



Eötvös Loránd Tudományegyetem
Informatikai Kar

Alkalmazott modul: Programozás

5. előadás

Strukturált programozás: Adattípusok megvalósítása elemi eszközökkel

Giachetta Roberto

`groberto@inf.elte.hu`

<http://people.inf.elte.hu/groberto>

Adattípusok megvalósítása elemi eszközökkel

Adattípusok

- *Adattípus* = *értékhalmoz* + *művelethalmoz*, ahol
 - *értékhalmoz*: a felvehető értékek halmaza (mindig véges, mivel a lefoglalható memóriaterület mérete is véges)
 - *művelethalmoz*: az értékhalmozon (esetlegesen más értékhalmozokon) értelmezett műveletek
- Pl.:
 - $bool = (\{true, false\}, \{and, or, not\})$, ahol
 $and: bool \times bool \rightarrow bool, \dots$
 - $int = (\{-2147483648, \dots, 2147483647\}, \{+, -, \cdot, /, \dots\})$, ahol \dots

Adattípusok megvalósítása elemi eszközökkel

Adattípusok

- A programozási nyelvek számos típussal rendelkeznek
 - a gépi szinten számára értelmezhető, egyszerű típusokat nevezzük *elemi*, vagy *primitív típusoknak*
 - a további típusok az *összetett típusok*, amelyek a *típuskonstrukciók* segítségével hozhatóak létre, úgymint:
 - *iterált*, vagy *sorozat* (\mathbb{D}^n , vagy \mathbb{D}^*), amely azonos típusú elemek sorozatát adja (pl. tömb)
 - *direktszorzat* ($\mathbb{D}_1 \times \mathbb{D}_2$), amely különböző típusú adatok egymástól független kombinációját adja (pl. rekord)
 - *unió* ($\mathbb{D}_1 \cup \mathbb{D}_2$), amely különböző típusú adatok egymástól függő kombinációját adja

Adattípusok megvalósítása elemi eszközökkel

Rekordok

- *Rekord*nak nevezzük több, egymástól független, és akár különböző típusú adat gyűjteményét, amelyeket egy egységben, illetve részeiben is tudunk kezelni
 - pl.:
 - egy komplex szám valós és képzetes részből áll
 - egy 3 dimenziós koordináta 3 számból (x, y, z) áll
 - egy dátum évet, hónapot, napot jelent
 - egy telefonkönyvi bejegyzésben név, cím és szám szerepel
 - az iterált típushoz hasonlóan alapvetően egy értékthalmazt definiál, amelyet a későbbiekben kiegészíthetünk művelethalmazzal

Adattípusok megvalósítása elemi eszközökkel

Rekordok

- A rekordba helyezett értékeket nevezzük a rekord *mezőinek*, vagy *adattagjainak*
 - mező lehet bármilyen, már deklarált típus (egyszerű, összetett, illetve más rekord is)
 - a mező automatikusan létrejön, külön lekérdezhető és módosítható
- Rekordokat C++-ban a **struct** kulcsszóval hozhatunk létre
 - a rekordok új típusok lesznek, amelyek tetszőlegesen felhasználhatóak
 - a rekordokat általában globálisan deklaráljuk (tehát minden blokkon kívül, a program elején)

Adattípusok megvalósítása elemi eszközökkel

Rekordok

- A rekord szerkezete C++-ban:

```
struct <rekord név>
{
    // blokkban adjuk meg a mezőket
    <típus 1> <mezőnév 1>;
    <típus 2> <mezőnév 2>;
    ...
}; // a végén le kell zárni
```

- Pl. a komplex szám, mint két valós szám:

```
struct Complex { // komplex szám rekordja
    float re; // valós rész
    float im; // képzetes rész
};
```

Adattípusok megvalósítása elemi eszközökkel

Rekordok

- A deklarációt követően felhasználhatjuk, mint bármelyik típust, létrehozhatunk változókat és konstansokat is belőle
 - a mezőkre tudunk külön hivatkozni, így egyenként módosíthatjuk, lekérdezhetjük az értékeiket *<változónév>.<mezőnév>* formában

- pl.:

```
Complex c; // komplex szám létrehozása
c.re = 0; // a mezőket külön beállíthatjuk ...
c.im = 1;
cout << c.re << " + i*" << c.im << endl;
    // ... és lekérdezhetjük
...
vector<Complex> cv; // típusként viselkedik
```

Adattípusok megvalósítása elemi eszközökkel

Példák

Feladat: Valósítsuk meg a komplex számok típusát. Olvassunk be 10 komplex számot billentyűzetről, majd írjuk ki az összegüket.

- valósítsuk meg a komplex számok típusát rekord segítségével, amely a valós és képzetes részt tartalmazza
- vegyük fel a komplex számok összeadásának műveletét (**add**), amely az egyetlen típusműveletünk lesz
- a számokat egy vektorban tároljuk
- további alprogramok segítségével valósítjuk be a számok beolvasását (**read**), valamint a vektor elemeinek összegzését (**sum**)

Adattípusok megvalósítása elemi eszközökkel

Példák

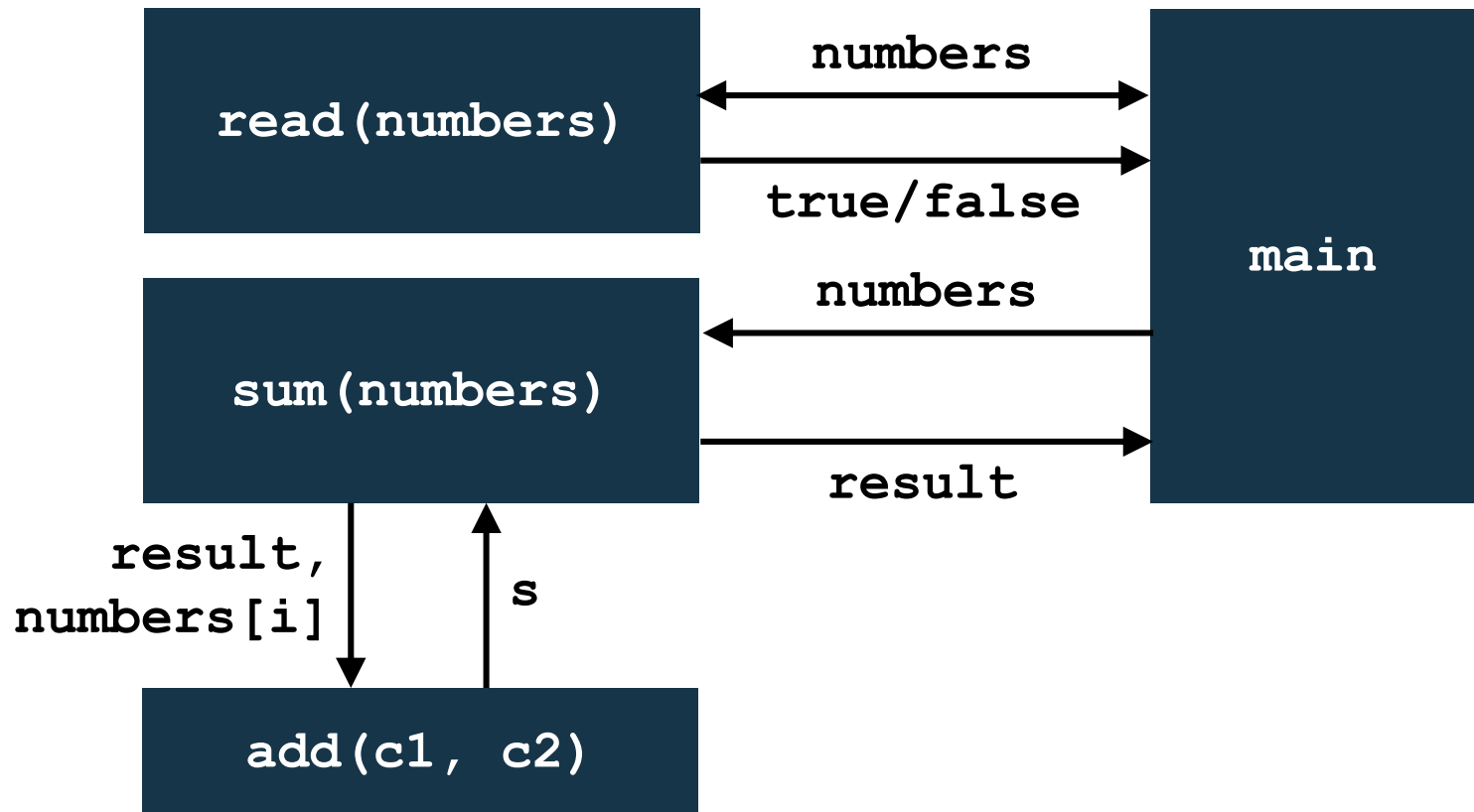
Tervezés:

- elméletben: $\mathbb{C} = (\mathbb{R} \times \mathbb{R}, \{+, -, \cdot, /\})$, ahol
 $+: \mathbb{C} \times \mathbb{C} \rightarrow \mathbb{C}$ az összeadás,
 $+(re_1 \times im_1, re_2 \times im_2) = (re_1 + re_2) \times (im_1 + im_2)$
- megvalósításban: $Complex = (float \times float, \{add\})$,
 ahol
 $add: Complex \times Complex \rightarrow Complex$ az összeadás,
 $add(re_1 \times im_1, re_2 \times im_2) = (re_1 + re_2) \times (im_1 + im_2)$

Adattípusok megvalósítása elemi eszközökkel

Példák

Tervezés:



Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
// a komplex típus
struct Complex {
    float re; // valós rész
    float im; // képzetes rész
};

Complex add(Complex c1, Complex c2);
// komplex számok összeadása

bool read(vector<Complex> numbers);
// számok beolvasása

Complex sum(vector<Complex> numbers);
// számok összegzése
```

Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
int main() {
    vector<Complex> numbers(10);
    if (!read(numbers)) {
        cout << "A számok beolvasása sikertelen!"
             << endl;
        return 1;
    }

    Complex result = sum(numbers);
    cout << "Az összeg: " << result.re
         << " + i*" << result.im;
    return 0;
}
```

Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
Complex sum(vector<Complex> numbers) {
    Complex result;
    result.re = 0; // eredmény kinullázása
    result.im = 0;

    for (unsigned int i = 0; i < numbers.size();
        i++){
        // összegzés tagonként
        result = add(result, numbers[i]);
    }
    return result;
}
```

Adattípusok megvalósítása elemi eszközökkel

Példák

Feladat: Valósítsunk meg egy névjegyzéket, ahol a bejegyzés névből, és születési dátumból áll, a születési dátum pedig évből, hónapból és napból. A program olvasson be tetszőleges számú adatot egy fájlból, és legyen lehetőségünk a mai napon született emberek neveit lekérdezni.

- valósítsuk meg a dátum (**Date**) és a bejegyzés (**Record**) típusokat, és ezek felhasználásával oldjuk meg a feladatot
- a fájlnevet adjuk meg argumentumként, feltételezzük, a fájl minden sora a következő szerkezetű:
<vezetéknév> <keresztnev> <év>.<hó>.<nap>
- a beolvasást ennek megfelelően végezzük, hibaellenőrzéssel

Adattípusok megvalósítása elemi eszközökkel

Példák

- a `readFile` függvény dolgozza fel a fájlt, az eredményt cím szerint kezelt vektorban adja vissza
- a `findBirthDay` függvény megkeresi (pontosabban kiválogatja) a szülinaposokat a kapott paraméter alapján (átadjuk a vektort, illetve a dátumot is)
- tágabb értelemben ezek is tekinthetők típusműveletnek

Tervezés:

- rekord típusa: $Record = (string \times Date, \{readFile, findBirthDay\})$
- dátum típusa: $Date = (int \times int \times int, \{findBirthDay\})$

Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
struct Date { // dátum típusa
    int year;
    int month;
    int day;
};
```

```
struct Record { // névjegyzék típusa
    string name;
    Date birthDate; // felhasználjuk a dátumot
};
```

```
bool readfile(string, vector<Record>&);
// adatok beolvasása fájlból
```


Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
vector<string> findBirthDay(vector<Record>, Date);  
    // szülinaposok keresése
```

```
int main() {  
    vector<Record> data; // adatok  
  
    if (!readFile("adatok.txt", data)) { ... }  
  
    Date d; // mai dátum  
    d.month = 10;  
    d.day = 8;
```

Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
vector<string> names = findBirthDay(data, d);  
    // kigyűjtés  
  
cout << "Ma ünnepli születésnapját: " << endl;  
for (int i = 0; i < names.size(); i++)  
    cout << names[i] << endl; // kiírás  
  
return 0;  
}
```

Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
vector<string> findBirthDay(vector<Record> data,
                           Date d) {
    vector<string> names;
    for (int i = 0; i < data.size(); i++) {
        if (data[i].birthDate.month == d.month &&
            data[i].birthDate.day == d.day) {
            // ha a hónap és a nap stimmel
            names.push_back(data[i].name);
        }
    }
    return names;
}
```

Adattípusok megvalósítása elemi eszközökkel

Újrafelhasználhatóság

- Hasonlóan az alprogramjainkhoz, a típusokat is úgy kell megvalósítanunk, hogy azok minél jobban újrafelhasználhatóak legyenek
 - a rekordon túl megfelelő művelethalmazzal kell ellátnunk a típust, ahol az alprogramok minél általánosabban megfogalmazottak, biztosítanak hibaellenőrzést, és használatuk kényelmes
 - tágabb értelemben a művelethalmaznak azon alprogramok gyűjteményét tekintjük, amelyek deklarációjában szerepel az adott típus
 - pl. paraméterként, vagy visszatérési értéként
 - egy művelet akár több művelethalmazba is bekerülhet

Adattípusok megvalósítása elemi eszközökkel

Példák

Feladat: Készítsük el a racionális szám típusát a beolvasás, kiírás, összeadás és szorzás műveletével. A főprogramban olvassunk be 5 racionális számot, és adjuk meg a szorzatukat és összegüket.

- készítsünk rekordot a racionális számnak (**Rational**), amely tartalmazza a számlálót (**nominator**) és a nevezőt (**denominator**)
- a beolvasás (**read**) és kiírás (**write**) függvénye kapja meg cím szerinti paraméterátadással az aktuális racionális számot, az összeadás (**add**) és szorzás (**multiply**) paraméterben kapjon két racionális számot, és a visszatérési értékük legyen az eredmény

Adattípusok megvalósítása elemi eszközökkel

Példák

Tervezés:

- $Rational = \left(\begin{array}{l} int \times (int \setminus \{0\}), \\ \{add, multiply, read, write\} \end{array} \right)$

Megoldás:

```
// racionális szám típusa:  
struct Rational { // racionális szám rekordja  
    int numerator;  
    int denominator;  
};  
  
void read(Rational &r) { // beolvasás  
    cin >> r.numerator >> r.denominator;  
}
```

Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
void write(Rational r) { // kiírás
    cout << r.numerator << "/" << r.denominator;
}
```

```
Rational multiply(Rational r1, Rational r2)
{ // szorzás két racionális szám között
    Rational res; // eredmény változó
    res.numerator = r1.numerator * r2.numerator;
    res.denominator =
        r1.denominator * r2.denominator;
    // az eredménybe végezzük el a műveleteket
    return res; // az eredményt adjuk vissza
}
```

Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
Rational add(Rational r1, Rational r2)
{
    // összeadás két racionális szám között
    Rational res;
    res.numerator =
        r1.numerator * r2.denominator +
        r2.numerator * r1.denominator;
    res.denominator =
        r1.denominator * r2.denominator;
    return res;
}
```


Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
int main() { // főprogram
    Rational t[5];
    // 5 elemű tömb racionális számokból
    cout << "Számok megadása: " << endl;
    for (int i = 0; i < 5; i++)
        read(t[i]);
    // használjuk a beolvasó függvényt

    Rational sum, mult; // eredmények
    sum.numerator = 0;
    sum.denominator = 1;
    // a 0 racionális szám a 0/1
```

Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
mult.numerator = mult.denominator = 1;
    // 1/1 racionális szám

for (int i = 0; i < 5; i++){ // műveletvégzés
    sum = add(sum, t[i]);
    mult = multiply(mult, t[i]);
}
cout << "Összeg: "; write(sum); cout << endl;
cout << "Szorzat: "; write(mult);
cout << endl;
return 0;
}
```

Adattípusok megvalósítása elemi eszközökkel

Operátorok saját típusokra

- A C++ lehetőséget biztosít, hogy saját típusainkhoz saját operátorokat készítsünk, amelyek a beépített operátorok módjára hívhatóak meg
 - az operátorok is ugyanúgy alprogramok, de speciális névvel rendelkeznek, és szabályozott a használatuk módja
 - a elnevezésként `operator<jel>`-t kell megadnunk, ahol a jel helyén az operátor jelét kell megadnunk (pl. `operator+`)
- Az operátorok esetében
 - a paraméterek száma és a visszatérési érték megléte kötött
 - a paraméterek és a visszatérési érték típusa viszont tetszőlegesen szabályozható

Adattípusok megvalósítása elemi eszközökkel

Operátorok saját típusokra

- Azonban nem valósítható meg olyan operátor, amely már létezik (pl. összeadás egész számok között)
- Pl. az összeadás műveletét megadhatjuk a komplex számra:

```
Complex operator+(Complex c1, Complex c2) {  
    ... // alprogram törzse  
    return s;  
    // a típusnak megfelelő értékkel térünk vissza  
}
```
- A deklarációt követően használhatjuk az operátort:

```
Complex a, b;  
...  
Complex c = a + b; // a fenti művelet hívódik meg
```

Adattípusok megvalósítása elemi eszközökkel

Operátorok saját típusokra

- A következő operátorokat készíthetjük el saját típusainkhoz:
 - 1 paraméterrel:
 - matematikai: +, -, ++, --
 - logikai: !
 - 2 paraméterrel (ekkor az első paraméter lesz a kifejezés balértéke, a második a jobbértéke):
 - matematikai: +, -, *, /, %
 - logikai: ||, &&, ^, &, |
 - értékadás: +=, -=, *=, /=
 - összehasonlítás: <, >, <=, >=, !=, ==
 - adatmozgatás: <<, >>

Adattípusok megvalósítása elemi eszközökkel

Példák

Feladat: Valósítsuk meg az előbbi racionális típust úgy, hogy az összeadás és szorzás műveleteit operátorral fogalmazzuk meg.

- a két függvénynevet lecseréljük operátorra, a működésük marad az előbbi
- a főprogramban már operátorként használjuk őket

Tervezés:

- $Rational = (int \setminus (int \setminus \{0\}), \{+, \cdot, read, write\})$, ahol
 $+$: $Rational \times Rational \rightarrow Rational$,
 \cdot : $Rational \times Rational \rightarrow Rational$

Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
Rational operator+(Rational r1, Rational r2) {  
    ...  
    return res;  
}
```

```
Rational operator*(Rational r1, Rational r2) {  
    ...  
    return res;  
}
```

```
...  
sum = sum + t[i];  
mult = mult * t[i];  
// az általunk írt operátorok hívódnak meg
```

Adattípusok megvalósítása elemi eszközökkel

Adatfolyam operátorok

- Az adatfolyamokat (pl. konzol, fájl, szövegfolyam) a C++ együttesen kezeli, ezért hasonló utasításkészlettel rendelkeznek
- A legtöbb adatfolyam tekintetében megkülönböztetjük a bemenő, illetve kimenő adatfolyamot, amelyek közös *gyűjtőnévvel* rendelkeznek
 - a bemenő adatfolyam gyűjtőneve az **`istream`**, a kimenő adatfolyamé az **`ostream`**
 - megtalálhatóak az **`<iostream>`** fájlban
 - pl. egy **`f`** fájl (ha **`f`** **`ifstream`** típusú) és a **`cin`** speciális esete az **`istream`**-nek, ezért tudjuk például mindkettővel használni a **`getline`** függvényt

Adattípusok megvalósítása elemi eszközökkel

Adatfolyam operátorok

- Lehetőségünk van adatfolyam operátorok (<<, >>) definiálására, amelyek adatfolyamokra tudnak írni, illetve onnan olvasni értékeket, így a saját típusainkat is egyszerűen tudjuk az adatfolyamokon kezelni
- Ezek az operátorok *speciális paraméterezést* igényelnek:
 - paraméterben meg kell kapniuk az adatot, valamint az adatfolyamot, ahol a műveletet el kell végezniük
 - a visszatérési érték szintén az adatfolyam
 - az adatfolyam lehet bemeneti (**istream**) és kimeneti (**ostream**), amelyek csak cím szerinti paraméterátadással adhatóak át

Adattípusok megvalósítása elemi eszközökkel

Adatfolyam operátorok

- Az operátor belsejében a paraméterben átadott adatfolyamot használhatjuk a beolvasásra és kiírásra, valamint a rekordunk mezőire már elkészített operátorokat (<<, >>, de lehet pl. `getline`-t is)
 - természetesen a beolvasás, kiírás módja tetszőleges

- Pl. a kiírás operátora komplex számra:

```
ostream& operator<<(ostream& s, Complex c)
{
    // s lesz az adatfolyam, tehát azt használjuk
    s << c.re << " + i*" << c.im;
    // operátorok már a mezők típusára futnak le
    return s; // adatfolyam visszaadása
}
```

Adattípusok megvalósítása elemi eszközökkel

Példák

Feladat: Írjuk át a racionális szám típusunkat úgy, hogy a beolvasás és kiírás a megszokott operátorokon keresztül történjen, `<számláló>/<nevező>` formában.

- a többi alprogramot változatlanul hagyjuk, a főprogramban pedig meghívjuk ezeket az operátorokat

Megoldás:

```
istream& operator>>(istream& s, Rational &r) {  
    // s lesz az adatfolyam  
    char ch; // a / jelet átugorjuk  
    s >> r.numerator >> ch >> r.denominator;  
    // s-ről beolvassuk a két értéket  
    return s; // s-t visszaadjuk  
}
```

Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
ostream& operator<<(ostream &s, Rational r){
    s << r.numerator << "/" << r.denominator;
    // s-re kiírjuk a megfelelő alakot
    return s;
}
...
for (int i = 0; i < 5; i++)
    cin >> t[i]; // saját operátorunk használata
...
cout << "Összeg:" << sum << endl;
cout << "Szorzat:" << mult << endl;
...
```

Adattípusok megvalósítása elemi eszközökkel

Operátorok túlterhelése

- Az operátorok ugyanúgy túlterhelhetők, mint más függvények
 - pl.: készíthetünk összeadást két racionális szám között, illetve egy racionális és egy egész szám között is
- Amennyiben valamilyen paraméterezéssel készítünk kétváltozós operátort, akkor a paraméterek a megadás sorrendjének megfelelően helyettesítődnek be
 - ha kommutatív műveletet szeretnénk megvalósítani, akkor mindkét sorrendben el kell készíteni a túlterhelése
 - pl. egész és racionális, valamint racionális és egész paraméterekkel is elkészíthetjük az összeadást, így kommutatív lesz

Adattípusok megvalósítása elemi eszközökkel

Értékadások

- Az alapvető értékadás operátor ($=$) alapértelmezés szerint létezik minden típusra, ezért ezt nem kell definiálnunk (de felüldefiniálhatjuk speciális módon, ha szükséges)
- Ugyanakkor lehetőségünk speciális értékadás operátorok ($+=$, $*=$, ...) definiálására, amelyek célja az értékmódosítás rövidítése
 - a speciális értékadás nem ekvivalens az adott operátor és az értékadás együttes használatával (pl. $+=$ nem ugyanaz, mint $=$ és $+$)
 - a szabványos működéshez az első paramétert cím szerint kell átadnunk, majd módosítanunk kell, és visszatérési értéként is azt kell visszaadnunk

Adattípusok megvalósítása elemi eszközökkel

Példák

Feladat: Készítsünk a racionális típusnak egészzel történő összeadás és szorzás műveleteket is, továbbá készítsünk értékadással kombinált operátorokat is. A főprogramban az eredményhez adjunk kettőt, a szorzatot pedig szorozzuk meg kettővel.

- így egy függvénynek három túlterhelése lesz
- a főprogramban mindkét változat meghívásra kerül

Megoldás:

```
// összeadás további változatai:
```

```
Rational operator+(Rational r, int e) { ... }
```

```
Rational operator+(int e, Rational r) { ... }
```

Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
// szorzás további változatai:
```

```
Rational operator*(Rational r,){ ... }
```

```
Rational operator*(int e, Rational r){ ... }
```

```
// értékadással kombinálva:
```

```
Rational operator+=(Rational r1, Rational r2){ ... }
```

```
Rational operator+=(Rational r, int e){ ... }
```

```
int operator+=(int e, Rational r){ ... }
```

```
Rational operator*=(Rational r1, Rational r2){ ... }
```

```
Rational operator*=(Rational r, int e){ ... }
```

```
int operator*=(int e, Rational r){ ... }
```


Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
...  
for (int i = 0; i < 5; i++){  
    sum += t[i];  
    sum *= t[i];  
    // összevonhatjuk az értékadással  
}  
...  
cout << "Összeg:" << sum + 2 << endl;  
cout << "Szorzat:" << sum * 2 << endl;  
    // ugyanígy jó lenne (2 + sum) és (2 * mult)  
    // is, mivel adott a másik túlterhelés is
```

Adattípusok megvalósítása elemi eszközökkel

Konstruktor műveletek

- Általában a rekordok létrehozása, és értékeinek beállítása elég körülményes, mivel minden mezőjének külön kell értéket adni
- Ez egyszerűsíthető, ha készünk egy beállító alprogramot, amely bizonyos paraméterek alapján beállítja a rekord tagjait
 - nem kötelező annyi paramétert adnunk, ahány tag van, bizonyos változóknak adhatunk alapértelmezett értéket
 - ellenben ezt a műveletet is meg kell hívunk valahol
- Van egy olyan művelet, amely dedikáltan (és automatikusan) akkor hívódik meg, amikor létrehozunk egy változót a rekordból, ez a *konstruktor* művelet

Adattípusok megvalósítása elemi eszközökkel

Konstruktor műveletek

- A konstruktor olyan alprogram, amely:
 - a rekord belsejében definiálható, a rekord változói mellett
 - tetszőlegesen paraméterezhető és túlterhelhető
 - arra szolgál, hogy a rekord mezőit kezdő értékekkel lássa el (*inicializálja* a tagokat)
 - mindig automatikusan meghívódik példány létrehozásakor, paraméteres esetben ekkor átadhatunk neki értékeket
 - neve megegyezik a rekorddal, és mivel létrehozza a példányt, a visszatérési értéke is, amit külön nem írunk le
 - igazából akkor is létezik, amikor nem írjuk meg, csak ekkor nem végez semmilyen inicializáló tevékenységet

Adattípusok megvalósítása elemi eszközökkel

Konstruktor műveletek

```
struct <rekordnév> {  
    <típus> <tag>;  
  
    <rekordnév>() { <tag> = <érték>; }  
    <rekordnév>(<típus> <változó>) {  
        <tag> = <változó>;  
    } // konstruktorok  
};  
  
<rekordnév> <változónév>;  
// ekkor lefut a 0 paraméteres konstruktor  
  
<rekordnév> <változónév>(<érték>);  
// ekkor lefut az 1 paraméteres konstruktor,  
// amennyiben a típusok egyeznek
```

Adattípusok megvalósítása elemi eszközökkel

Példák

Feladat: Egészítsük ki a racionális szám típusát konstruktorokkal.

- három konstruktort valósítunk meg:
 - egy 0 paraméterest, amely beállítja az alapértelmezett 0 (0/1) racionális értéket
 - egy 1 paraméterest, amely egy egész számból tud racionális számot létrehozni
 - egy 2 paraméterest, amely a megadott paraméterek (számláló, nevező) alapján hozza létre a hányadost, ebben az esetben a második paraméter értékét ellenőrizzük is

Adattípusok megvalósítása elemi eszközökkel

Példák

Tervezés:

- $Rational = \left(\begin{array}{c} int \times (int \setminus \{0\}), \\ \{+, \cdot, + =, \cdot =, \gg, \ll, Rational\} \end{array} \right)$, ahol

$+: Rational \times Rational \rightarrow Rational,$

$+: Rational \times int \rightarrow Rational,$

$+: int \times Rational \rightarrow Rational,$

...

$Rational: \rightarrow Rational,$

$Rational: int \rightarrow Rational,$

$Rational: int \times int \rightarrow Rational$

Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

```
struct Rational {  
    ...  
    Rational() { // 0 paraméteres konstruktor  
        numerator = 0; denominator = 1;  
    }  
    Rational(int nr) { // 1 paraméteres  
        numerator = nr; denominator = 1;  
    }  
    Rational(int num, int denom) { // 2 paraméteres  
        numerator = num;  
        denominator = denom != 0 ? denom : 1;  
    }  
};
```

Adattípusok megvalósítása elemi eszközökkel

Példák

Megoldás:

...

```
Rational sum;
```

```
    // 0 paraméteres konstruktor hívása, vagyis
```

```
    // 0/1-t kapunk
```

```
    // ugyanez:
```

```
    // Rational sum(0);
```

```
    // Rational sum(0, 1);
```

```
Rational mult(1);
```

```
    // 1 paraméteres konstruktor hívása, vagyis
```

```
    // 1/1-t kapunk
```

```
    // ugyanez egy sorban:
```

```
    // Rational sum, mult(1);
```