könyvtár kivétel osztályai

A szabvány C++ könyvtár deklarál pár kivétel osztályt. Nézzük ezeket header szerint.

<exception>

exception

A szabvány C++ kivételek ősosztálya. Természetesen ebből származtatva saját kivétel osztályokat is létrehozhatunk. Ha ilyet kapunk el, az alábbi függvényt használhatjuk, vagy ha saját osztályt hozunk létre, akkor az alábbi függvényt érdemes felülcsapni:

```cpp
virtual const char* what() const throw()
```

Amíg ezen a szinten kapjuk el a kivételeket, ez a legtöbb, amit kiszedhetünk belőle: egy stringet. Természetesen ha saját osztályt készítünk, akkor további paraméterek átadására is lehetőség van.

Ha saját osztályt készítünk a `what()` függvény felülcsapásán kívül érdemes copy konstruktort (ha a sima másolás nem felel meg) és esetleg `operator=` függvényt írnunk.

bad_exception

Ezt már ismerjük.

<new>

bad_alloc
: A `new` operátor dobja, ha nem sikerül az allokációt végrehajtania.

<stdexcept>

runtime_error
: Csak futás közben észrevehető hibák esetén dobja, pl. mert nem megfelelő értékek jöttek ki (pl. nullával való osztás). A gyermekosztályai: `range_error`, `overflow_error`, `underflow_error`.

logic_error
: A program logikában történő hibák esetén dobja. A gyermekosztályai: `invalid_argument`, `out_of_range`, `length_error`, `domain_error`.

Sablonok

A konstansoknál és az `inline` függvényeknél megpróbáltunk megszabadulni a preprocesszoros megoldásoktól. Van azonban még egy terület, amit ezek nem képesek lefedni, amit a preprocesszoros direktíva igen: a típusfüggetlenség.

Vegyük az alábbi kódrészletet:

```
#define MIN(a, b) ((a) < (b) ? (a) : (b))
```

Ez ugyanúgy fog működni `int`-re, `float`-ra, `long long`-ra, vagy bármilyen osztályra, amely az `operator<`-t definiálja. Erre megoldást nyújtanak a C++ sablonok:

templates/minmax.cc

```
#include <iostream>

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

// a template kulcsszo parameterenel a class szo
// ugyanazt jelenti, mint a typename, T tetszoleges
// tipus lehet, nem csak class
template<class T> T max(const T &a, const T &b)
{
    return a < b ? b : a;
}

int main()
{
    std::cout << min(5, 3)/2 << std::endl;
    std::cout << max(5, 3)/2 << std::endl;
    std::cout << max<float>(5, 3)/2 << std::endl;

    return 0;
}
```

templates/minmax.txt

```
1
2
2.5
```

A sablon paramétereket a fordító kikövetkeztetheti, adhatunk alapértelmezett paramétereket (a következőben példában látható lesz), de megadhatjuk a paraméterek értékét közvetlenül is. Ez néhány esetben szükséges is lehet, ha pl. csak a függvény visszatérési értékének típusa függ a paramétertől, a fordító nemigen tudja kikövetkeztetni ilyen esetben.

Nagyon fontos tudni, hogy a sablon függvények nem igazi függvények, önmagukban nem fordíthatóak. Ha sablonokat megosztva akarunk használni, teljes egészében a header állományba kell írni.

A sablonok megalkotásakor először a `class` kulcsszót használták a típusok megjelöléséhez, hogy ne kelljen új kulcsszót bevezetni, de végül a szabványosítás során az egyértelműbb `typename` szó mellett döntöttek (amit egyéb okokból egyébként is be kellett vezetni), a `class` ilyen felhasználását viszont a kompatibilitás megőrzése végett meghagyták.

Nézzük meg, hogyan néz ki egy sablon osztály deklarációja:

templates/markrel.cc

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

using namespace std;
```

```
using namespace std;

template<typename T, int MAX_MARKS=256> class MarkRelease
{
    char *objectBuffer;
    int size;
    int count;
    int marks[MAX_MARKS];
    int numMarks;
public:
    MarkRelease(int arraySize);
    ~MarkRelease()
    {
        delete[] objectBuffer;
    }

    void mark()
    {
        if(numMarks >= MAX_MARKS)
            throw logic_error("Too many marks");
        marks[numMarks++] = count;
    }

    void release()
    {
        if(numMarks <= 0 && count <= 0)
            throw logic_error("No more marks to set back to");
        int newCount;
        if(numMarks > 0)
            newCount = marks[--numMarks];
        else
            newCount = 0;
        T *objects = reinterpret_cast<T*>(objectBuffer);
        // "placement delete" nincs, ilyenkor egyszeruen meghivjuk
        // a destruktort es majd char tombkent felszabaditjuk a teruletet
        // maga az objektum gyakorlatilag megsemmisultnek tekintheto,
        // a memoria alatta majd a ~MarkRelease()-ben fog felszabadulni
        for(int i=count-1; i >= newCount; --i)
            objects[i].~T();

        count = newCount;
    }

    T* allocate();
};

template<typename T, int MAX_MARKS=256>
MarkRelease<T, MAX_MARKS>::MarkRelease(int arraySize):
    objectBuffer(new char[arraySize*sizeof(T)]),
    size(arraySize),
    count(0),
    numMarks(0)
{
}

template<typename T, int MAX_MARKS=256>
T* MarkRelease<T, MAX_MARKS>::allocate()
{
    if(count >= size)
        throw logic_error("Buffer overflow");
    // ez a placement new, ahol nem allokal uj memoriateruletet
    // egyszeruen az altalunk megadott mutatoval fog visszaterni
    // a konstruktor meghivasa utan
    return new (objectBuffer+sizeof(T)*(count++)) T();
}

class A
{
    static int counter;
    const int number;
public:
    A(): number(counter++)
    {
        cout << "\tA(" << number << ")" << endl;
    }

    ~A()
    {
        cout << "\t~A(" << number << ")" << endl;
    }
};

int A::counter = 0;

int main()
```

```
{
    MarkRelease<A> ml(24);
    ml.allocate();
    cout << "Mark" << endl;
    ml.mark();
    for(int i=0; i<8; ++i)
        ml.allocate();
    cout << "Mark" << endl;
    ml.mark();
    for(int i=0; i<3; ++i)
        ml.allocate();
    cout << "Release" << endl;
    ml.release();
    cout << "Release" << endl;
    ml.release();
    cout << "Release" << endl;
    ml.release();
    return 0;
}
```

templates/markrel.txt

```
    A(0)
Mark
    A(1)
    A(2)
    A(3)
    A(4)
    A(5)
    A(6)
    A(7)
    A(8)
Mark
    A(9)
    A(10)
    A(11)
Release
    ~A(11)
    ~A(10)
    ~A(9)
Release
    ~A(8)
    ~A(7)
    ~A(6)
    ~A(5)
    ~A(4)
    ~A(3)
    ~A(2)
    ~A(1)
Release
    ~A(0)
```

Látható, hogy a függvény implementációt is sablonként kell definiálnunk, ami elég körülményessé teszi a leírását, merő tömörségből és átláthatóságból a konstruktort és az `allocate` metódust leszámítva az összes függvény definíciót az osztályba írtam.

A szabvány sablon könyvtár

A standard template library (STL) sok hasznos generikus adatszerkezetet tartalmaz, hogy ezeknek a sziszifuszi lekódolásával már ne kelljen foglalkozni.

Legyen az a feladat, hogy egy szöveges állományban kell megszámolnunk, hogy melyik szó hányszor szerepel és sorrendben kiírni úgy, hogy az első helyen a legritkább szó szerepeljen, a végére pedig a leggyakoribb szó kerüljön. Fárasztó munka lenne a nulláról megírni, de az STL segítségével könnyedén megoldható:

templates/stl.cc

```
#include <iostream>
#include <fstream>
#include <vector>
#include <map>
#include <string>
#include <algorithm>

using namespace std;

// az algorithm sort fuggvenye nem a C-ben megszokott komparatort
// varja, hanem egy olyan tipusu erteket, amit felparameterezhet
// ket osszehasonlitando ertekkel, es az eredmenynek akkor kell
// igaznak lennie, ha az elso kisebb volt, mint a masodik

// ez az ertek lehet egy sima fuggveny is, jelen esetben viszont
// inkabb egy operator()-t tamogato osztaly, mert a kapott ertekek
// alapjan a referencia mapben hivatkozott ertekeket fogjuk felhasznalni
// a rendezeshez
```

```cpp
// az osztalyt a kiirasi sorrend megallapitasahoz fogjuk felhasznalni,
// ahol az egyes szavakat aszerint akarjuk rendezni a tombben, hogy
// hanyszor fordulnak elo
template<typename K, typename V> class LessByValue
{
    const map<K, V> &ref;
public:
    LessByValue(const map<K, V> &map): ref(map)
    {
    }

    // true-val kell visszaternie, ha a < b
    bool operator()(const K &a, const K &b)
    {
        // a typename kulcsszo itt kell, mert egyebkent a fordito
        // nem tudja, hogy a const_iterator egy tipus, vagy egy statikus
        // fuggveny vagy valtozo

        // nyugi, en sem tudtam, hogy itt most hasznalni kell,
        // csak azt, hogy ha pampog a fordito, akkor meg kell
        // probalni a typename kulcsszot beszurva is :)

        // a map find fuggvenye normal esetben egy map iteratort
        // ad vissza

        // mivel ez a map konstans, ezert nem sima, hanem
        // const_iteratort kapunk
        typename map<K, V>::const_iterator ai = ref.find(a);
        typename map<K, V>::const_iterator bi = ref.find(b);

        // ha nem talalja, akkor az iterator az end() erteket
        // adja vissza

        // elore vesszuk azokat, amik annyira ritkak, hogy a
        // mapbe se kerultek bele
        if(ai == ref.end())
            return bi != ref.end();
        if(bi == ref.end())
            return false;
        // az iterator egy pointerkent viselkedik, es egy
        // pair<string, int> tipusu ertekre mutat

        // ha a es b ugyanannyiszor fordul elo, akkor
        // rendezzuk lexikografikusan
        if(ai->second == bi->second)
            return a < b;
        return ai->second < bi->second;
    }
};

const char * const IGNORED_CHARS = ".,!?\"'()";

int main()
{
    ifstream input("donut.txt");
    // szo => hanyszor fordul elo osszerendeles
    map<string, int> words;
    // ebben fogjuk majd a szavakat az elofordulasi
    // szamuk szerint rendezni
    vector<string> order;
    while(!input.eof())
    {
        string word;
        // kiolvassuk a szot
        input >> word;
        // vesszuk az elso nem kozpontozo karaktert
        int first = word.find_first_not_of(IGNORED_CHARS);
        // vesszuk az utolso nem kozpontozo karaktert
        int last = word.find_last_not_of(IGNORED_CHARS);

        // ha talaltunk nem kozpontozo karaktert
        // a string::find... fuggvenyei string::npos erteket adnak
        // vissza, ha nem talaltak semmit
        if(first != string::npos && last != string::npos && first <= last)
        {
            word = word.substr(first, last-first+1);
            // a transform az elso ket parameterben kap ket forras iteratort
            // (az elsotol kezdve halad egeszen a masodikig)
            // a harmadik parametere az, ahova az eredmeny kerul
            // (ahogy lathato, siman megengedett, hogy ugyanaz legyen)
            // a forras minden elemere meghivja a negyedik parameterben
            // megadott fuggvenyt (vagy osztalyt, ami a megfelelo formaju
```

```
            // operator()-t tamogatja), es a visszateresi erteket irja
            // ki a megadott helyre

            // a tolower a std nevterben mar definialva van, ezert most
            // kulon jelolom, hogy a globalis nevter tolower fuggvenyere
            // gondoltam
            transform(word.begin(), word.end(), word.begin(), ::tolower);
            // ha nem szerepel a szo a mapben
            if(words.find(word) == words.end())
            {
                // akkor ez az elso elofordulas
                words[word] = 1;
                // es akkor az order tombben sem szerepelt eddig,
                // az is tuti
                order.push_back(word);
            } else
            {
                // egyebkent noveljuk a map erteket
                ++words[word];
            }
        }
    }

    // rendezzuk az order tombot aszerint, hogy a szo hanyszor fordul elo
    sort(order.begin(), order.end(), LessByValue<string, int>(words));

    // kiirjuk a vegeredmenyt
    for(vector<string>::iterator orderIt = order.begin();
        orderIt != order.end();
        ++orderIt)
    {
        map<string, int>::iterator it = words.find(*orderIt);
        if(it != words.end())
        {
            cout.width(4);
            cout << it->second << " " << it->first << endl;
        }
    }

    return 0;
}
```

A donut.txt bemenetre esetén a válasza az stl.txt-ben található.