Pázmány Péter Katolikus Egyetem
Információs Technológiai és Bionikai Kar

Bevezetés a Programozásba II

2. előadás

Adattípusok megvalósítása egységbe zárással

© 2014.02.17. Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

Adattípusok megvalósítása egységbe zárással
Adattípusok

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

Adattípusok megvalósítása egységbe zárással
Adattípusok

- A programozási nyelvek számos típussal rendelkeznek
  - a gépi szinten számára értelmezhető, egyszerű típusokat nevezzük *elemi*, vagy *primitív típusok*nak
  - a további típusok az *összetett típusok*, amelyek a *típuskonstrukció*k segítségével hozhatóak létre, úgymint:
    - *iterált*, vagy *sorozat* ($\mathbb{D}^n$, vagy $\mathbb{D}^*$), azonos típusú elemek sorozatát adja gyakran használt (pl. tömb)
    - *direktszorzat* ($\mathbb{D}_1 \times \mathbb{D}_2$), különböző típusú elemek együttes kombinációja, gyakran használt (pl. rekord)
    - *unió* ($\mathbb{D}_1 \cup \mathbb{D}_2$), értékei különböző típusok értékei lehetnek, ritkán használt

Adattípusok megvalósítása egységbe zárással
Adattípusok elemi eszközökkel

- A típuskonstrukció megadja az értékhalmazt, pl.:

```
struct Rational { // racionális szám rekordja
   int numerator;
   int denominator;
};
```

- Emellett tetszőleges műveleteket valósíthatunk meg alprogramok segítségével, pl.:

```
Rational operator*(Rational r1, Rational r2) {
   // két racionális szám szorzata
   Rational res; // eredmény változó
   res.numerator = r1.numerator * r2.numerator;
   …
   return res; // az eredményt adjuk vissza
}
```


Adattípusok megvalósítása egységbe zárással
Példa

Feladat: Valósítsuk meg a téglalapok típusát. A téglalap szélességével, illetve magasságával reprezentálható, illetve lekérdezhető a területe, valamint a kerülete.

- létrehozzuk a téglalap rekordját (`Rectangle`), amely tartalmazza a két adatot (`width`, `height`)
- a rekord mellé megvalósítjuk a két lekérdező műveletet (`area`, `perimeter`), amelyek paraméterben kapják meg a téglalapot, és visszatérési értékben adják meg az eredményt
- a könnyű használhatóság végett a beolvasó (`>>`), illetve kiíró (`<<`) operátorokat is megvalósítjuk
- a főprogramban beolvasunk egy téglalapot, majd kiírjuk az értékeit

Adattípusok megvalósítása egységbe zárással
Példa

Megoldás:

```
struct Rectangle { // téglalap rekordja
   double width; // szélesség
   double height; // magasság
};
double area(Rectangle rec) { // téglalap területe
   return rec.width * rec.height;
}
…
istream& operator>>(istream& s, Rectangle& rec) {
   // beolvasó operátor
   s >> rec.width >> rec.height; return s;
}
…
```

Adattípusok megvalósítása egységbe zárással
Példa

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 (`numerator`) és a nevezőt (`denominator`)
- a műveleteket, beleértve a beolvasást és kiírást ,operátorok segítségével valósítjuk meg (`+`, `*`, `<<`, `>>`), ügyelve arra, hogy a racionális számok egész számokkal is kompatibilisek (így többféle művelet kell), illetve, hogy a műveletek összevonhatóak értékadással (`+=`, `*=`)

Adattípusok megvalósítása egységbe zárással
Példa

Megoldás:

```
struct Rational { // racionális szám rekordja
   int numerator;
   int denominator;
};

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


Adattípusok megvalósítása egységbe zárással
Példa

Megoldás:

```
Rational operator*(Rational r1, Rational r2) { … }
Rational operator*(Rational r, int e) { … }
Rational operator*(int e, Rational r) { … }
…

// összeadás értékadással:
Rational operator+=(Rational& r1, Rational r2)
{ … }
Rational operator+=(Rational& r, int e) { … }
int operator+=(int& e, Rational r) { … }
…
```


Adattípusok megvalósítása egységbe zárással
Adattípusok elemi eszközökkel

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

Adattípusok megvalósítása egységbe zárással
Az egységbe zárás

- A típushoz megadott művelethalmazt célszerű szorosan összefűzni az értékhalmazzal, hogy valóban együttest képezzen, ezt *egységbe zárás*nak (*enkapszulációnak*) nevezzük
  - a rekordot összeillesztjük a hozzá tartozó művelethalmazzal, így alkotva az egységes típust
  - a típus adatai a *mezők* (vagy *attribútumok*), műveletei a *metódusok* lesznek
  - a metódusok közvetlenül láthatják a mezőket, így nincs szükség paraméterekre
- A típushoz létrehozásához továbbra is a `struct` konstrukciót használjuk, de alprogramokat is helyezünk az adatok mellé

Adattípusok megvalósítása egységbe zárással
Metódusok használata

- A metódusok deklarációja hasonlít a sima függvénydeklarációhoz, csak a rekord belsejében végezzük:

```
struct <rekordnév>{
   …
   <típus> <metódusnév>(<paraméterek>){
      <metódustörzs>
   }
};
```

- A metódusokat csak változón keresztül érhetjük el, ezért mindig kell legyen egy példány a típusból, ekkor `<változónév>.<metódusnév>(<paraméterek>)` formában meghívhatjuk a függvényt

Adattípusok megvalósítása egységbe zárással
Példa

Feladat: Módosítsuk a téglalap megvalósítását úgy, hogy a kerület és terület lekérdezés metódus legyen.

- nincs szükségünk paraméterátadásra, a művelet közvetlenül futtatható a változóra
- a formális paraméter helyett közvetlenül használhatjuk a mezőket

Megoldás:

```
struct Rectangle { // téglalap típusa
   double width; // szélesség (mező)
   double height; // magasság
```


Adattípusok megvalósítása egységbe zárással
Példa

Megoldás:

```
    double area() { // téglalap területe (metódus)
      // a paraméterre már nincs szükségünk
      return width * height;
         // a mezők közvetlenül elérhetőek
    }
    double perimeter() {
       // téglalap kerülete (metódus)
       return (width + height) * 2;
    }
};

…
```


Adattípusok megvalósítása egységbe zárással
Példa

Megoldás:

```
int main()
{
    Rectangle rec1;
    …
    cout << "Area: " << rec1.area() << endl;
       // további műveletek használata
       // (metódusként)
    cout << "Perimeter: " << rec1.perimeter()
         << endl;

    return 0;
}
```


Adattípusok megvalósítása egységbe zárással
Konstruktor műveletek

- Amikor a példányosítunk egy típust, kezdeti értékeket kell beállítanunk a mezőinek, ezt *inicializálás*nak nevezzük
  - pl. négyzet méretei, racionális szám számlálója, nevezője
  - amennyiben ezt kihagyjuk, a példány nem megfelelő kezdőállapottal indul (mint egy változó kezdeti értékadás nélkül)
- Az inicializálás automatikusan elvégezhető *konstruktor* művelet használatával
  - egy olyan metódus, amely mindig példányosításkor fut le, és feladata a kezdeti értékek beállítása a mezőknek
  - neve megegyezik a típussal, visszatérési értéke nincs

Adattípusok megvalósítása egységbe zárással
Konstruktor műveletek

- A konstruktorra ugyanazon szabályok vonatkoznak, mint más metódusokra (paraméterezhető, túlterhelhető, látja a mezőket):

```
struct <típusnév>{
   <típus> <mező>;

   <típusnév>() { <tag> = <érték>; }
      // 0 paraméteres konstruktor

   <típusnév>(<típus> <változó>) {
      // 1 paraméteres konstruktor (túlterhelés)
      <mező> = <változó>;
   }
   … // egyéb metódusok
};
```


Adattípusok megvalósítása egységbe zárással
Konstruktor műveletek

- A konstruktorra futtatása automatikusan történik példányosításkor, de amennyiben paraméteres, akkor paraméterrel kell a példányt létrehozni:

```
<rekordnév> <változónév>;
// ekkor lefut a 0 paraméteres konstruktor

<rekordnév> <változónév>(<érték>);
// ekkor lefut az 1 paraméteres konstruktor,
// amennyiben a típusok egyeznek
```

- Ha nem hozunk létre saját konstruktort, a típus egy alapértelmezett (0 paraméteres) konstruktort kap
  - bármilyen konstruktor létrehozása esetén az alapértelmezett megszűnik

Adattípusok megvalósítása egységbe zárással
Példa

Feladat: Egészítsük ki a racionális szám típusát három konstruktorral, egy 0 paraméteressel, amely a 0 racionális számot állítja elő, illetve egy 1 paraméteressel, amely egy egész szám alapján hozza létre a racionálist, illetve 2 paraméteresen, amely nevezőt és számlálót kap.

- 2 paraméteresnél ellenőrizzük, hogy a nevező megfelelő-e

Megoldás:

```
struct Rational {
   // racionális szám rekordja
   int numerator;
   int denominator; // mezők
```


Adattípusok megvalósítása egységbe zárással
Példa

Megoldás:

```
   // konstruktorok:
   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 egységbe zárással
Példa

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);
```


Adattípusok megvalósítása egységbe zárással
Érték inicializálás

- A kezdő értékeknek van egy még speciálisabb megadási lehetősége, az *érték inicializálás*
  - kettőspont után megadjuk a mező nevét, majd a formális paramétert zárójelben, ha többet akarunk megadni, akkor vesszővel választjuk el:

```
<típusnév>(<típus> <paraméter>) :
   <mezőnév>(<kezdőérték>), // inicializálások
   <mezőnév>(<kezdőérték>), … {
    … // konstruktortörzs
}
```

  - hivatkozás mezők (`&`) esetén szükséges, alapesetben használata egyenértékű a konstruktortörzsben történő értékadással

Adattípusok megvalósítása egységbe zárással
Deklaráció és definíció szétválasztása

- Metódusok (beleértve a konstruktort is) esetén is szeparálhatjuk a deklarációt a definíciótól, ekkor a rekordba csak a függvény szintaxisát írjuk, a törzsét később
  - a törzsnél meg kell jelölnünk, melyik típus tagja a `::` (scope) operátor segítségével:

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


Adattípusok megvalósítása egységbe zárással
Példa

Feladat: Módosítsuk a téglalap típusát úgy, hogy adunk hozzá érték inicializálást használó konstruktorokat, illetve a metódusok törzsét leválasztjuk.

Megoldás:

```
struct Rectangle { // téglalap típusa
   …
   Rectangle() : width(0), height(0) { }
      // konstruktor művelet, amelyben érték
      // inicializálást használunk
   Rectangle(double w, double h) :
      width(w), height(h) { }
      // 2 paraméteres konstruktor művelet
```

Adattípusok megvalósítása egységbe zárással
Példa

Megoldás:

```
    double area();
       // téglalap területe (metódus deklarációja)
    double perimeter();
       // téglalap kerülete (metódus deklarációja)
};

double Rectangle::area()
    // téglalap területe (metódus definíciója)
{
    return width * height;
       // a mezők közvetlenül elérhetőek
}
…
```


Adattípusok megvalósítása egységbe zárással
Operátorok metódusként

- Lehetőségünk van operátorok megfogalmazására egységbe zárás mellett is, ekkor egy speciális szabály, hogy az operátor első paramétere az a típuspéldány lesz, amelybe beágyaztuk
  - azaz 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, bizonyos speciális értékadások) csak a típuson kívül írhatóak meg
  - általában az operátorokat külön szoktuk deklarálni a típusokhoz

Adattípusok megvalósítása egységbe zárással
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);
```


Adattípusok megvalósítása egységbe zárással
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);
…
```