Eötvös Loránd Tudományegyetem
Informatikai Kar

# Alkalmazott modul: Programozás

## 6. előadás

## Strukturált programozás: újrafelhasználható adattípusok

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

# Adattípusok megvalósítása

## Újrafelhasználhatóság

- Az adattípusok megvalósításának célja, hogy az összetett szerkezetű adatok is jól, könnyen kezelhetővé váljanak
  - amennyiben csak egy programon használnánk az új adattípust, akkor nem feltétlenül térülne meg a befektetett energia, ezért arra kell törekednünk, hogy az így létrehozott típusok *újrafelhasználható*ak legyenek
- Az újfelhasználhatóságot a C++ számos eszközzel támogatja:
  - egységbe zárás, láthatóság kezelés
  - fordítási egységek használata
  - konstansok kezelése (const correctness)
  - sablonok használata

# Egységbe zárás

## Az egységbe zárás fogalma

- Lehetőségünk van a rekord belsejében is megadni alprogramokat a mezők mellett, amelyek közvetlenül a rekord típusú változóhoz fognak kötődni
  - amikor változót deklarálunk a típusból, az alprogramok automatikusan értelmezésre kerülnek rajta
  - az alprogramok közvetlenül hozzáférnek a rekord mezőihez, nem kell paraméterben átadnunk a rekordot
- A típus műveleteinek beágyazását a típusba nevezzük *egységbe zárás*nak (*enkapszuláció*nak)
  - a beágyazott alprogramokat *metódus*oknak nevezzük
  - ezek a szűkebb értelemben vett típusműveletek

# Egységbe zárás

## Metódusok használata

- A metódusokat csak változón keresztül érhetjük el, ezért mindig kell legyen egy példány a típusból, ekkor **`<változónév>.<metódusnév>(<paraméterek>)`** formában meghívhatjuk az alprogramot
- A metódusok deklarációja megegyezik a megszokott alprogram deklarációval, csak a rekord belsejében végezzük:

```
struct <rekordnév> {
    <típus> <név>(<paraméterek>){
        <törzs>
    }
};
```

- Tulajdonképpen a konstruktor is egy metódus

# Egységbe zárás

## Példa

*Feladat:* Valósítsuk meg az egyetemi hallgató típusát. A hallgatónak van neve, illetve azonosítója, és rendelkezik valamennyi pénzmennyiséggel, amelyet növelhet ösztöndíjjal, illetve csökkenthet költekezéssel.

- elkészítjük a hallgató típusát (**`Student`**), amely rendelkezik három mezővel (**`name`**, **`id`**, **`money`**)
- létrehozunk egy konstruktort, amely paraméterben kapja a hallgató nevét és azonosítóját, a kezdeti pénzmennyiség 0
- létrehozunk két metódust az ösztöndíj utalásra (**`grantScholarship`**), illetve a költésre (**`spendMoney`**), természetesen költeni csak akkor tud, ha van pénze (egyébként csak annyit költhet, amennyi pénze van)

# Egységbe zárás

## Példa

*Tervezés:*

- $Student = \begin{pmatrix} string(name) \times string(id) \times int\ (money), \\ \{spendMoney, grantScolarship\} \end{pmatrix}$, ahol

  $spendMoney: Student \times int \to Student,$

  $spendMoney(s, amount) =$

  $(s.name, s.id, s.money\ - amound)$

  $grantScolarship: Student \times int \to Student,$

  $grantScolarship(s, amount) =$

  $(s.name, s.id, s.money + amound),$

# Egységbe zárás

## Példa

*Megoldás:*

```
struct Student { // hallgató típusa
    string name; // név
    string id; // azonosító
    int money; // pénz

    Student(string i, string n){
        // konstruktor, amely megkapja az azonosítót
        // és a nevet
        id = i;
        name = n;
        money = 0; // a pénzt kinullázzuk
    }
```

# Egységbe zárás

## Példa

*Megoldás:*

```
    void grantScholarship(int amount){
        // ösztöndíj utalás
        if (amount > 0) money += amount;
            // ellenőrizzük, hogy az érték
            // megfelelő-e
    }
    void spendMoney(int amount){ // pénzköltés
        if (amount > 0)
            money =
                money > amount ? money - amount : 0;
            // 0 alá nem csökkenhet az összeg
    }
};
```

# Egységbe zárás

## Példa

*Megoldás:*

```
int main()
{
    Student h("ABCDEF", "Huba Hugó");
    h.grantScholarship(15000); // metódus hívás
    h.spendMoney(3600);
    h.spendMoney(2000);
    cout << h.name << " pénze: "
         << h.money << endl;
    return 0;
}
```

# Egységbe zárás

## Deklaráció és definíció szétválasztása

- Metódusok esetén is szeparálhatjuk a deklarációt a definíciótól, ekkor a rekordba csak a metódus szintaxisát írjuk, a törzsét később
  - a definíciónál meg kell jelölnünk, melyik típus metódusát adjuk meg a `::` (*scope*) operátor segítségével:

```
struct <típusnév>{
    …
    <típus> <metódusnév>(<paraméterek>);
};
…
<típus> <típusnév>::<metódusnév>(<param.>) {
    <metódustörzs>
}
```

# Egységbe zárás

## Példa

*Feladat:* Módosítsuk a hallgató típusát úgy, hogy leválasztjuk a metódusok törzsét a deklarációról.

*Megoldás:*

```
struct Student { // hallgató típusa
    string name; // név
    string id; // azonosító
    int money; // pénz

    Student(string i, string n); // konstruktor
    void grantScholarship(int amount);
       // ösztöndíj utalás
    void spendMoney(int amount); // pénzköltés
};
```

# Egységbe zárás

## Példa

*Megoldás:*

```
…
Student::Student(string i, string n) {
   // jelölnünk kell a típust
   id = i;
   name = n;
   money = 0; // a pénzt kinullázzuk
}
void Student::grantScholarship(int amount) {
   if (amount > 0)
      // ellenőrizzük, hogy az érték megfelelő-e
      money += amount;
}
…
```

# Egységbe zárás

## Operátorok metódusként

- Lehetőségünk van operátorainkat is metódusként megvalósítani, ekkor egy speciális szabály, hogy az operátor első paramétere az a típuspéldány lesz, amelybe beágyaztuk
  - így minden operátornak eggyel csökken a paraméterszáma, így pl. a **+** operátornak nulla és egy paraméteres változata lesz metódusként
  - bizonyos operátorok (értékadás, indexelés) csak metódusként, bizonyos operátorok (adatátvitel, egyes speciális értékadások) csak a típuson kívül adhatunk meg
  - a legtöbb operátort a típuson kívül szokás megadni

# Egységbe zárás

## Példa

*Feladat:* Módosítsuk a racionális szám típusát úgy, hogy az egyszerű operátorokat a típuson belül valósítjuk meg.

*Megoldás:*

```
struct Rational { // racionális szám típusa
    int numerator;
    int denominator;

    // konstruktorok:
    Rational();
    Rational(int);
    Rational(int, int);
```

# Egységbe zárás

## Példa

*Megoldás:*

```
    // operátor metódusok:
    Rational operator+(Rational);
    Rational operator+(int);
    Rational operator*(Rational);
    Rational operator*(int);
};

// további operátorok a típushoz:
Rational operator+(int e, Rational r);
Rational operator*(int e, Rational r);
istream& operator>>(istream& s, Rational &r);
ostream& operator<<(ostream &s, Rational r);
…
```

# Láthatóság kezelés

## Adatok elrejtése

- Típusoknál nem mindig célszerű, hogy külsőleg minden tag elérhető és változtatható legyen
  - egymástól függő, vagy korlátozott értékek rossz beállítása *inkonzisztens állapotba* hozhatja a példányt
  - pl. racionális szám esetén a nevező beállítása 0-ra
- A tagokat elrejthetjük a külvilág (többi programegység) elől a *láthatóság szabályozásával*
  - az elrejtendő tagokat **`private:`** kulcsszó mögé tesszük, ezek csak a típuson belül lesznek láthatóak
  - a mindenki számára elérhető tagokat a **`public:`** kulcsszó mögé tesszük

# Láthatóság kezelés

## Láthatóság szabályozó kulcsszavak

- Rejtett tagok külső elérésekor fordítási hibát kapunk
- A láthatósági kulcsszavak használata nem kötelező, ha nem tesszük meg, **`struct`** esetében alapértelmezetten minden látható lesz
- A kulcsszavakat elég csak egyszer kiírnunk, az azután következő elemekre érvényesek lesznek:

```
struct <típusnév>{
    public:
        // publikus tagok felsorolása
    private:
        // rejtett tagok felsorolása
};
```

# Láthatóság kezelés

## Lekérdező/beállító műveletek

- Az elrejtett tagokat elnevezhetjük más módon (pl. _ jellel kezdve), így könnyen azonosíthatjuk őket
- Amennyiben külső hozzáférést szeretnénk biztosítani a rejtett mezőkhöz biztonságos módon, készíthetünk publikus *beállító* (*setter*), illetve *lekérdező* (*getter*) műveleteket
- Pl.:

```
struct MyType {
private:
    int _value; // rejtett érték
public:
    void setValue(int v) { … } // beállítás
    int getValue() { return _value; } // lekérdezés
};
```

# Láthatóság kezelés

## Példa

*Feladat:* Módosítsuk a racionális szám típusát úgy, hogy a mezőket elrejtjük a külvilágtól, és lekérdező, valamint beállító műveleteket hozunk létre.

*Megoldás:*

```
struct Rational {
private: // a mezők rejtettek
    int _numerator;
    int _denominator;

public: // a műveletek publikusak
    Rational();
    Rational(int);
    Rational(int, int);
```

# Láthatóság kezelés

## Példa

*Megoldás:*

```
    // lekérdező és beállító műveletek:
    void setNumerator(int value);
    void setDenominator(int value);
    int numerator();
    int denominator();
    …
};
// további operátorok a típushoz:
Rational operator+(int e, Rational r);
Rational operator*(int e, Rational r);
istream& operator>>(istream& s, Rational &r);
ostream& operator<<(ostream &s, Rational r);
…
```

# Láthatóság kezelés

## Példa

*Megoldás:*

```
Rational::Rational() {
    _numerator = 0;
    _denominator = 1;
}
…
void Rational::setDenominator(int value) {
    _denominator = value == 0 ? 1 : value;
  // speciális beállítás
}
…
int Rational::denominator() {
    return _denominator;
}
```

# Láthatóság kezelés

## Példa

*Megoldás:*

```
// külső értékmódosítási operátorok:
Rational operator+=(Rational& r1, Rational r2) {
   r1.setNumerator(r1.numerator() *
       r2.denominator() + r2.numerator() *
       r1.denominator());
   // a műveletet a lekérdező/beállító
   // műveleteken keresztül végezzük el
   r1.setDenominator(r1.denominator() *
       r2.denominator());
   return r1;
}
```

# Láthatóság kezelés

## Szabályok

- Általában a mezőket és a segédműveleteket elrejtjük, a metódusokat láthatóvá tesszük
- Ha a konstruktort elrejtjük, akkor nem lehet példányosítani a típust, azaz nem tudunk belőle változót létrehozni
  - néha hasznos, ha a típust direkt nem akarjuk példányosítani
- A típusok tekintetében a C++ két kategóriát különböztet meg
  - **`struct`**: alapértelmezetten minden látható
  - **`class`**: alapértelmezetten minden rejtett
- A konvenció szerint a **`class`** kulcsszót összetett, a **`struct`** kulcsszót egyszerű típusokra használjuk

# Láthatóság kezelés

## Típus kulcsszavak

- Pl.:

```
struct MyType {          // struct -> public:
     MyType();           // látható
     ~MyType();          // látható
     string func1();     // látható
public:
     int func2();        // látható
     strint value1;      // látható
private:
     int func3();        // rejtett
     int _value2;        // rejtett
     int _value3;        // rejtett
};
```

# Láthatóság kezelés

## Típus kulcsszavak

- Pl.:

```
class MyType {          // struct -> private:
     MyType();          // rejtett
     ~MyType();         // rejtett
     string func1();    // rejtett
public:
     int func2();       // látható
     strint value1;     // látható
private:
     int func3();       // rejtett
     int _value2;       // rejtett
     int _value3;       // rejtett
};
```

# Láthatóság kezelés

## Példa

*Feladat:* Módosítsuk a racionális szám típusát úgy, hogy egyszerűsítést is végrehajtunk az értékbeállítások után.

- ez lehet rejtett művelet, hiszen csak belülről használjuk
- egyúttal adjunk másik kulcsszavat is a típusnak

*Megoldás:*

```
class Rational {
private: // a mezők és a segédműveletek rejtettek
   …
   void simplify(int& a, int& b); // egyszerűsítés
   …
```

# Láthatóság kezelés

## Példa

*Megoldás:*

```
void Rational::simplify(int& a, int& b) {
    int c = a, d = b;
    // euklideszi algoritmus futtatása
    while (c != d)
    {
        if (c > d)
            c -= d;
        else
            d -= c;
    }
    a /= c;
    b /= d;
}
```

# Láthatóság kezelés

## Példa

*Megoldás:*

```
…
Rational::Rational(int num, int denom) {
     _numerator = num;
     _denominator = denom != 0 ? denom : 1;
     simplify(_numerator, _denominator);
}

void Rational::setNumerator(int value)
{
     _numerator = value;
     simplify(_numerator, _denominator);
}
…
```

# Láthatóság kezelés

## Barát műveletek

- Néha kényelmetlen, hogy a típushoz tartozó külső metódusok (pl. operátorok) nem láthatják a rejtett mezőket, holott igazából nem a külvilághoz tartoznak
- A típusban lehetőségünk van megadni *barát* (*friend*) műveletek halmazát, olyan külső műveleteket, amelyek láthatják a rejtett értékeket is
  - a műveletek szintaxisát a típusban kell leírni a **`friend`** kulcsszó mellett (ez nem számít deklarációnak, csak egy jelzésnek)
  - az azonos szintaxissal létrehozott külső műveletek láthatják a rejtett értékeket

# Láthatóság kezelés

## Barát műveletek

- Pl.:

```
struct MyType {
private:
    int _value; // rejtett érték

    friend void AddValue(MyType, int);
        // barát művelet jelzése
};
…
void AddValue(MyType m, int v){
    // barát művelet definíciója
    m._value += v;
    // módosíthatjuk a rejtett értéket
}
```

# Láthatóság kezelés

## Példa

*Feladat:* Módosítsuk a racionális szám típusát úgy, hogy a külső operátorok barát műveletek legyenek.

*Megoldás:*

```
class Rational {
    …

    // barát műveletek:
    friend Rational operator+(int e, Rational r);
    friend Rational operator*(int e, Rational r);
    friend istream& operator>>(istream& s,
                               Rational &r);
    …
```

# Láthatóság kezelés

## Példa

*Megoldás:*

```
// külső értékmódosítási operátorok:
Rational operator+=(Rational& r1, Rational r2) {
   r1._numerator = r1._numerator *
       r2._denominator + r2._numerator *
       r1._denominator;
      // a műveletet a rejtett mezőkön végezzük el
   r1._denominator = r1._denominator *
       r2._denominator;
   r1.simplify(r1._numerator, r1._denominator);
      // rejtett egyszerűsítés meghívása
   return r1;
}
…
```

# Fordítási egységek

## Fájlok C++-ban

- Nagy mennyiségű programkódnál a kódfájl áttekintetlen lesz, ekkor célszerű a kódot több fájlban elhelyezni
- 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 egységét, ami a fordítóprogram egyszeri lefuttatásával, a program többi részétől elkülönülten lefordítható

# Fordítási egységek

## Fájlok C++-ban

- C++-ban a fordítási egységek két fájlból állnak:
  - a *fejlécfájl* (*header*, **`.h`**, **`.hpp`**) tartalmazza a típusok felületét, rekordok és alprogramok deklarációját
    - tartalmazhatja az alprogramok megvalósítását is, de ez nem kötelező
  - a *törzsfájl* (*source*, **`.cpp`**) tartalmazza az alprogramok 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)

# Fordítási egységek

## Programok fordítása

- Az előfordító bemásolja a törzsfájlokba a hivatkozott fejlécfájlok tartalmát
  - ehhez az **`#include`** direktívát használjuk, az `"..."`-ben hivatkozott fájlokat az aktuális könyvtárban, a `<...>`-ben hivatkozott fájlokat a központi könyvtárban keresi
  - a fejlécfájlokban lévő további hivatkozásokat is feldolgozza (ezért tilos körkörös hivatkozást adni)
- Az így előállított kódot a fordítóprogram fordítja le gépi kódra (**`.o`**, **`.a`**, vagy **`.lib`** kiterjesztésben)
- A szerkesztőprogram összeilleszti a különböző fájlokat, és elkészíti a futtatható állományt (**`.exe`**)

# Fordítási egységek

## Programok tördelése több fájlba

- Típusainkat, alprogramjainkat elhelyezhetjük külön fájlokban, a deklarációs részeket a fejlécfájlba, a megvalósítást a törzsfájlba tesszük
  - a két fájl nevének ajánlatos megegyeznie
  - a törzsfájlban megadjuk a hozzátartozó fejlécfájl hivatkozását
  - további fájlhivatkozásokat a megszokott módon tesszük, de a fejlécfájlban megadott hivatkozásokat nem kell megismételnünk a törzsfájlban
  - a fejlécfájlban biztonsági okokból nem adjuk meg a használt névtereket (pl. `std`), csak az egyes típusoknál hivatkozunk rájuk

# Fordítási egységek

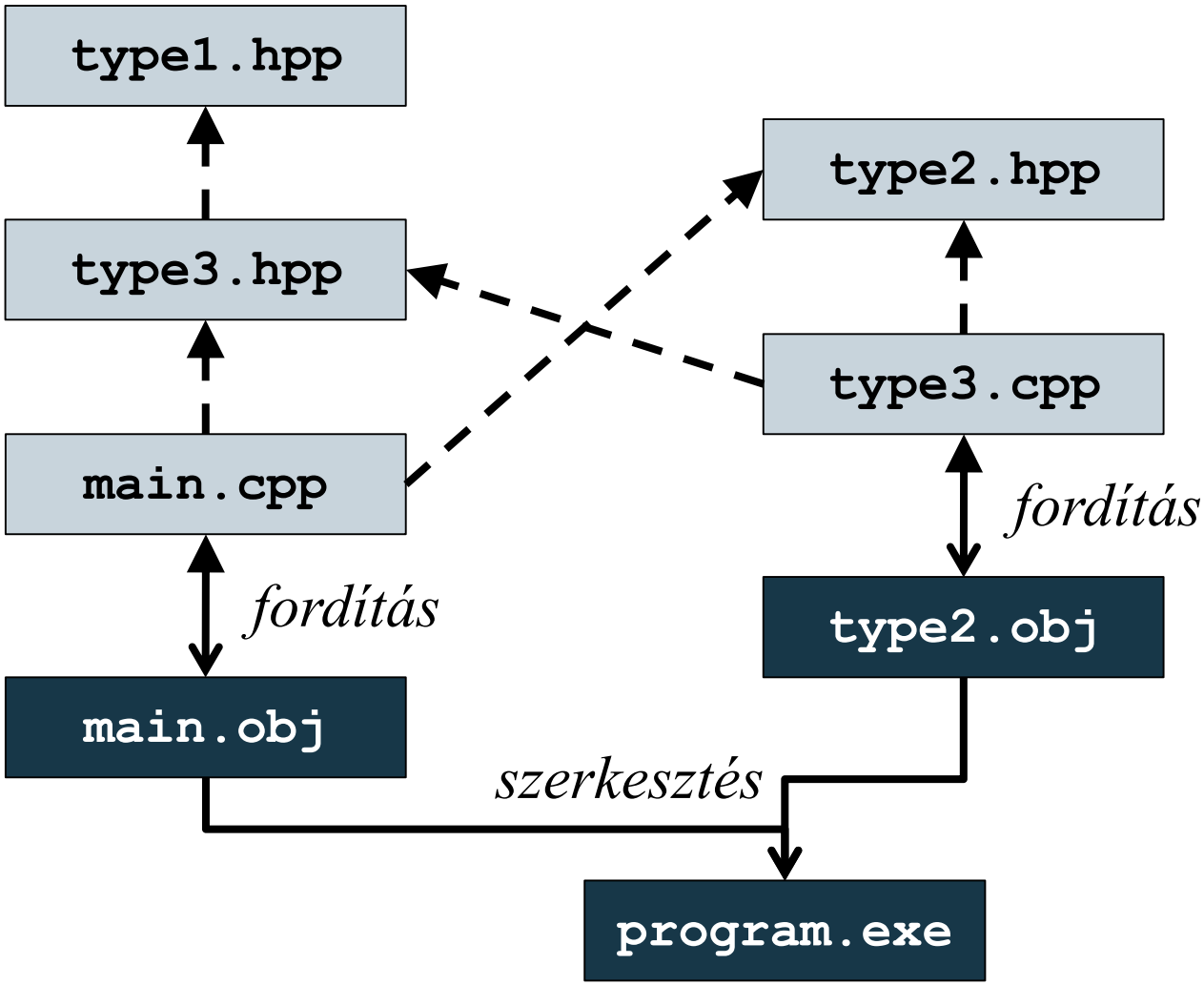
## Programok tördelése több fájlba

- Mivel a teljes fejlécfájl kódja 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 a programkód nem külön objektumfájlba fordul, de ez a használatot nem befolyásolja
  - akkor érdemes alkalmazni, amikor a megvalósítási részt nem akarjuk szétválasztani, elrejteni a felülettől
- A főprogramunk (`main` függvény) fájlja (`main.cpp`) általában nem tartalmaz típusokat, alprogramokat, csak a hivatkozásokat a többi fájlra

# Fordítási egységek

## Programok tördelése több fájlba

- Pl.:



# Fordítási egységek

## Fejlécfájlok felépítése

- A fejlécfájlok használatánál figyelni kell arra, hogy a tartalma csak egyszer kerüljön felhasználásra, különben többszörös definíciók születnek (pl. egy típus kétszer is bekerül a programkódba)
- A fejlécfájl tartalmát ezért egy elágazásba kell helyeznünk, amely a fordítás számára jelöli, amennyiben már használatba került, ezt előfordítási direktívák segítségével érhetjük el
  - a **`#define <név>`** utasítással létrehozhatunk egy makrót, amely egy új elnevezés lesz a fordító számára
  - az **`#ifndef <név>`** … **`#endif`** elágazással készíthetünk egy kódsorozatot, amelyet csak akkor vesz figyelembe a fordító, amennyiben még nem definiált a makró

# Fordítási egységek

## Fejlécfájlok felépítése

- Ennek megfelelően a fejlécfájlt a következő keretbe építjük:

```
#ifndef <név>
#define <név>

… // a fejlécfájl tartalma

#endif
```

- A név általában megegyezik a fájl nevével, de egyébként bármi lehet (konvenció szerint csupa nagy betűvel írjuk)
- A törzsfájlok tartalma is bekerülhet többszörösen a programkódba (szerkesztéskor), de megfelelő használattal ez elkerülhető

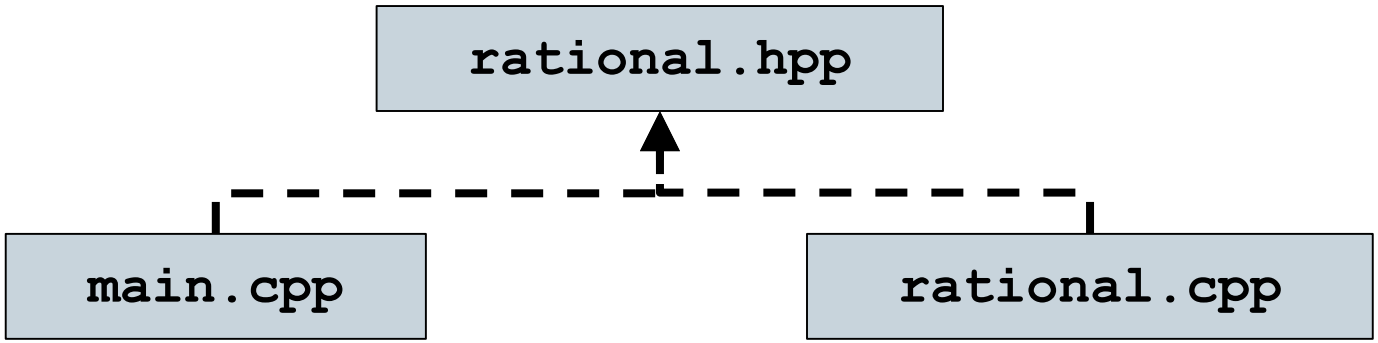
# Fordítási egységek

## Példa

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

- létrehozzuk a **`rational.hpp`** fejlécfájlt, valamint a **`rational.cpp`** forrásfájlt, ezekbe helyezzük a típust, a főprogram egy külön törzsfájlba (**`main.cpp`**) kerül

*Tervezés:*



# Fordítási egységek

## Példa

*Megoldás (*`rational.hpp`*):*

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

#include <iostream>
// nincs névtér használat megadva

class Rational { … } // racionális szám típusa

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

# Fordítási egységek

## Példa

*Megoldás (*`rational.cpp`*):*

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

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

# Fordítási egységek

## Példa

*Megoldás (*`main.cpp`*):*

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

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

# Konstansok kezelése

## Konstansok

- A programjainkban sokszor használunk *konstans*okat, amelyek lehetnek név nélküli, vagy elnevezett konstansok
  - elnevezett konstansokat a **`const`** kulcsszóval tudunk létrehozni, ekkor a névhez rögzített értéket rendelhetünk, pl.: **`const double pi = 3.14159265358979323846;`**
  - lehetőségünk van konstansokat változónak, illetve változóértéket konstansnak értékül adni, pl.:

```
int u; u = 5; // u változó, kezdetben 5 értékkel
const int v = u;
   // v egy elnevezett konstans, amely megkapja
   // u értékét, azaz 5-t
int w = v; // w már változó
w++; // így módosíthatjuk az értékét
```

# Konstansok kezelése

## Konstansok

- Konstansokat alprogramoknál a paraméterátadásban, vagy a visszatérési értékben is használhatunk
  - konstans paraméter esetén nem módosíthatjuk a paraméterben kapott értéket
  - konstans visszatérési érték esetén az érték nem módosítható
  - pl.:

```
const int SomeFunction(const float param){ … }
```

- Természetesen a saját típusainkból is létrehozhatunk konstansokat, pl.: `const MyType m;`
  - a konstruktor művelet beállítja a kezdőértékeket, de ezt követően nem módosíthatunk semmin

# Konstansok kezelése

## Konstans referenciák

- Nem csupán egyszerű konstansokat, de *konstans referenciák*at (álneveket) is létrehozhatunk
  - pl.:

```
MyType m; // változó
const MyType& n = m; // konstans referencia
```

  - ekkor nem jön létre új érték, de nem módosíthatunk az értékeken a későbbiek során a referencián keresztül (a változón keresztül igen)
  - ha paraméterátadásban használjuk, akkor elérjük, hogy csak bemenő irányú legyen a paraméter, ugyanakkor ne másolódjon a memóriában, ami *hatékony és biztonságos programműködés*t tesz lehetővé

# Konstansok kezelése

## Konstans metódusok

- Amennyiben egy típuspéldányt konstansként (vagy konstans referenciaként) hozunk létre, csak olyan metódusai hívhatóak, amelyek nem módosítják az objektum mezőit, ezeket nevezzük *konstans metódusok*nak
  - konstans metódusban nem módosíthatóak a mezők (ez fordítási időben ellenőrzésre kerül)
  - a kódban a **`const`** kulcsszóval jelöljük a metódust:

```
class <típusnév> {
    <típus> <metódusnév>(<paraméterek>) const {
        <nem módosító műveletek>
    }
    …
};
```

# Konstansok kezelése

## Konstans metódusok

```
class MyType {
  private:
     int _value;
  public:
     int getValue() const { // konstans művelet
        return _value;
     } // nem módosít semmilyen értéket
      void setValue(int v) { _value = v; }
};

…
const MyType mt; // konstans objektum
cout << mt.getValue(); // ez megengedett
mt.setValue(10); // erre fordítási hibát kapunk
```

# Konstansok kezelése

## Példa

*Feladat:* Módosítsuk a racionális szám típusát, hogy konstans értékekkel is jól, hatékonyan dolgozzon.

- a lekérdező műveleteket, operátorokat konstans műveletnek jelöljük
- paraméterátadáskor az érték szerinti másolás helyett konstans referenciát használunk

*Megoldás (`rational.hpp`):*

```
class Rational {
   …
   void simplify(int& a, int& b) const;
      // egyszerűsítés, lehet konstans művelet
   …
```

# Konstansok kezelése

## Példa

*Megoldás (*`rational.hpp`*):*

```
// lekérdező és beállító műveletek:
void setNumerator(int value);
void setDenominator(int value);
int numerator() const;
   // a lekérdezések konstans műveletek
int denominator() const;

// operátor metódusok:
Rational operator+(const Rational&) const;
   // az operátorok is konstans műveletek
Rational operator+(int) const;
…
```

# Konstansok kezelése

## Példa

*Megoldás (*`rational.hpp`*):*

```
// további operátorok a típushoz:
Rational operator+(int e, const Rational& r);
Rational operator*(int e, const Rational& r);
std::istream& operator>>(std::istream& s,
                         Rational &r);
std::ostream& operator<<(std::ostream &s,
                         const Rational& r);
…
```