Eötvös Loránd Tudományegyetem
Informatikai Kar

Objektumelvű alkalmazások fejlesztése

2. gyakorlat

Adattípusok megvalósítása

© 2011.09.28. Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

Adattípusok megvalósítása
Az adattípus fogalma

- *Adattípus* = *értékhalmaz* (típusban használt értékek összessége) + *műveléthalmaz* (típus elemein értelmezett eljárások, függvények)
  - pl.: az `int` típus esetén az értékhalmaz adott határok közötti egész számok, műveléthalmaz az értékadás és a matematikai műveletek
- A programozási nyelvek lehetőséget adnak a beépített típusok mellett *saját típusok* létrehozására
  - a beépített típusokat a létező típusok használatával hozzuk létre
  - az értékhalmazt a már meglévő típusokból határozzuk meg típuskonstrukciók segítségével (tehát az eddigi típusokat rakjuk halmazba, alkotunk belőle tömböt, …)

Adattípusok megvalósítása
Rekordok használata

- a műveléthalmazt alkotó függvényekben az értékhalmazon végzünk műveleteket
- A C++-ban adattípus konstrukciójához használt eszköz a *rekord*
  - a rekordban tetszőleges típusú változókat tárolhatunk tetszőleges kombinációban (ezekből tömböket, illetve más adatszerkezeteket is eltárolhatunk)
  - írhatunk függvényeket, amelyek ezeket a változókon végeznek műveleteket (ez eddig annyit jelentett, hogy a függvény első paraméterében megkapta azt a rekordtípusú változót, amin dolgoznia kellett)
  - a függvények megjelenhetnek operátorként is, ami kényelmesebb használatot tesz lehetővé

Adattípusok megvalósítása
Rekordok használata

- Pl. a komplex szám típusa:

```
struct Complex{ // rekord
   double Re; // valós rész
   double Im; // képzetes rész
};
Complex Zero(){ // a 0 komplex szám létrehozása
   Complex res; res.Re = 0; res.Im = 0;
   return res;
}
Complex FromDoubles(double re, double im){
   // létrehozás valósakból
   Complex res; res.Re = re; res.Im = im;
   return res;
}
```


Adattípusok megvalósítása
Rekordok használata

```
double ToDouble(Complex nr){ // valóssá alakítás
   return nr.Re;
}
void Conjugate(Complex& nr){ // konjugálás
   nr.Im = -nr.Im;
}
//... további műveletek

// használat:
Complex x = Zero(); // 0+0i lesz
x = FromDoubles(3.5, -3); // 3.5-3i lesz
Conjugate(x); // 3.5+3i lesz
double d = ToDouble(x); // 3.5-t ad vissza
```


Adattípusok megvalósítása
Problémák

- Ezzel a módszerrel kapcsolatban több probléma is felmerül:
  1. a függvények paraméterében mindig át kell adnunk a rekordot, és ügyelnünk kell a jó paramétert átadására
  2. egy változó létrehozásakor kezdeti értékek beállítását (vagy az azt elvégző függvény meghívását) nekünk manuálisan kell megtennünk, ha elfelejtjük, akkor az alkalmazás hibásan fog működni
- Célszerű lenne, ha
  1. a függvényeket közvetettebben hozzáilleszteni a típushoz, vagyis jó lenne, ha egy változón keresztül tudnánk meghívni őt, és akkor arra a változóra lenne érvényes
  2. a kezdeti beállítások megtörténnek automatikusan

Adattípusok megvalósítása
Egységbe zárás

- Lehetőségünk van a rekord belsejében is megadni függvényeket a változók mellett, ezáltal közvetlenül a rekord típusú változóhoz fognak kötődni a függvények
  - amikor változót deklarálunk a típusból, a tagfüggvények automatikusan értelmezésre kerülnek rajta
  - a tagfüggvények közvetlenül hozzáférnek a rekord mezőihez, nem kell paraméteren keresztül átadnunk azt a rekordot, amin a műveltet végezzük
- A típus műveletinek beágyazását a típusba (pontosabban a típus rekordjába) nevezzük *egységbe zárás*nak (enkapszulációnak), ekkor a típus mezőit *adatta*goknak, műveleteit *tagfüggvény*eknek nevezzük

Adattípusok megvalósítása
Egységbe zárás

- A tagfüggvényeket csak változón keresztül érhetjük el, ezért mindig kell legyen egy példány a típusból, ekkor

  <változónév>.<függvénynév>(<paraméterek>) formában meghívhatjuk a függvényt
- A tagfüggvények deklarációja hasonlít a sima függvénydeklarációhoz, csak a rekord belsejében végezzük:

```
struct <rekordnév>{
    …
    <típus> <függvénynév 1>(<paraméterek>){
       <függvénytörzs>
    }
    …
};
```


Adattípusok megvalósítása
Egységbe zárás

- Pl. a komplex szám típusa egységbe zárással:

```
struct Complex{ // rekord
   double Re; // valós rész
   double Im; // képzetes rész
   // tagfüggvények:
   void Zero(){ // a szám nullázása
      // nem kell új változót létrehozni
      Re = 0; Im = 0;
      // nem kell változónevet megadni
      // nem kell visszatérési érték
   }
   void FromDoubles(double re, double im){
      Re = re; Im = im;
   }
```


Adattípusok megvalósítása
Egységbe zárás

```
   double ToDouble(){ // valóssá alakítás
      // nem kell paraméter
      return Re;
   }
   void Conjugate(){ // konjugálás
      Im = -Im;
   }
};

// használat:
Complex x; x.Zero(); // 0+0i lesz
x.FromDoubles(3.5, -3); // 3.5-3i lesz
x.Conjugate(); // 3.5+3i lesz
double d = x.ToDouble(); // 3.5-t ad vissza
```


Adattípusok megvalósítása
Deklaráció és definíció szétválasztása

- Továbbra is lehetőségünk van szétválasztani a függvény definícióját a törzsétől
  - ekkor a függvény definícióját a rekordba írjuk, és a rekord után valahol a :: (scope) operátor segítségével megadhatjuk a függvény törzsét

```
struct <rekordnév>{
   … <típus> <fv.név>(<paraméterek>);
};
…
<típus> <rekordnév>::<fv.név>(<paraméterek>)
{
   <függvénytörzs>
}
```


Adattípusok megvalósítása
Deklaráció és definíció szétválasztása

- Pl. a komplex szám típusa szétválasztással:

```
struct Complex{ // rekord
   double Re;
   double Im;
   // tagfüggvény deklarációk:
   void Zero();
   void FromDouble(double nr);
   double ToDouble();
   void Conjugate();
};

// tagfüggvény definíciók:
void Complex::Zero() { Re = 0; Im = 0; }
// ...
```

Adattípusok megvalósítása
A konstruktor művelet

- Létezik olyan tagfüggvény, amely akkor fut le, amikor a típust példányosítjuk változó deklarációval, ez a *konstruktor*
  - ebben lehetőségünk van vele kezdeti érték beállításra
  - olyan előzetes parancsok lefuttatására, amelyeket a további műveletek előtt mindenképpen el kell végeznünk
  - a konstruktornak *nincs típusa* (visszatérési értéke)*, neve megegyezik a rekord nevével, paraméterezése tetszőleges* lehet, egyébként szokványos függvényként viselkedik

```
struct <rekordnév>{
   …
   <rekordnév>(<paraméterek>){<függvénytörzs>}
};
```


Adattípusok megvalósítása
A konstruktor művelet

- A konstruktort nem lehet külön meghívni, akkor hívódik meg, amikor létrehozzuk az adott típusú változót, amennyiben a konstruktornak van paramétere, akkor azt zárójelben kell megadnunk a példányosításkor

```
<rekordnév> <változónév>(<aktuális paraméterek>);
```

- A rekordhoz nyilván nem kötelező konstruktort megadnunk, de ha megadunk, akkor többet is megadhatunk polimorfizmus segítségével
  - különböző konstruktoroknak különböző paraméterlistája kell, hogy legyen, működésük lehet különböző
  - a példányosításkor megadott paraméterek függvényében történik a megfelelő konstruktor meghívása

Adattípusok megvalósítása
A konstruktor művelet

- Pl. A komplex szám két konstruktorral:

```
struct Complex{ // rekord
   double Re; // valós rész
   double Im; // képzetes rész

   // konstruktorok:
   Complex(){ // a 0 szám létrehozása
      Re = 0; Im = 0;
   }
   Complex(double re, double im){
      // a szám létrehozása valósakból
      Re = re; Im = im;
   }
   //... további tagfüggvények
```


Adattípusok megvalósítása
A konstruktor művelet

- Konstruktort zárójelben megadott paraméterek alapján hívhatjuk meg példányosításkor
  - ha nincs konstruktor, akkor az alapértelmezett konstruktor hívódik meg, amely lényegében csak létrehozza a megfelelő adattagokat
  - ha nem adunk zárójelet, akkor a 0 paraméterű konstruktort keresi, ha van konstruktor, de nincs 0 paraméterű, akkor hibát jelez
  - 0 paraméteres konstruktor meghívása:

```
<rekordnév> <változónév>;
<rekordnév> <változónév>();
```

  - ha a megadott paramétereknek megfelelő konstruktort nem találja a fordítóprogram, akkor hibát jelez

Adattípusok megvalósítása
A konstruktor művelet

- Pl. a komplex szám használata:

```
Complex x; // 0+0i lesz
Complex y(); // 0+0i lesz
Complex z(3.5, -3); // 3.5-3i lesz
z.Conjugate(); // 3.5+3i lesz
double d = z.ToDouble(); // 3.5-t ad vissza
// Complex w(3.5); hiba lenne, mert nincs
// megfelelő konstruktor
```


Adattípusok megvalósítása
A destruktor művelet

- Létezik olyan tagfüggvény, amely a változó megsemmisülésekor fut le (amikor a program elhagyja az őt tartalmazó blokkot, vagy kiadunk egy `delete` utasítást), ez a *destruktor*
  - olyan utasításokat tárolunk benne, amelyek „kitakarítják” az általunk használt memóriaterületet
  - *nincs típusa, nincs paramétere, ezért csak egy írható*
  - a destruktor neve a rekord nevével megegyezik kiegészítve a ~ (tilde) karakterrel
  - csak akkor kell implementálnunk, ha speciális memóriaműveleteket alkalmazunk a típusban, mert az egyszerű változókat a program kitakarítja magától

Adattípusok megvalósítása
A destruktor művelet

- Pl.:

```
struct Demo{
   Demo(){ cout << "Hello!" << endl; }
   ~Demo(){ cout << "Byebye!" << endl; }
};

int main(){
   Demo d; // itt fut le a konstruktor
   return 0;
} // itt fut le a destruktor

// eredménye:
// Hello!
// Byebye!
```


Adattípusok megvalósítása
Egységbe zárás

Feladat: Készítsük el az egyetemi hallgató típusát, aki rendelkezik névvel, Neptun azonosítóval, illetve az általa elvégzett kurzusokkal. Legyen lehetősége új kurzus elvégzésére, és lehessen lekérdezni a teljesített kurzusok számát.

Megoldás:

```
struct Hallgato{ // hallgató típusa
   // adattagok:
   string Nev;
   string NeptunKod;
   vector<string> Kurzusok;
   long Bankszamla;
```


Adattípusok megvalósítása
Egységbe zárás

```
   // tagfüggvények:
   Hallgato(string n, string kod, long bsz){
      Nev = n; NeptunKod = kod; Bankszamla = bsz;
      cout << Nev << " hallgató megkezdte
                      tanulmányait." << endl;
   }
   ~Hallgato(){
      cout << Nev << " hallgató befejezte
                      tanulmányait." << endl;
   }

   int KurzusokSzama() { return Kurzusok.size(); }
};
```


Adattípusok megvalósítása
Adatok elrejtése

- Típusok megvalósításánál fontos tényező, hogy azok tagfüggvényei, adattagjai mennyire láthatóak a külvilágból
- Vannak olyan adatok, függvények, amelyeket nem szeretnénk, ha a típuson kívül is látnának, mert
  - a típusainkat mások is használatba vehetik, ha többen dolgoznak egy projekten, és típus megvalósításának módját el akarjuk rejteni a külvilágtól
  - csak segédváltozók, segédfüggvények, amelyek a többi működéséhez kellenek, a típus használatához nem szükségesek
  - külön futtatásuk, értékbeállításuk inkonzisztens állapotba helyezné a példányt, ezért biztosítani kell, hogy ne is legyen lehetőség a megváltoztatásukra

Adattípusok megvalósítása
Láthatóság

- Ezeket az elrejteni kívánt részeket van módunk láthatatlanná tenni a többi programrész (külvilág) számára a *láthatóság szabályozásával*
- Minden tagfüggvény, adattag esetében megadhatjuk, hogy látható-e a külvilágból, vagy sem, és eszerint csoportosíthatjuk őket kulcsszavak bevezetésével
  - amiket el akarunk rejteni `private:` kulcsszó mögé tesszük, ezek csak az osztályon belül lesznek elérhetőek
  - amiket nem akarunk elrejteni `public:` kulcsszó mögé tesszük, ezek mindenhonnan elérhetőek lesznek
  - ezeket beírhatjuk minden sorba, de felesleges, elég egy helyre, és utána írni az összes elemet, amit ide akarunk tenni, tehát mindegyiket elég egyszer leírnunk

Adattípusok megvalósítása
Láthatóság

- A láthatósági kulcsszavak használata nem kötelező, ha nem tesszük meg, `struct` esetében alapértelmezetten minden látható lesz
- Ha rejtett tagot próbálunk elérni kívülről, akkor fordítási hibát kapunk, tehát futás közben a külvilág garantáltan csak látható értékekkel és műveletekkel fog operálni
- A kulcsszavak használata:

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

Adattípusok megvalósítása
Láthatóság

Feladat: Készítsük el az egyetemi hallgató típusát láthatóság kezeléssel. A jelszót tegyük rejtetté.

Megoldás:

```
struct Hallgato{ // hallgató típusa
private: // rejtett
   string NeptunJelszo;

public: // publikus
   string Nev;
   string NeptunKod;
   long Bankszamla;
   vector<string> Kurzusok;
```


Adattípusok megvalósítása
Láthatóság

```
   Hallgato(string n, string kod, long bsz){
      Nev = n; NeptunKod = kod; Bankszamla = bsz;
      NeptunJelszo = /* valamit generálunk */
   }
   int KurzusokSzama() { return Kurzusok.size(); }
};
```

- Kívülről a rejtett tulajdonságok nem elérhetőek:

```
Hallgato h("Eleven Elek","JJJJJJ",1111111);
cout << h.Nev; // elérhető, mert publikus
cout << h.kurzusok.push_back("Bev.Prog. 2");
   // elérhető, mert publikus
cout << h.NeptunJelszo;
   // HIBA, rejtett tag
```


Adattípusok megvalósítása
Láthatóság

- Ez nem elég jó, mert bizonyos tulajdonságoknak csak az olvasását, vagy az írását kívánjuk kívülről megengedni, ezekhez *lekérdező* (*get*), illetve *beállító* (*set*) tagfüggvényeket rendelhetünk.

- Pl.:

```
struct Hallgato{
private: // rejtett az összes adattag
   string nev;
   string neptunKod;
   string neptunJelszo;
   long bankszamla;
   vector<string> kurzusok;
```


Adattípusok megvalósítása
Láthatóság

```
public: // publikus
   Hallgato(string n, string kod, long bsz){
      //...
   }
   int KurzusokSzama() { return kurzusok.size(); }
   // új lekérdező műveletek:
   string Nev() { return nev; }
   string NeptunKod() { return neptunKod; }
   long Bankszamla() { return bankSzamla; }
   vector<string> Kurzusok() { return kurzusok; }
   // új beállító művelet:
   void UjKurzus(string nev) { // kurzus hozzáadás
      kurzusok.push_back(nev);
   }
};
```


Adattípusok megvalósítása
Láthatóság

- Ha a konstruktort elrejtjük, akkor nem lehet példányosítani a típust, azaz nem tudunk belőle változót létrehozni
  - ez néha hasznos, mert előfordul, hogy csakugyan olyan típusra van szükségünk, amelyet konkrétan nem használhatunk (bővebben később)
- Ha írunk destruktort, akkor annak publikusnak kell lennie, különben fordítási idejű hibát kapunk
- Minden további tag bármilyen láthatóságot kaphat, ha valamire nincs külső hivatkozás, akkor azt elrejthetjük, ha van külső hivatkozás, akkor azt előbb meg kell szűntetni
- Általában az adattagokat, és a hozzájuk való hozzáférést megfelelő látható tagfüggvényeken keresztül valósítjuk meg (külön a beállításhoz és a lekérdezéshez)

Adattípusok megvalósítása
Láthatóság

- Az típusok tekintetében láthatóság szempontjából a C++ két fajtát különböztet meg
  - `struct`: alapértelmezetten minden látható
  - `class`: alapértelmezetten minden rejtett
- Ha nem adunk meg láthatóságot valamely elemre, akkor az alapértelmezett láthatóság szerint viselkedik
- Ha egyszer megadjuk a láthatóságot, akkor onnantól a láthatóság az osztályleírás végéig, vagy a következő láthatóság kulcsszóig marad életben
- A konvenció szerint használjuk a `class` kulcsszót típusokra, a `struct` kulcsszót pedig rekordokra

Adattípusok megvalósítása
Láthatóság

- Pl.:

```
struct Demo{           // struct -> public:
   Demo();             // publikus
   ~Demo();            // publikus
   string Fv1();       // publikus
public:
   int Fv2();          // publikus
   strint Ertek1;      // publikus
private:
   int Fv3();          // rejtett
   int Ertek2;         // rejtett
   int Ertek3;         // rejtett
};
```


Adattípusok megvalósítása
Láthatóság

- Pl.:

```
class Demo{            // class -> private:
   Demo();             // rejtett
   ~Demo();            // rejtett
   string Fv1();       // rejtett
public:
   int Fv2();          // publikus
   strint Ertek1;      // publikus
private:
   int Fv3();          // rejtett
   int Ertek2;         // rejtett
   int Ertek3;         // rejtett
};
```


Adattípusok megvalósítása
Láthatóság

Feladat: Készítsük el az egyetemi hallgató típusát úgy, hogy a kurzusnak a kreditértékét is nyilvántartjuk, és le tudjuk kérdezni az összes elvégzett kreditet.

Megoldás:

```
struct Kurzus{ // kurzus típusa
   string Nev;
   int Kredit;
};

class Hallgato{
private:
   vector<Kurzus> kurzusok;
   //...
```


Adattípusok megvalósítása
Láthatóság

Megoldás:

```
public:
   //...
   void UjKurzus(string nev, int kredit){
      Kurzus k; k.Nev = nev; k.Kredit = kredit;
      kurzusok.push_back(k);
   }
   int KreditekSzama{ // összegzés tétele
      int szum = 0;
      for (int i = 0; i < kurzusok.size(); i++)
         szum += kurzusok[i].Kredit;
      return szum;
   }
};
```


Adattípusok megvalósítása
Fordítási egységek

- Ez bizonyos programhossz után a kódfájlunk áttekintetlen lesz, ezért lehetőségünk van a programokat több fájlból létrehozni, és összeszerkeszteni.
- 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 megvalósítása
Fordítási egységek

- C++-ban a fordítási egységek két fájlból állnak:
  - a *fejlécfájl* (*header,* **`.h`**, **`.hpp`**) tartalmazza a típusok felületét, rekordok és függvények deklarációját, illetve tartalmazhatja a függvények megvalósítását is, de ez nem kötelező
  - a *törzsfájl* (*source,* **`.cpp`**) tartalmazza a függvények definícióját, megvalósítását
- A beépített könyvtárak is ilyen fájlokból tevődnek össze, a programjaikban már sokszor hivatkoztunk fejlécfájlokra (pl.: **`iostream`**, **`string`**, **`graphics`**, **`vector`**, …), amelyekhez megfelelő törzsfájlok tartoztak (általában előre lefordítva)

Adattípusok megvalósítása
Fordítási egységek

- A fordítás során elsőként 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`, `.a`, vagy `.lib` kiterjesztésben)
- A szerkesztőprogram összeilleszti a különböző fájlokat, és elkészíti a futtatható állományt (`.exe`), vagy dinamikusan szerkesztett könyvtárat (`.dll`)

Adattípusok megvalósítása
Fordítási egységek

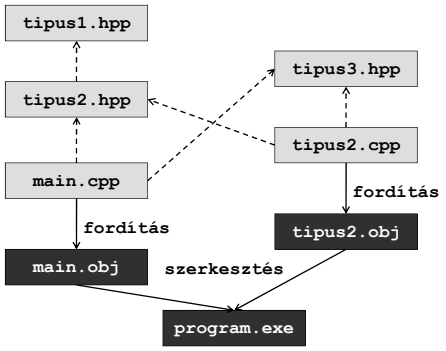
- Saját típusainkat, függvényeinket is elhelyezhetjük külön fájlokban, és ezeket berakhatjuk a programjainkba
  - a típusunk deklarációs részét a fejlécfájlba tesszük
  - a típusunk megvalósítási részét a törzsfájlba tesszük
  - a két fájl nevének ajánlatos megegyeznie
  - a fejlécfájlban, vagy a törzsfájlban meg kell adnunk, ha további fájlbeillesztésre van szükségünk (a fejlécfájlba már beírt hivatkozásokat nem kell megismételnünk)
  - a törzsfájlban meg kell adunk a hozzá tartozó fejlécfájl nevét
- A fejlécfájlban nem adjuk a használt névtereket (pl. `std`, `genv`, …), csak az egyes típusoknál hivatkozunk rájuk

Adattípusok megvalósítása
Fordítási egységek

- 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, illetve nem olyan bonyolult a megvalósítás, hogy érdemes lenne
- A főprogramunk (`main` függvény) fájlja (`main.cpp`) ezentúl nem tartalmaz típusokat, további függvényeket, csak a hivatkozásokat a többi fájlra

Adattípusok megvalósítása
Fordítási egységek

- Pl.:



Adattípusok megvalósítása
Fordítási egységek

- Egy fejlécfájlt többször is beilleszthetünk a programunkba, ekkor ugyanaz a kód többször is bekerülhet, ami ahhoz vezet hogy valami többször is deklarálva lehet, ezt el kell kerülnünk
- Van direktíva, amely 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
  - ha az egész fájlt beletesszük az elágazásba, és megnevezzük, akkor nem kerülhet bele többször a kódba
- A törzsfájlok nem kerülhetnek be többször a kódba, ezért azok esetében nem kell védelemről gondoskodni

Adattípusok megvalósítása
Fordítási egységek

- Tehát a biztonság érdekében minden fejlécfájlunkba be kell írnunk:

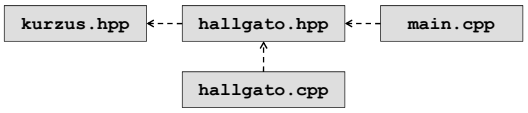
```
#ifndef file_nev
#define file_nev
// fájl tartalma
#endif
```

- A `file_nev` általában megegyezik a fájl nevével, de bármit írhatunk oda (általában csupa nagy betűvel szokás írni)
- Minden tényleges kódot az elágazásba helyezzünk el, beleértve a további `#include` utasításokat (ezeket igazából az elágazás elé is elhelyezhetjük, mert a bennük lévő hasonló elágazások elvégzik a dolgukat)

Adattípusok megvalósítása
Fordítási egységek

Feladat: Készítsük el az egyetemi hallgatót több fordítási egységre tördelve.

- külön fordítási egységbe tesszük a kurzust, amelynek elég pusztán egy fejlécfájlt adni
- a hallgató implementációját szeparáljuk az interfészétől, és külön fejléc- és törzsfájlba helyezzük
- megtesszük a megfelelő hivatkozásokat



Adattípusok megvalósítása
Fordítási egységek

Megoldás:

```
// kurzus.hpp:
#ifndef KURZUS_HPP // többszöri beágyazás védelem
#define KURZUS_HPP

#include <string> // kell a string

struct Kurzus{
   std::string Nev; // megadjuk a névteret
   int Kredit;
};

#endif // kurzus.hpp vége
```


Adattípusok megvalósítása
Fordítási egységek

Megoldás:

```
// hallgato.hpp:
#ifndef HALLGATO_HPP
#define HALLGATO_HPP

#include "kurzus.hpp"
// kell a Kurzus, ezzel bekerül a string is
#include <vector>

class Hallgato{
   //…
public:
   Hallgato(std::string n, std::string kod,
            long bsz);
```


Adattípusok megvalósítása
Fordítási egységek

Megoldás:

```
     //…
};
#endif // hallgato.hpp

// hallgato.cpp:
#include <iostream>
#include "hallgato.hpp" // kell a hallgato fejléc
using namespace std; // használunk névteret

Hallgato::Hallgato(string n,string kod,long bsz){
   nev = n; neptunKod = kod; bankszamla = bsz;
}
```


Adattípusok megvalósítása
Fordítási egységek

Megoldás:

```
//…
int Hallgato::KreditekSzama{
   int szum = 0;
   for (int i = 0; i < kurzusok.size(); i++)
      szum += kurzusok[i].Kredit;
   return szum;
} // hallgato.cpp vége

// main.cpp:
#include <iostream>
#include "hallgato.hpp" // kell a Hallgato
using namespace std;
//…
```


Adattípusok megvalósítása
Fordítási egységek

Megoldás:

```
int main(){
   Hallgato x("Gem Géza","SDGHGD", 111111111);
   h.UjKurzus("Bev. Prog. 1", 4);
   //...
   cout << "Kreditek száma: " << h.KreditekSzama()
        << endl;
   return 0;
}
```

Adattípusok megvalósítása
Fordítási egységek

Feladat: Ne csak a hallgató ismerje a kurzusait, hanem a kurzushoz is vegyük fel a hallgatót. Ennek megfelelően a kurzusnak adhatunk maximális létszámot (ami lehet 0, ha nincs létszáma a kurzusnak).

- a kurzushoz nem magát a hallgatót, csupán annak hivatkozását (mutatóját) kell felvennünk, ezért felveszünk egy mutatókat tároló vektort
- ehhez a kurzushoz is be kellene hivatkoznunk a hallgató osztályát, ám ez kereszthivatkozást okozna, ezért ehelyett a kurzus előtt deklaráljuk a hallgató osztályt
- alakítsuk át a kurzust rekordból osztályra, és kezeljük cím szerint, azaz mutatók segítségével (különben külön kurzus példányokat állítanánk)

Adattípusok megvalósítása
Fordítási egységek

Megoldás:

```
// kurzus.hpp:
//...
class Hallgato; // hallgató típus deklarációja

class Kurzus{ // kurzus típusa
public:
   Kurzus(std::string nev, int kredit,
          unsigned int maxLetszam);

   bool UjHallgato(Hallgato* h);
   // új hallgató felvétele a kurzushoz
```


Adattípusok megvalósítása
Fordítási egységek

Megoldás:

```
   std::string Nev() { return nev; }
   int Kredit() { return kredit; }
   int Letszam(){ return hallgatok.size(); }

private:
   std::string nev;
   int kredit;
   unsigned int letszam;

   std::vector<Hallgato*> hallgatok; // hallgatók
};
```


Adattípusok megvalósítása
Fordítási egységek

Megoldás:

```
// hallgato.hpp:
//...
#include "kurzus.hpp"

class Hallgato{  // hallgató típusa
public:
   //...
   void UjKurzus(Kurzus* k);
private: // rejtett az összes adattag
   //…
   std::vector<Kurzus*> kurzusok;
   // a kurzusokat cím szerint kezeljük
};
```


Adattípusok megvalósítása
Fordítási egységek

Megoldás:

```
//...
Kurzus k("Objektumelvű alkalmazások", 2, 15);
Hallgato h("Gem Géza");
if (k.UjHallgato(&h))
   // ha a kurzus felveszi a hallgatót
   h.UjKurzus(&k);
   // akkor a hallgató is felveheti a kurzust
//…
```


Adattípusok megvalósítása
Alapértelmezett paraméterek

- Ahogy a szokványos függvények esetében is, tagfüggvényeknél is lehetőségünk van *alapértelmezett paraméterek*et megadnunk:

  <típus> <fv. név>(<típus> <név> = <érték>);

  - függvényhíváskor ezeket a paramétereket nem kötelező megadnunk, ekkor az alapértelmezett érték fog behelyettesítődni, tehát a fenti függvényt meghívhatjuk 0, vagy 1 paraméterrel is
  - bármely paraméternek adhatunk alapértelmezett értéket
  - ha egy paraméternek alapértelmezett értéket adnunk, akkor a mögötte lévő összes paraméternek alapértelmezett értéket kell adnunk

Adattípusok megvalósítása
Alapértelmezett paraméterek

- Ha túlterhelést használunk, továbbra is ügyelnünk kell arra, hogy a fordító egyértelműen meg tudja határozni, melyik függvényt kell meghívni, pl.:

```
class Demo{
public:
   Demo();
   Demo(bool value = false); // hiba
   // nem tudná megkülönböztetni az előzőtől
   Demo(bool value1, bool value2 = false);
   Demo(bool value1 = false, bool value2);
   /* hiba, az utolsó paraméternek is kell
      alapértelmezett érték */
   //...
};
```


Adattípusok megvalósítása
Érték inicializálás

- A konstruktorban kapott paramétereket rendszerint a konstruktor törzsében adjuk értékül egy adattagnak
  - ez az értékadás elvégezhető úgynevezett *értékinicializálás* segítségével is, ami még a konstruktor törzse előtt fut le
  - kettőspont után megadjuk az adattag nevét, majd a formális paramétert zárójelben, ha többet akarunk megadni, akkor vesszővel választjuk el:

```
<típusnév>(<típus> <paraméter>) :
   <adattag>(<paraméter>){
   // törzs
}
```

  - amennyiben az osztályban van referencia adattagunk, akkor annak kezdőértéket csak inicializálással adhatunk

Adattípusok megvalósítása
Érték inicializálás

- Pl.:

```
class Kurzus{ // kurzus osztálya
  //...
  Kurzus(std::string nev, int kredit = 0,
         unsigned int maxLetszam = 0)
     : kurzusNev(nev), kurzusKredit(kredit),
       kurzusMaxLetszam(maxLetszam) {}
};
//...
Hallgato::Hallgato(string nev, std::string kod,
                   long szamlaszam)
   : hallgatoNev(nev), neptunKod(kod),
     bankszamla(szamlaszam){ //...
}
```


Adattípusok megvalósítása
Nyílt rekurzió

- Egy típuspéldányban elérhető annak valamennyi tulajdonsága, akár rejtett, akár nem, hiszen az objektum mindig tisztában van saját állapotával
- Lehetőségünk van magát a teljes példányt is elérni magán belül a `this` kulcsszó használatával
  - csak osztályon belül használható
  - ez egy mutatót ad vissza az aktuális objektumra, amely ugyanúgy használható, mint bármely más mutató, amelyet az objektumra állítottunk, azaz lekérdezhető általa az összes objektumtulajdonság:
    `this->`*`<tulajdonságnév>`*
  - ha a konkrét objektumra van szükségünk, akkor a megszokott módon `*this`-t használunk

Adattípusok megvalósítása
Nyílt rekurzió

- Pl.:

```
class Demo{
public:
   Demo(){ demoAttribute = 0; }
   void GiveValueMethod(int value){
      this->demoAttribute = value;
      // ugyanez: demoAttribute = value;
   }
   Demo ReturnMyselfMethod(){
      return *this; // visszaadja az objektumot
   }
private:
   int demoAttribute;
};
```


Adattípusok megvalósítása
Nyílt rekurzió

Feladat: Amikor a hallgató felveszi a kurzust, akkor a kurzusnak is fel kell vennie a hallgatót, sőt először neki kell, mert lehet, hogy fel se tudja venni. Ennek megfelelően módosítjuk a programot.

- a kurzushoz nem magát a hallgatót, csupán annak hivatkozását (mutatóját) kell felvennünk, ezért felveszünk egy mutatókat tároló vektort

Megoldás:

```
void Hallgato::UjKurzus(Kurzus* k){
   if (k->UjHallgato(this)) // aktuális hallgató
      kurzusok.push_back(k);
}
```

Adattípusok megvalósítása
Adatok másolása

- Objektumok statikus, vagy dinamikus létrehozásakor létrejön egy példány belőlük a memóriában, hasonlóan, mint az egyszerű változóinknál, ezt másolhatjuk a programban
- A példányoknak megkülönböztetjük kétféle másolatát:
  - *mély másolat* (*deep copy*): a teljes objektum minden adattagjával lemásolódik a memóriában egy ugyanakkora memóriaterületre
    - értékadásnál és érték szerinti paraméterátadásnál
    - a másolaton végzett értékmódosítások nem lesznek hatással az eredeti példányra
    - nem hatékony, mert lefoglalja újra az objektumnak szükséges memóriaterületet

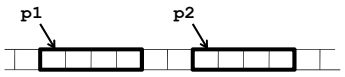
Adattípusok megvalósítása
Adatok másolása

  - *sekély másolat* (*shallow copy*): a hivatkozás (referencia) másolódik csupán le a memóriában, maga a változó nem
    - amikor referencián, vagy mutatón keresztül kezeljük a objektumot, illetve cím szerinti paraméterátadás esetén
    - a másolatra végzett módosítások hatással lesznek az eredeti példányra
    - hatékony, mert csak a memóriacím (4-8 byte) másolódik le, ezért nem foglal felesleges memóriát
- A felesleges memóriahasználat elkerülése érdekében célszerű tehát mindig sekély másolatot készíteni, kivéve, ha tényleg szükségünk van másodpéldányra a módosításokhoz

Adattípusok megvalósítása
Adatok másolása

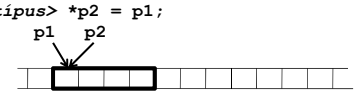
- mély másolat:

```
<típus> p1;
<típus> p2 = p1;
```



- sekély másolat:

```
<típus> *p1 = new <típus>;
<típus> *p2 = p1;
```



Adattípusok megvalósítása
Adatok másolása

Feladat: Egészítsük ki a hallgatóink kezelését bankszámla típussal, amely tartalmazza a tulajdonosát (referenciaként), illetve a forgalmak vektorát, és az alapján megállapítja az egyenleget.

- a kurzushoz nem magát a hallgatót, csupán annak hivatkozását (mutatóját) kell felvennünk, ezért felveszünk egy mutatókat tároló vektort
- a bankszámlát a hallgató konstruktorában hozzuk létre

Adattípusok megvalósítása
Adatok másolása

Megoldás:

```
// bankszamla.hpp:
// ...
class Hallgato; // ide is fel kell venni a
                // hallgató deklarációját

class BankSzamla // bankszámla típus
{
   public:
      BankSzamla(Hallgato& h);
      Hallgato& Tulajdonos() { return tulaj; }
      std::string SzamlaSzam() { return szamla; }
      void Betet(int ertek);
      void Kivet(int ertek);
```


Adattípusok megvalósítása
Adatok másolása

Megoldás:

```
      int Egyenleg();
   private:
      std::string szamla;
      Hallgato& tulaj;
      // a tulajdonos referenciával van megadva
      std::vector<int> forgalmak;
      std::string GeneralSzamlaSzam();
};

// bankszamla.hpp:
#include "bankszamla.hpp"
```

Adattípusok megvalósítása
Adatok másolása

Megoldás:

```
class Hallgato{  // hallgató típusa
   //...
   BankSzamla* bankszamla;
   //...
};

//...
Hallgato::Hallgato(string nev) : hallgatoNev(nev){
   bankszamla = new BankSzamla(*this);
   // bankszámla az aktuális hallgatóhoz
}
```


Adattípusok megvalósítása
Adatok másolása

- Amikor egy objektumot másolunk, akkor alapértelmezetten lemásolódik minden adattagja (mint a rekordoknál)
  - azonban az objektumok szerkezete összetett lehet, tartalmazhat például mutatókat más objektumokra
  - ha egy objektumban mutató található egy másik objektumra, akkor a másolás során a tartalmazott objektumnak csak sekély másolata készül el, ez nem mindig megfelelő
  - felül kell definiálnunk a másolás folyamatát ahhoz, hogy ezt befolyásolni tudjuk, és a további, az objektumhoz tartozó adatokat le tudjuk másolni, a C++ erre biztosít lehetőséget
  - az objektumlétrehozást (konstruktor) szeretnénk túlterhelni

Adattípusok megvalósítása
A másoló konstruktor

- Az objektummásolást a *másoló konstruktor* (*copy constructor*) valósítja meg
  - egy már létező objektum alapján hoz létre újat
  - ez egy olyan konstruktor, amely paraméterben megkapja egy ugyanolyan osztályú objektum referenciáját, és törzsében elvégzi a megfelelő másolási műveleteket
  - ügyelnünk kell, hogy ne módosítsuk az eredeti objektumot, ezért célszerű a paramétert konstansnak megadni
  - ügyelnünk kell, hogy amennyiben saját magába akarjuk másolni az objektumot, akkor ne végezzük el a módosításokat (tehát először mindig ellenőrizzük, hogy az aktuális objektum megegyezik-e a paraméterben kapottal)

Adattípusok megvalósítása
A másoló konstruktor

- A másoló konstruktor használata:

```
class <típus>{
public:
   <típus>() ; // 0 paraméteres konstruktor
   <típus>(const <típus> &other);
   // másoló konstruktor
   …
};

<típus>::<típus>(const <típus>& other){
   <másolási műveletek>
}
```


Adattípusok megvalósítása
A másoló konstruktor

- A másoló konstruktor törzsében tud hivatkozni a másik objektum adattagjaira, beleértve a rejtetteket is
  - általában ezeket egyenként értékül adjuk az új objektumnak
  - ha mutató is van az adattagok között, akkor az általa mutatott változókat is le kell másolnunk
  - ha szükséges, akkor további inicializálásokat végezhetünk a másoló konstruktorban
- A másoló konstruktor a következő esetekben fut le:
  - közvetlen meghívás: `<típus> a; <típus> b(a);`
  - deklaráció értékadással: `<típus> b = a;`
  - érték szerinti paraméterátadás

Adattípusok megvalósítása
A másoló konstruktor

Feladat: Hallgató másolásánál nem másolódik a hozzá tartozó bankszámla, ezért kell egy másoló konstruktort írnunk a hallgatóhoz.

- ehhez felveszünk egy tagfüggvényt a bankszámlában, amely módosítja a tulajdonost
- felhasználjuk a bankszámla másoló konstruktorát

Megoldás:

```
class Hallgato{
   //...
   Hallgato(const Hallgato& masik);
   // másoló konstruktor
};
```

Adattípusok megvalósítása
A másoló konstruktor

Megoldás:

```
Hallgato::Hallgato(const Hallgato& masik){
   // egyszerű tagok átmásolása:
   hallgatoNev = masik.hallgatoNev;
   neptunKod = masik.neptunKod;
   neptunJelszo = masik.neptunJelszo;
   kurzusok = masik.kurzusok;
   // mutatóval hivatkozott adattag:
   bankszamla =
      new BankSzamla(*(masik.bankszamla));
   // a bankszámla másoló konstruktorát futtatjuk
   bankszamla->UjTulajdonos(*this);
      // tulajdonos átállítása
}
```


Adattípusok megvalósítása
Az értékadás

- Hasonlóan másolás történik egy objektumra, amikor értékadást végzünk két változó között az adott típusból

```
<osztálynév> a, b;
<osztálynév> c = a; // ekkor a konstruktor fut le
b = a; // ekkor értékadás történik
```

- Ekkor azonban nem a másoló konstruktor fut le, hanem az értékadás operátor, ezért ilyenkor, amikor nem az alapértelmezett értékadásra van szükségünk, felül kell definiálnunk az értékadás (`=`) operátort
  - csak akkor van rá szükségünk, ha mutatóval hivatkozunk a típus egyik tagjára, és nem csupán a mutatót szeretnénk átmásolni
  - az értékadás csak tagfüggvényként definiálható

Adattípusok megvalósítása
Az értékadás

  - rögzítetten egy paramétere van, amit célszerű konstans referenciaként megadni a paraméter típusa tetszőleges lehet, ezért a tagfüggvény túlterhelhető a paramétertípussal
  - a többszörös értékadás alkalmazhatósága, és a felesleges másolás elkerülése érdekében a művelet visszatérési értéke az aktuális objektum (`*this`) referenciája
- Az értékadásban hasonló tevékenységet végezzük, mint a másoló konstruktorban
  - ellenőriznünk kell, hogy nem-e saját magának adjuk értékül az objektumot
  - ügyelnünk kell a korábban dinamikusan létrehozott adattagok törlésére, hiszen már létező értéket írunk felül

Adattípusok megvalósítása
Az értékadás

- Az értékadás operátor létrehozása:

```
class <típus>{
public:
   //...
   <típus>& operator=(const <típus> &other){
      if (this == &other)
      // ha ugyanoda mutatnak a memóriában
         return *this; // akkor visszaadjuk az
                       // objektumot változatlanul
      <törlési műveletek>
      <másolási műveletek>
      return *this; // visszaadjuk az objektumot
   }
};
```


Adattípusok megvalósítása
Az értékadás

Feladat: Egészítsük ki a hallgatót az értékadással is (ugyanazon oknál fogva, mint a másoló konstruktor esetében).

  - mivel a bankszámlát dinamikusan hoztuk létre, mielőtt létrehoznánk az új bankszámlát, törölnünk kell a korábbit

Megoldás:

```
//..
class Hallgato{
   //...
   Hallgato& operator=(const Hallgato& masik);
   // értékadás operátor
};
```


Adattípusok megvalósítása
Az értékadás

Megoldás:

```
Hallgato& Hallgato::operator=(const Hallgato&
                              masik){
   if (&masik == this)
      return *this;

   delete bankszamla; // bankszámla törlése

   // másolási műveletek:
   hallgatoNev = masik.hallgatoNev;
   neptunKod = masik.neptunKod;
   neptunJelszo = masik.neptunJelszo;
   kurzusok = masik.kurzusok;
```

Adattípusok megvalósítása
Az értékadás

Megoldás:

```
    bankszamla =
       new BankSzamla(*(masik.bankszamla));

    bankszamla->UjTulajdonos(*this);

    return *this; // hivatkozás visszaadása
}
```


Adattípusok megvalósítása
Konstans műveletek

- Amennyiben egy objektumot konstansnak veszünk, akkor csak olyan műveletei futtathatóak, amelyek nem módosítják annak adattagjait, ezek az úgynevezett *konstans műveletek*
  - a kódban jelölnünk kell a konstans műveletet:

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

  - konstans műveletben nem módosíthatóak az adattagok, így valóban futtatható konstans példányra a függvény

Adattípusok megvalósítása
Konstans műveletek

- Pl.:

```
class Demo{
   private:
      int value;
   public:
      int GetValue() const { // konstans művelet
         return value;
      } // nem módosít semmilyen értéket
      void SetValue(int v) { value = v; }
};

const Demo demo; // konstans példány
cout << demo.GetValue(); // megengedett
demo.SetValue(10); // hiba, nem konstans művelet
```


Adattípusok megvalósítása
Konstans műveletek

Feladat: Módosítsuk úgy a hallgatókat kezelő programunkat, hogy konstansnak jelöljük meg azon műveleteket, amelyek valóban alkalmazhatóak konstansokon.

- az összes lekérdező művelet definiálható konstansnak
- ügyeljünk arra, hogy amikor referenciát ad vissza a kurzusok lekérdezése, akkor azt is meg kell jelölnünk konstansnak, különben módosítani lehetne a kurzusokat

Adattípusok megvalósítása
Konstans műveletek

Megoldás:

```
class Kurzus{ // kurzus típusa
   //…
   std::string Nev() const { return kurzusNev; }
   int Kredit() const { return kurzusKredit; }
   int MaxLetszam() const;
   int Letszam() const;
};
```


Adattípusok megvalósítása
Konstans műveletek

Megoldás:

```
class Hallgato{  // hallgató típusa
public:
   //…
   std::string Nev() const { return hallgatoNev; }
   std::string NeptunKod() const;
   BankSzamla* Bankszamla() const;
   const std::vector<Kurzus*>& Kurzusok() const;
   // a visszatérési érték is konstans
};
```