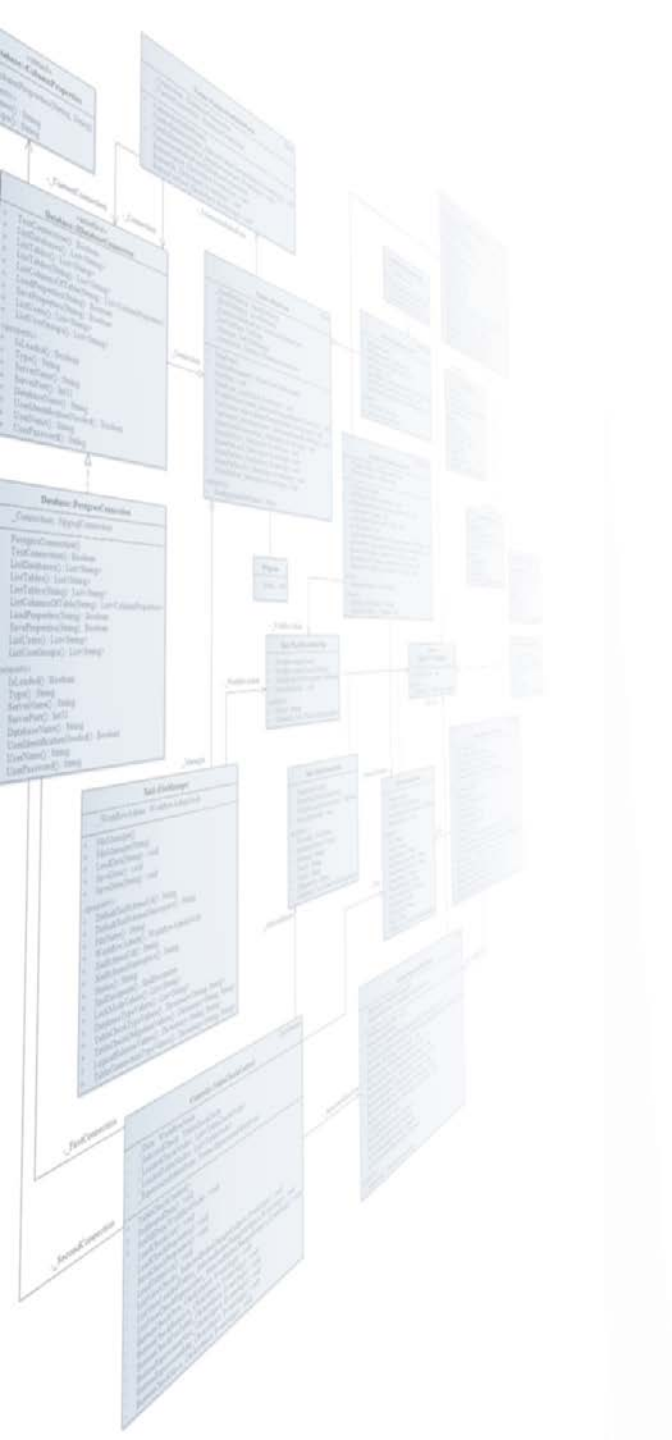


Pázmány Péter Katolikus Egyetem
Információs Technológiai és Bionikai Kar

Bevezetés a Programozásba II

4. előadás

Adattípusok hordozhatósága

© 2014.03.03. Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

Adattípusok hordozhatósága

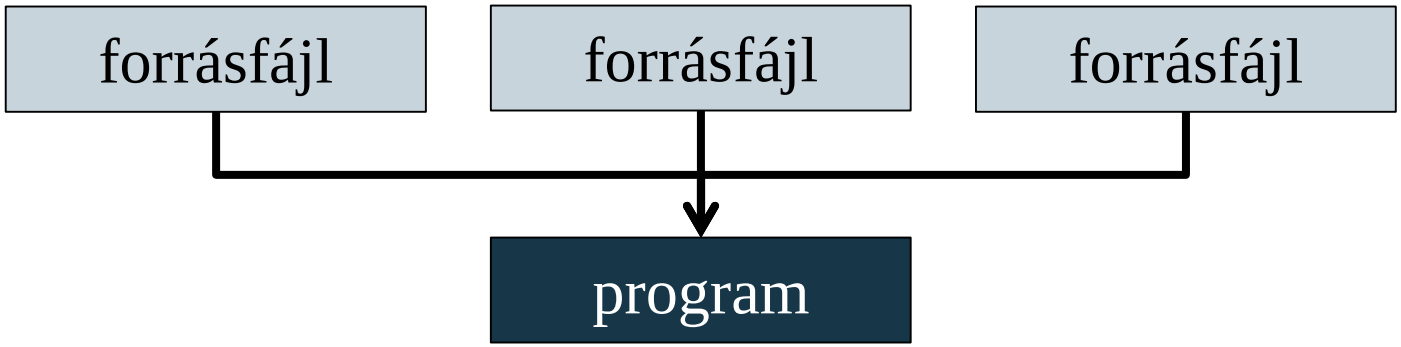
Programok tördelése

- A *strukturált programozás* során a programot adattípusok segítségével valósítjuk meg, amelyek *újrafelhasználható*an implementálunk
  - a típus könnyen átvihető, és használható más alkalmazásokban
  - ennek korlátot szab, hogy a típusokat leíró kódrészeket át kell másolnunk más programokba
  - nagyobb programok esetén a típusok egy fájlban történő elhelyezése jelentősen megnövelheti a fájlméretet, ami rontja az áttekinthetőséget
  - amennyiben a programkódot többet készítik, eleve problémás egy közös fájlon dolgozni

Adattípusok hordozhatósága

Programok tördelése

- A programunkat nem csak egy, de több forrásfájlban is elhelyezhetjük, azaz a *programkódot több fájlba is tördelhetjük*
  - minden típus külön fájlba kerülhet, így függetlenedik a többi kódtól, és könnyen hordozható lesz, és felhasználható más alkalmazásokban
  - a fejlesztők a saját fájljaikon dolgoznak, és nem zavarják más munkáját



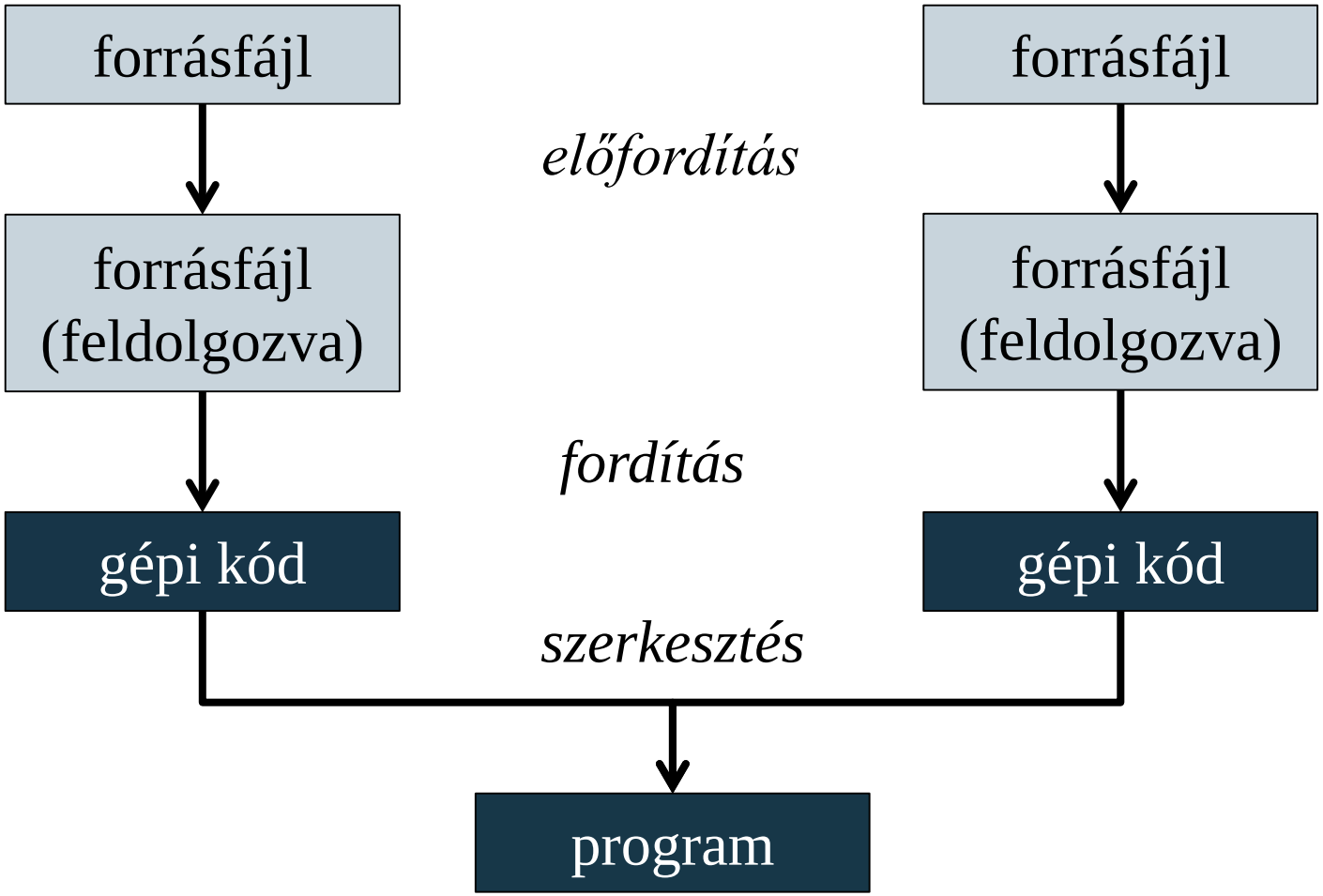
Adattípusok hordozhatósága

Programok fordítása

- A több fájlban létrehozott programkód együttesen fogja megoldani a feladatot, ehhez a környezetnek gondoskodnia kell arról, hogy a fájlok tartalma egy futtatható állományba kerüljön
- A program fordítása során a *fordítóprogram* (*compiler*) feladata az egyes fájlok kódjának átalakítása gépi kódra, míg a *szerkesztőprogram* (*linker*) feladata ezen gépi kódok összeillesztése egy futtatható állománnyá
  - ezt kiegészítheti az *előfordító*, amely előzetes átalakításokat végez a kódon
- *Fordítási egység*nek nevezzük a nyelvnek az az egysége, ami a fordítóprogram egyszeri lefuttatásával, a program többi részétől elkülönülten lefordítható

Adattípusok hordozhatósága

Programok fordítása



Adattípusok hordozhatósága

Forrásfájlok

- C++-ban a fordítási egységek két forrásfájlból állhatnak:
  - a *fejlécfájl* (*header*, **`.h`**, **`.hpp`**) tartalmazza a típusok felületét, rekordokat, metódusok és alprogramok deklarációját
    - tartalmazhatja az alprogramok megvalósítását is, de ez nem kötelező
  - a *törzsfájl* (*source*, **`.cpp`**) tartalmazza az alprogramok, metódusok definícióját, megvalósítását
- A beépített könyvtárak is ilyen fájlokból tevődnek össze, a programjaikban sokszor hivatkozunk fejlécfájlokra (pl.: **`iostream`**, **`string`**, **`vector`**, …), amelyekhez megfelelő törzsfájlok tartoztak (általában előre lefordítva)

Adattípusok hordozhatósága

Forrásfájlok

- Az előfordító befordítja a törzsfájlokba a megjelölt fejlécfájlok tartalmát, ekkor azok teljes tartalma bekerül a törzsfájlunkba
  - ehhez az **`#include`** direktívát használjuk
    - az `"..."`-ben jelölt fájlokat az aktuális könyvtárban, a `<...>`-ben jelölt fájlokat a központi könyvtárban keresi
  - amennyiben a befordított fejlécfájlokban van további hivatkozás, akkor azokat is belefordítja, és így tovább
- Az így előállított kódot a fordítóprogram fordítja le gépi kódra (**`.o`**, **`.obj`**)
- A szerkesztőprogram összeilleszti a különböző fájlokat, és elkészíti a futtatható állományt (**`.exe`**)

Adattípusok hordozhatósága

Forrásfájlok

- Típusainkat, alprogramjaikat elhelyezhetjük külön fájlokban
  - a típus a deklarációs részét a fejlécfájlba, a megvalósítási részét a törzsfájlba tesszük
  - a két fájl nevének ajánlatos megegyeznie
  - a fejlécfájlban, vagy a törzsfájlban meg kell adnunk, ha további fájlbeillesztésre van szükségünk (ha a fejlécfájlban már írtunk valamilyen hivatkozást, ezt nem kell újra beírnunk a törzsfájlba)
  - a törzsfájlban megadjuk a hozzátartozó fejlécfájl nevét
- A fejlécfájlban nem adjuk meg a használt névtereket (pl. **`std`**), csak az egyes típusoknál hivatkozunk rájuk

Adattípusok hordozhatósága

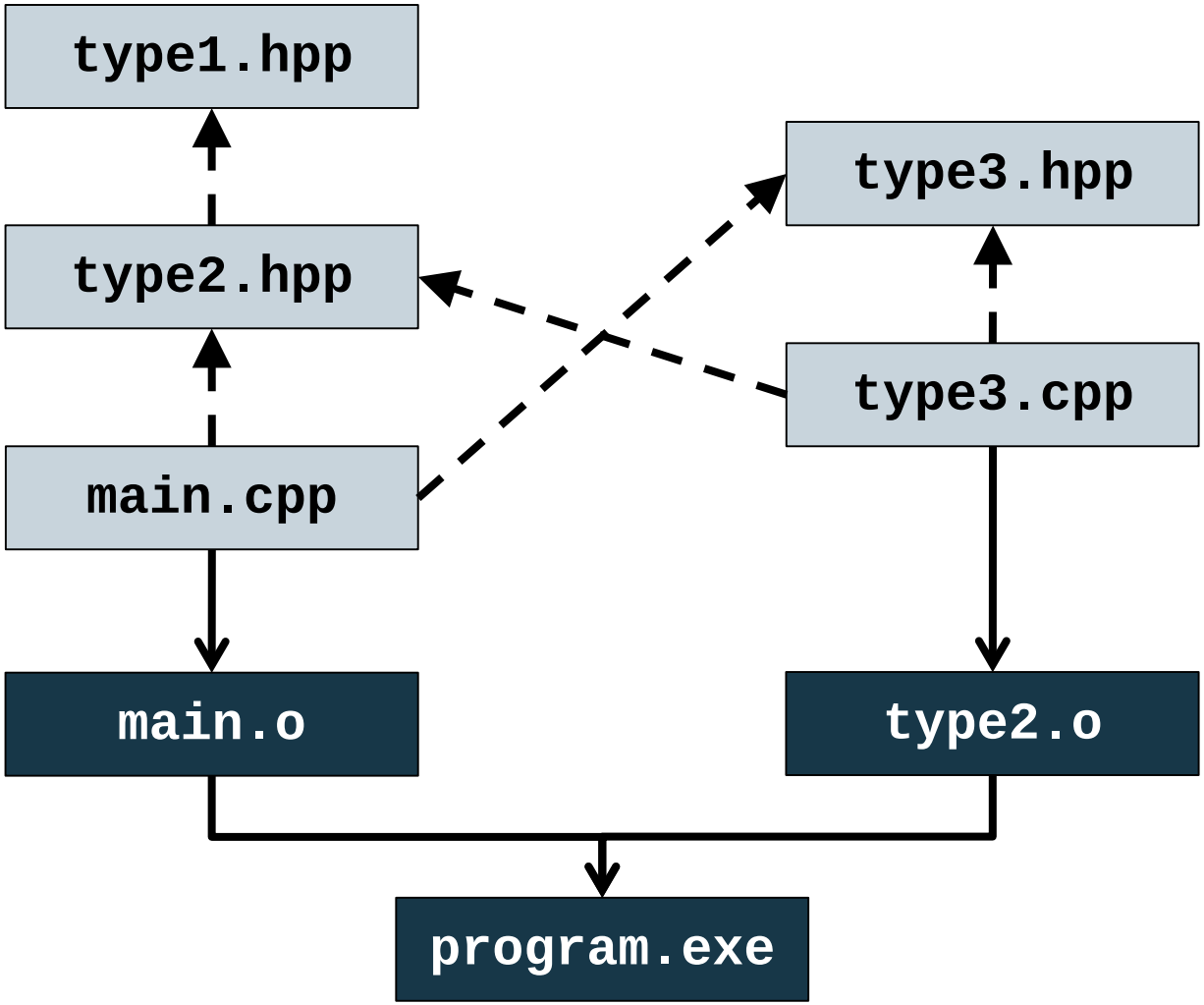
Forrásfájlok

- Mivel a teljes fejlécfájl kód bekerül a törzsfájlunkba, ezért a törzsfájl tartalmát egy az egyben behelyezhetjük a fejlécfájlba
  - ugyanúgy elhelyezhetjük a típus megvalósítását a típus felületén belül, nem kell szétválasztanunk a kódot
  - ekkor ugyanúgy fordítódik a programkód, de nem külön objektumfájlba helyeződik (az eredmény ugyanaz)
  - akkor érdemes alkalmazni, amikor a megvalósítási részt nem akarjuk szétválasztani, elrejteni a felülettől
- A főprogramunk (**`main`** függvény) fájlja (**`main.cpp`**) célszerűen nem tartalmaz típusokat, alprogramokat, csak a hivatkozásokat a többi fájlra

Adattípusok hordozhatósága

Forrásfájlok

- Pl.:



Adattípusok hordozhatósága

Fejlécfájlok

- Egy fejlécfájlt többször is beilleszthetünk a programunkba, ekkor ugyanaz a kód többször is bekerülhet, ami ahhoz vezet hogy valami többször is deklarálva lehet
- Egy előfordítási direktíva gondoskodik arról, hogy egy fájl csak egyszer kerüljön beillesztésre
  - a **`#define`** utasítással nevet adhatunk egy kódsorozatnak
  - az **`#ifndef … #endif`** elágazásban lekérdezhetjük, hogy már definiálva van-e egy adott kódsorozat
  - célszerű az egész fájlt az elágazásba helyezni, így garantáltan nem ismétlődik a tartalom

Adattípusok hordozhatósága

Fejlécfájlok

- A biztonság érdekében minden fejlécfájlunk tartalmazza:

```
#ifndef name
#define name

… // a fájl tartalma

#endif
```

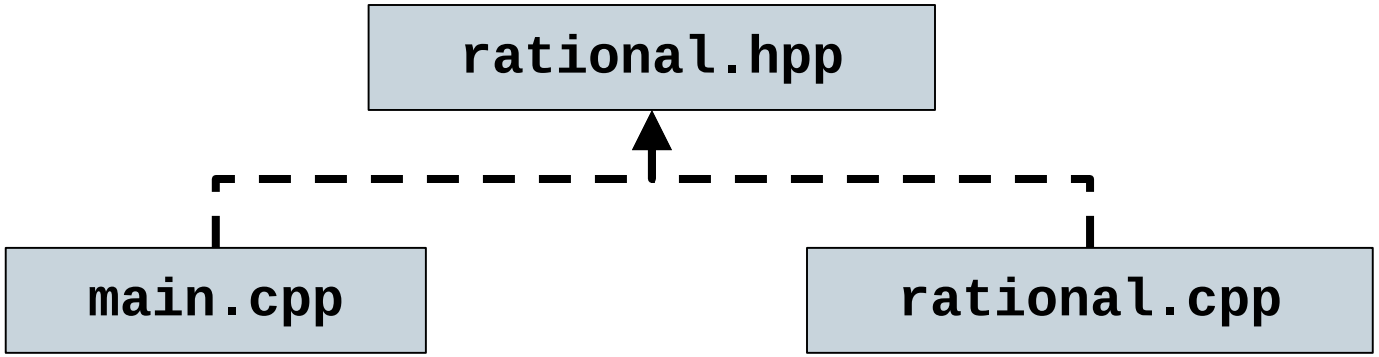
- A **name** általában megegyezik a fájl nevével, de bármit írhatunk oda (konvenció szerint csupa nagy betűvel írjuk)
- A törzsfájlok nem kerülhetnek be többször a kódba, ezért azok esetében nem kell védelemről gondoskodni

Adattípusok hordozhatósága

Példa

Feladat: Valósítsuk meg a racionális szám típusát külön fordítási egységben.

- létrehozzuk a **`rational.hpp`** fejlécfájlt, valamint a **`rational.cpp`** forrásfájlt, ezekbe helyezzüka típust, a főprogram egy külön törzsfájlba (**`main.cpp`**) kerül
- mindkét törzsfájl hivatkozni fog a fejlécfájlra



Adattípusok hordozhatósága

Példa

Megoldás (`rational.hpp`*):*

```
#ifndef RATIONAL_HPP
   // védelem a többszörös behelyezés ellen
#define RATIONAL_HPP

#include <iostream>
// nincs névtérhasználat megadva

class Rational { … } // racionális szám típusa

// további operátorok a típushoz:
Rational operator+(int e, Rational r);
…
#endif // RATIONAL_HPP
```

Adattípusok hordozhatósága

Példa

Megoldás (`rational.cpp`*):*

```
#include <iostream>
#include "rational.hpp"
   // a racionális típus használata
using namespace std;

// főprogram:
int main()
{
    Rational t[5];
    …
}
```

Adattípusok hordozhatósága

Példa

Megoldás (`rational.cpp`*):*

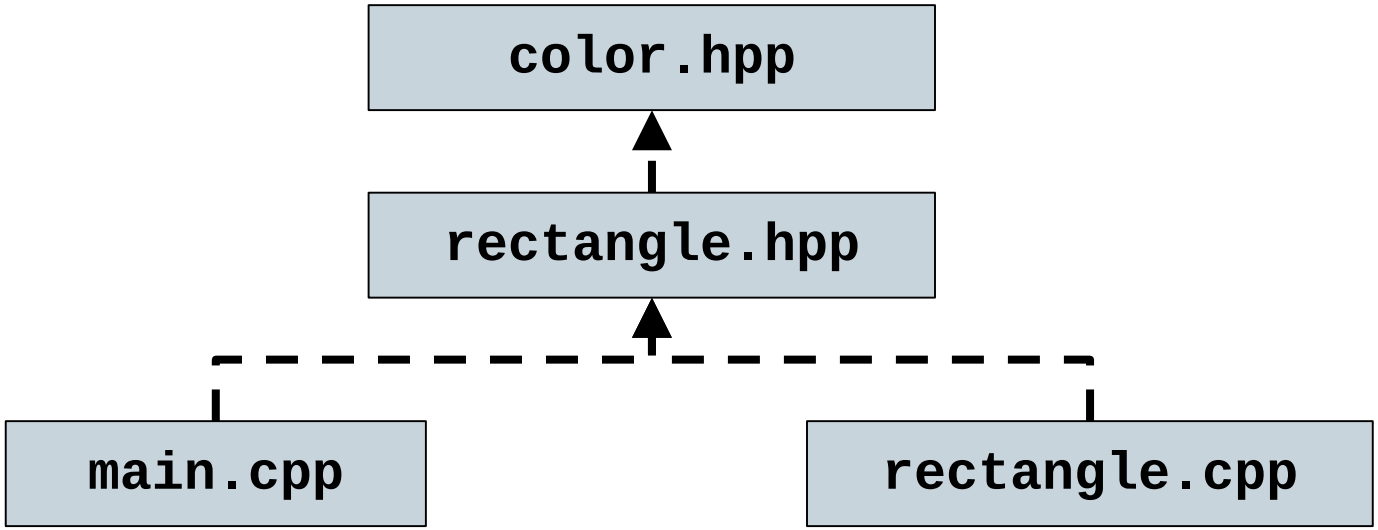
```
#include "rational.hpp"
   // szükséges a fejlécfájl használata
using namespace std;
   // itt adjuk meg a névtérhasználatot

void Rational::simplify(unsigned int& a,
                        unsigned int& b){ … }
…
int operator*=(int& e, Rational r) { … }
```

Adattípusok hordozhatósága

Példa

Feladat: Valósítsuk meg a téglalap és szín típusokat külön fordítási egységben.

- a színt csak egy fejlécfájlban (**`color.hpp`**) helyezzük el, a téglalapnak pedig készítünk külön fejlécfájlt (**`rectangle.hpp`**), és törzsfájlt (**`rectangle.cpp`**)



Adattípusok hordozhatósága

Példa

*Megoldás (**color.hpp**):*

```
class Color {
   …
   friend genv::canvas& operator<<(
      genv::canvas& c, Color col);
   …
};

genv::canvas& operator<<(genv::canvas& c, …){
   c << genv::color(col._r, col._g, col._b);
      // több helyen is jelölnünk kell a
      // névtérhasználatot
   return c;
}
```

Adattípusok hordozhatósága

Programkönyvtárak

- Az azonos tevékenységi körben készített újrafelhasználható típusokat, esetleges további alprogramokat összegyűjthetünk egy *gyűjtemény*be, amely így egységesen tartalmazza egy adott feladatcsoport kapcsán használható tartalmat
  - pl. grafikus felület létrehozása és kezelése, adatbázis-kezelés, alapvető adatszerkezetek és algoritmusok
  - a gyűjtemény egyben kezelhető, publikálható, később bővíthető
  - az így létrehozott gyűjteményeket nevezzük *programkönyvtár*aknak (*library*)
  - a programok jelentős része támaszkodik programkönyvtárakra a működés során

Adattípusok hordozhatósága

Programkönyvtárak

- Az alapvető programkönyvtárakat nevezzük *nyelvi könyvtárak*nak
  - közvetlenül a nyelvvel és a fejlesztőkörnyezettel együtt publikálják (pl. C++ standard library)
  - a legtöbb program számára szükséges funkciókat biztosítja, pl. konzol képernyő kezelése, fájlok, alapvető adatszerkezetek
- Megvalósításában egy programkönyvtár lehet
  - *nyílt forrású*: közvetlenül a forrásfájlokat tartalmazza
  - *zárt forrású*: nem tartalmazza a (teljes) forrást, csak a gépi kódú lefordított változatát

Adattípusok hordozhatósága

Programkönyvtárak típusai

- A futtatás szempontjából a programkönyvtár lehet:
  - *statikus*: amelynek tartalma teljes egészében bekerül az eredmény alkalmazás kódjába fordítási időben
    - előnye, hogy csak a fordítás után a programkönyvtár már nem játszik szerepet
    - kezdetben minden programkönyvtár statikus volt
    - a forráskód közvetlen használata mindig statikus programkönyvtárat ad
    - zárt forráskód esetén a programkönyvtár lefordított változata mellett (`.a`, `.la`) meg kell adnunk a fejlécfájlokat (`.h`, `.hpp`) is, hogy a típusokat a kódban használni tudjuk

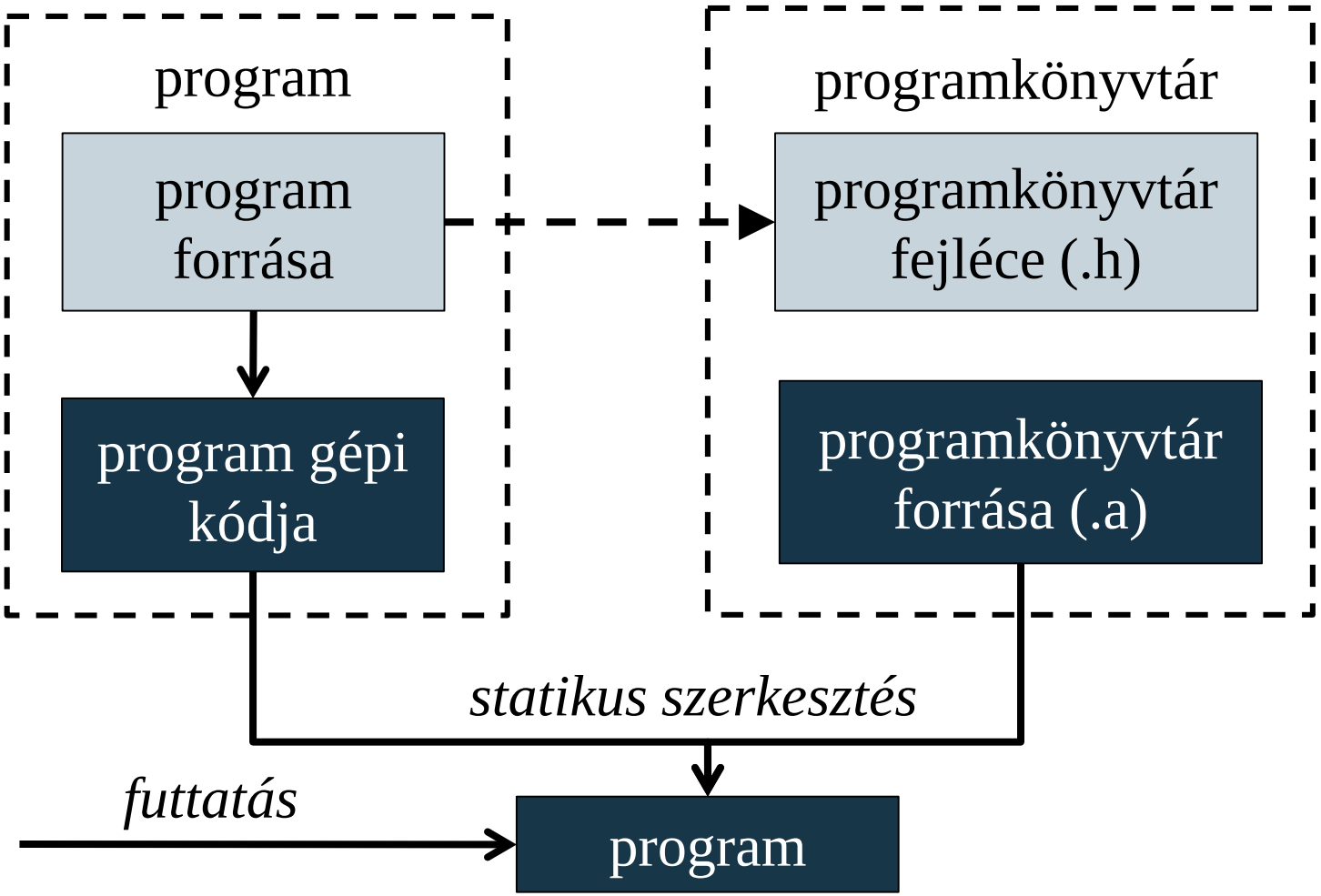
Adattípusok hordozhatósága

Programkönyvtárak típusai

- A futtatás szempontjából a programkönyvtár lehet:
  - *dinamikus*: amelynek tartalma egy külön állományban helyezkedik el, és futás közben töltődik be a memóriába
    - pl. Windows esetén a *Dinamic-linked Library* (*DLL*)
    - előnye, hogy nem kell a forráskódot egy állományba fordítani
    - ha nem sikerül betölteni a könyvtárat, futási idejű hibát kapunk
    - speciálisan a könyvtár lehet *megosztott* (*shared library*), vagyis egyszerre több alkalmazás is használhatja
      - az operációs rendszer számos megosztott programkönyvtárat tartalmaz

Adattípusok hordozhatósága

Programkönyvtárak használata



Adattípusok hordozhatósága

Programkönyvtárak használata