Deklarációk és definíciók. Scope-ok. Hogyan lesz kódból futtatható állomány? Header file-ok. Structok és osztályok. Konstruktorok. Const és mutable módosítók. This.

C++ programozás – 3. óra

Széchenyi István Egyetem
©Csapó Ádám

`http://dropbox.com/...`

2017

Deklarációk és definíciók

- Mielőtt egy nevet / azonosítót használhatunk C++ programban, **deklarálnunk** kell.
  - Ez (nagyjából) azt jelenti, hogy a típusát meg kell adnunk, hogy a fordító tudja, milyen fajta entitásra vonatkozik a név.

```
char ch;
string s;
auto count = 1;
const double pi = 3.1415926535897;
extern int error_number; //(csak ez meg a kovetkezo nem definicio is egyben! ld. kovetkezo folia)
double sqrt(double);

const char* name = "Njal";
const char* season[] = {"spring", "summer", "fall", "winter"};
vector<string> people{nae, "Skarphedin", "Gunnar"};

constexpr int fac(int n){return (n<2) ? 1: n*fac(n-1); }
```

- Látható, hogy deklaráció többet is megtehet, mint hogy névhez típust társítson. Legtöbb fenti példa egyben definíció is!

Deklarációk és definíciók II

- **Definíció**: olyan deklaráció, mely minden információt tartalmaz ahhoz, hogy a programban az adott entitást használhatssuk
  - Konkrétan: ha valaminek a reprezentálásához *memóriára* van szükség, akkor azt a memóriát a definíció le is foglalja
  - Ezzel szemben a deklaráció annyit mond, hogy van egy dolog ami bizonyos módon használható, de nem gondoskodik a memória lefoglalásáról
- Máshogy megfogalmazva:
  - deklaráció: interfész
  - definíció: implementáció
- E szerint külön fájlokban hozzuk létre az interfészt, ide pedig nem tartozik a memóriafoglalás.
- A fenti példában szinte minden memóriafoglalással is jár, kivéve az *error_number* változót és *sqrt* függvényt.

Deklarációk és definíciók III

- Anélkül hogy túlságosan leegyszerűsítő módon fogalmaznánk, azt mondhatjuk hogy egy deklaráció sorrendben az alábbi 5 részből áll:
  - Opcionálisan: prefix minősítők (pl. *static* vagy *virtual*)
  - Egy alaptípus (pl. $vector < double >$ vagy *const int*)
  - Egy deklarátor, ami opcionálisan nevet is tartalmazhat (pl. $p[7]$ vagy n)
  - Opcionálisan: szuffix minősítők (pl. *const* vagy *noexcept*)
  - Opcionálisan: inicializáló vagy függvénytörzs (pl. $= \{7, 5, 3\}$ vagy $\{return\ x;\}$)
- A függvény- és névtér-definíciókat leszámítva mindig kötelező pontosvesszővel lezárni a deklarációt.

Scope-ok

- Minden deklaráció valamilyen scope-ba (hatókör) bevezet egy új nevet. A scope lehet:
  - **Lokális szintű**: a nevet függvényben vagy lambda kifejezésben deklaráltuk. A deklaráció pontjától az adott blokk végéig tart a hatóköre (egy blokkot egy {}-pár zár közre)
  - **Osztály szintű**: egy név "tagnév" vagy "osztály tagnév" ha egy osztályon belül, de annak bármely függvényén, lambdájá, osztályán vagy enum osztályán kívül definiáltuk. Ilyenkor a név érvényes az osztály deklarációjának nyitó kapcsos-zárójelétől egészen annak záró kapcsos-zárójeléig.
  - **Névtér szintű**: névtérben, de függvényen, lambdán, osztályon, enum osztályon kívül definiált név. A név a deklaráció pontjától a névtér végéig érvényes. A névterek gyakran több fordítási egységből is elérhetőek (fordítási egységekről ld. később)

Scope-ok II

- Scope lehet továbbá:
  - **Globális**: bármely függvényen, osztályon, enum osztályon vagy névteren kívül definiált név. Ekkor a deklaráció pontjától az adott fájl végéig él a név. Más fordítási egységekből is el lehet őket érni (ld. később *extern* minősítőt)
  - **Utasítás szintű**: for, while, if vagy switch utasítás ()-zárójelein belül definiált név az utasítás végéig él
  - **Függvény szintű**: goto-val hivatkozott label (soha ne használjuk!!) az adott függvény törzséhez tartozik

Hogyan lesz kódból futtatható állomány?

- A forráskódokat a fordítóprogram az alábbi lépésekben dolgozza fel:

1. Preprocesszálás (makrók, #include-dal történő hivatkozások betöltése) $\rightarrow$ eredmény: translation unit (fordítási egység)
   - egy adott fordítási egység általában egy .cpp fájlból keletkezik (plusz header file-okból amiket include-oltunk, ld. később)
2. Translation unit feldolgozása a C++ nyelv szabályai szerint
   - A külön fordíthatóság érdekében a programozónak minden infót meg kell adnia ahhoz, hogy a fordító izoláltan elemezni tudja a fordítási egységeket (jelesül: minden változót, függvényt stb. legalábbis deklarálni kell)
   - Látni fogjuk, hogy ez header file-ok segítségével könnyebb. Egy header file akár több .cpp fájlhoz is tartozhat $\rightarrow$ nem szükséges mindent újra deklarálni, bizonyos hibák könnyebben elkerülhetőek
3. Linkelés: külön lefordított részek egyesítése egyetlen futtatható állományba (vagy static, dynamic library-be)

Header file-ok

- Header file-ok include-olása így történik:

```
#include<iostream>
#include "sajat_konyvtar.h"
```

- Az *include* direktíva annyit mond a preprocesszornak, hogy másolja be a header file tartalmát a fordítási egység azon pontjára, ahol a direktíva szerepel.
- Általában jó stílus, ha header file-ban csak interfészt adunk meg, nem definiálunk dolgokat. Így kisebbek lesznek a lefordított fájok.
  - Másrészt egy header file-t több forrás is include-olhat $\rightarrow$ ha az implementáció változna, az összes include-oló forrást újra le kellene fordítani!
  - Ha viszont az implementáció egyetlen külön .cpp fájlban lakik, csak azt kell újra lefordítani (illetve linkelni a többivel)

Header file-ok - II.

- Fentiek alapján így használhatunk pl. header file-okat:

```
//s.h:
struct S{int a; char b;};
void f(S*);

//file1.cpp
#include "s.h"
//itt hasznalhatjuk f() fv-t

//file2.cpp
#include "s.h"
void f(S* p)
{
  //itt pedig definialjuk
  //ha az implementacio valtozik, csak file2.cpp-t kell ujra forditani!
  //az s.h fajlt include-olo forrasokat viszont nem kell
}
```

Header file-ok - III.

- A **one-definition rule** (ODR) szerint minden fordítási egységben csak egyszer lehet adott nevű entitást definiálni.
  - Ha egy fordítási egységben kétszer definiálunk valamit, a fordító kajabálni fog
  - Mi történik, ha két fordítási egységben is definiáljuk ugyanazt a dolgot? Ez is baj lehet, de fordító ilyenkor nem tud szólni, hiszen a translation unit-ok külön fordulnak!
    - Nem gond, mert a linker szól! (akkor sem engedi át, ha a definíció ugyanaz)
- Az ODR miatt kell készülnünk arra az esetre, ha mégis van olyan header file amiben definíció is szerepel:
  - Ilyen header file-t közvetetten sem include-olhatunk többször (hiszen az include direktíva a header file-okat szóról szóra bemásolja a fordítási egységekbe)!
  - Pl. mi történik, ha include-olunk két header file-t, és mindkettő include-olja egyazon, harmadik header file-t, amelyben szerepel definíció is?

Header file-ok - IV.

- Ez ellen védenek az ún. **include guard**-ok, amiket szintén a preprocesszor dolgoz fel.
- Header file általában két részből áll:
  - 1. *include guard*: védelem az ellen, hogy adott header file-t egyazon fordítási egység többször include-olja
  - 2. Maga a tartalom
- Például:

```
//ez a ket sor az include guard. ADD_H barmilyen egyedi nev lehet, altalaban a h fajl nevebol jon
#ifndef ADD_H
#define ADD_H

inline int add(int x, int y){
  return x+y;
}; //inline fuggveny definicio

//header guard vege
#endif
```

Header file-ok - V.

- Hány header file tartozzon egy programhoz, hány .cpp fájl include-oljon egyet?
  - Nincs általános szabály, a lényeg hogy jól átlátható legyen a program kódja de ne is bonyolítsuk túl!
- Általában működik az inkrementális építkezés
  - kezdetben egy közös .h fájl az összes deklarációval, melyet a deklarációkat megvalósító .cpp fájlok include-olnak. Később inkrementálisan felbontható!
- Jó ökölszabály: a változókat .cpp fájlban definiáljuk és a többi helyen, ahol használjuk őket, **extern** kulcsszóval deklaráljuk őket
  - extern jelentése: "*Ez csak deklaráció. Hiába tűnik definíciónak, nem kell hozzá memóriaterületet lefoglalni, mert máshol már lefoglaltuk (más fordítási egységben)*"
  - így amikor egyazon header fájlt többször include-oljuk, elkerüljük a többszörös definiálást!

Header file-ok - VI.

```
//file1.cpp
extern int a; //extern jelentese: ‘‘ez csak deklaracio, mem. teruletet nem szabad foglalni’’

//hasznalhatjuk a-t, ertekerol majd a linker gondoskodik!

//file2.cpp
int a = 5;
```

Linkelés

- A programozó felelőssége, hogy minden translation unit-ban deklarálva legyenek azok a névterek, osztályok, fv-ek, stb. amiket használ, és hogy a különböző translation unit-ok ezeket konzisztens módon használják.
- Pl. ez OK:

```
//file1.cpp
int x = 1;
int f() { /* valamit csinal */}

//file2.cpp
extern int x;
int f();
void g(){ x = f(); }
```

Linkelés II

- Itt viszont 3 hiba is van:

```
//file1.cpp
int x = 1;
int b = 1;
extern int c;

//file2.cpp
int x; //azt jelenti h x=0
extern double b;
extern int c;
```

- x kétszer van deklarálva és definiálva
- b-t kétszer deklaráltuk eltérő típussal
- c nincs definiálva
- (fordító ezeket nem látja de a linker hibát dob)

Linkelés III

- Visszatérve, amikor azt írjuk, hogy $\#include < iostream >$, majd felhasználjuk az *std* :: *cout* függvényt, akkor azt tudjuk, hogy:
  - Az iostream header deklarálja az *std* névtérben levő *cout* függvényt
  - A C++ runtime support könyvtár implementálja is, ehhez pedig a fordító automatikusan linkeli a kódunkat.
  - Ha nem létezne iostream header file, akkor az *std* :: *cout* felhasználása előtt az összes szükséges deklarációt meg kellett volna adnunk, ugyanis adott fordítási egységben nem használható fel olyan név, ami nincs deklarálva!

Osztályok C++-ban

- C++ osztályok: beépített típusokhoz hasonló kényelemmel saját típusok definiálása
- Származtatás és template-ek útján ráadásul hierarchikus és parametrikus kapcsolatot is lehetővé tesznek típusok, osztályok között
- Beépített típusok: koncepciók reprezentációi.
  - Pl. *float* típus, operátoraival együtt (+,-,*, stb.) a matematikai valós számok reprezentációja
- Van, hogy ez nem elég: saját típusokat kell definiálni:
  - Pl. játékban *Explosion* típus
  - Pl. szövegszerkesztőben $List < Paragraph >$ típus

Osztályok C++-ban II

- Jól megválasztott osztály-hierarchia előnyei, többek között:
  - Könnyebben lehet érvelni a program helyességéről
  - Általában tömörebb (rövidebb) kód
  - Fordító több hibás felhasználást tud detektálni
- Új típus definiálásakor különválasztjuk egymástól:
  - A megvalósítás körülményeihez tartozó részleteket (pl. adatrejtés: tök mindegy, hogy belül mit hogyan reprezentálok!)
  - A típus használatához szükséges részleteket (pl. fv-ek listája)
- Ebben a C++ nyelv által definiált fájl-típusok is segítenek:
  - Header file-ok (.h, .hh, .hpp): típusok deklarálása
  - Forrás file-ok (.cpp, .cxx, .cc): típusok megvalósítása

Osztályok C++-ban III

- Az osztályok felhasználó által definiált típusok, melyek:
  - Tagváltozókat és tagfüggvényeket tartalmazhatnak
  - Tagfüggvényei definiálhatják a létrehozás / inicializálás, másolás, mozgatás és törlés (destrukció) jelentését
  - Tagjai objektumok esetén . (pont) és mutatók esetén -> (nyíl) operátorokkal érhetőek el
  - Felüldefiniálhatják többek között a +, !, és [] operátorokat
  - Saját névteret alkotnak tagváltozóik és tagfüggvényeik tekintetében
  - Interfészüket public tagokkal, megvalósítási részleteiket pedig private / protected tagokkal valósítják meg
- A **struct** olyan osztály, melynek minden tagja public. Osztálynak viszont alapból minden tagja private.

Osztályok C++-ban IV

- Példa:

```
class X
{
private:
  int m;
public:
  X(int i=0) : m(i){}
  int mf(int i)
  {
    int old = m;
    m = i;
    return old;
  }
}; //pontosvesszo mindig kell, mert nem fv vagy nevter definicio

//inicializalas es felhasznalas
X var(7);

int user(X var, X* ptr)
{
  int x = var.mf(7);
  int y = ptr->mf(9);
  int z = var.m; //hiba: privat adattag nem hozzaferheto
}
```

Osztályok C++-ban V

- Struct esetén minden publikus:

```
struct Date
{
  int d,m,y;
  void init(int dd, int mm, int yy); //inicializalashoz
  void add_year(int n);
  void add_month(int n);
  void ad_day(int n);
};

void Date::init(int dd, int mm, int yy)
{
  d = dd;
  m = mm;
  y = yy;
}

Date mydate;
mydate.init(5, 12, 1977);
mydate.d = 6; //OK mert struct ezert minden publikus
```

Osztályok C++-ban VI

- **Inicializáláskor alapesetben minden objektum másolódik**, az összes tagváltozójával és metódusával együtt:

```
Date my_birthday;
my_birthday.init(30, 12, 1950);

Date d1 = my_birthday; //inicializalas masolassal
Date d2 {my_birthday}; //inicializalas masolassal
```

- Ha nem ilyen működést szeretnénk (pl. ha mozgatni szeretnénk, mert nagyon költséges a másolás), vannak más lehetőségek (ld. később)
  - Biztos azonban, hogy ilyenkor magunknak kell gondoskodnunk az inicializálásról, mert *az alapértelmezett inicializálás másolással történik*

Osztályok C++-ban VII

- A Date típus nem ideális
  - nem mondtuk meg, hogy csak a tagfüggvények függhetnek a Date reprezentációjától
  - nem kötöttük ki, hogy csak a tagfüggvények férhetnek hozzá a tagváltozókhoz!
- Ezek eléréséhez már osztályra van szükségünk.

```
class Date
{
  int d,m,y; //alapesetben minden privat, ezert megvalosul az adatrejtes!
public:
  void init(int dd, int mm, int yy); //inicializalashoz
  void add_year(int n);
  void add_month(int n);
  void ad_day(int n);
};

void Date::add_year(int n)
{
  y += n;
}
```

Osztályok C++-ban VIII

- Ilyenkor már alapesetben minden privát lesz:

```
void timewarp(Date& d)
{
  d.y -= 200; //error: Date::y privat!
}

//Az init fv-re ezert mindennel nagyobb szuksegunk van
Date dx;
dx.m = 3; //hiba!
dx.init(25, 3, 2011); //OK
```

- Az adatrejtésnek sok előnye van. Pl. ha egy dátum hülyeség (mondjuk Dec. 36, -20), erről csak az osztály megvalósítása tehet.
  - A debuggolás első lépése (lokalizáció) már futás előtt megtörténik!
- Megjegyzés: címek manipulálásával persze privát tagváltozókat is felülírhatunk, de ez csalás lenne
  - a nyelvnek nem a rosszindulatú használat ellen kell védenie

Konstruktorok

- Az olyan fv-ek, mint amilyen a korábbi *init()* volt sok hibalehetőséget rejtenek.
- Például: Sehol sincs kikötve, hogy meg kell hívni, ezért a programozó el is felejtheti!
- Sokkal jobb, ha konstruktort használunk!
- A **konstruktor** olyan függvény, mely az osztály tagfüggvénye, melynek neve megegyezik az osztály nevével, és melynek nincs visszatérési értéke

```
class Date
{
  int d,m,y;
public:
  Date(int dd, int mm, int yy); //konstruktor
};
```

Konstruktorok II

- Ezután már kötelező konstruktort használni!

```
Date today = Date(23, 6, 1983);
Date xmas(25, 12, 1990); //tomor valtozat, ez is OK
Date xmas2{25, 23, 2014}; //ugyanezek mennek kapcsos zarojellel is

Date my_birthday; //hiba: inicializalas hianyzik!
Date masikHiba(10, 12); //hiba: harmadik argumentum hianyzik
```

- Az inicializálásnak több szintaxisa lehet, de érdemes kapcsos zárójeleket használni. Ennek több előnye van:
  - Nincs implicit szűkítés ("narrowing"). Pl. char-bol int OK, de forditva nem. Double-bol sem enged float-ot csinálni
  - Ritka esetekben nem használhatjuk csak, amikor a kapcsos zarojeleknek más jelentése van. Pl. vector osztály esetében:

```
vector<int> v1(5); //5-hosszu vektort foglalunk le
vector<int> v2{5}; //vektor, melynek elso eleme 5
```

Konstruktorok III

- Osztályhoz akárhány konstruktor definiálható, csak az a lényeg hogy különböző számú és/vagy típusú argumentumuk legyen:

```
class Date{
  int d, m, y;
public:
  //...
  Date(int, int, int);
  Date(int, int);
  Date(int);
  Date(); //default konstruktor: peldaul lehet a mai nap
  Date(const char*); //string reprezentacioban megadott datum
};

Date today{4}; //peldaul honap es ev a mai nap alapjan
Date july4{"July 4, 1995"};
Date guy{5,11}; //ev a mai nap alapjan

//ez a ketto csak akkor OK, ha van default konstruktor (mint ahogy most van)
Date now; //default, peldaul lehet a mai nap...
Date start{}; //ugyanugy default
```

Konstruktorok IV

- Elegánsabb lesz a kód, ha több konstruktor definiálása helyett default argumentumokat használunk:

```
class Date
{
  int d,m,y;
public:
  Date(int dd=0, int mm=0, int yy=0); //egyszeru es nagyszeru
  Date(int dd=0, char*); //hiba: ld utolso megjegyzes
  //...
};

Date::Date(int dd, int mm, int yy)
{
  d == dd ? dd : today.d;
  m == mm ? mm : today.m;
  y == yy ? yy : today.y;
}
```

- Default argumentumot sose követhet nem-default (hogyan lenne értelmezhető, amikor kevesebb paraméterrel hívják meg a fv-t?)

Konstruktorok V

- Másik lehetőség a redundancia ellen: tagváltozók inicializálása

```
class Date
{
  int d;
  int m {today.m};
  int y {today.y};
public:
  Date(int);
  Date();
  Date(const char*);
  //...
};

//eleg csak d-t inicializalni, hiszen minden mast inicializaltunk
Date::Date(int dd) : d{dd}
{
  //ellenorizzuk, hogy a datum ervenyes-e
}
```

Automatikus konverziók, explicit kulcsszó

- Ha egy fv valamilyen objektumot vár, néha implicit módon is példányosítani tudjuk az argumentumot:

```
complex<double> d{1}; // d == (1,0)
void my_fct(Date d);

void f()
{
  Date d {15}; //d a mai nappal egyezo honap es ev 15. napja
  //...
  my_fct(15); //obskurus, de mukodhet!
  d = 15; //ez is obskurus, de mukodhet! (=-operator is egy fv...)
}
```

- Érthető módon nem mindig örülünk hogy ez így működik, nehezen olvasható kódhoz vezet!

Automatikus konverziók, explicit kulcsszó II

- Mindez megakadályozható, ha használjuk az *explicit* kulcsszót
- Az **explicit** kulcsszót konstruktorok elé szokás tenni. Ilyenkor azt eredményezi, hogy *valahányszor ilyen típusú objektumot kell egy függvénynek átadni, nem használhatjuk rövidítésképpen valamely konstruktorának argumentumait.*

```
class Date
{
  int d, m, y;
public:
  explicit Date(int dd=0, int mm=0, int yy=0);
  //...
};

Date d1 {15}; //OK, ez explicit
Date d2 = Date{15}; //OK, ez is explicit
Date d3 = {15}; //hiba! nem explicit
Date d4 = 15; //ez is ugyanugy hiba
```

- A saját osztályunkban kiköthetjük, hogy más programozók / program

Konstans tagfüggvények (const és mutable)

- Const kulcsszóról már volt szó: "*megígérem, hogy a változó értékét nem módosítom*". De ugyanez értelmezhető függvényekre is.
- Const metódus garantáltan nem módosítja az osztály tagváltozóit, állapotát (és ezt a fordító ki is kényszeríti)

```
class Date
{
//...
public:
  int day() const { return d; }
  int month() const {return m; }
  int year() const;
  //...
  void add_year(int n);
};

int Date::year() const
{
  return ++y; //ez hiba, a fordito sem engedi!
}
```

Konstans tagfüggvények (const és mutable) II

- Const objektumra nem hívható meg nem konstans metódus (érthető okokból)

```
void f(Date& d, const Date& cd)
{
  int i = d.year(); //OK
  d.add_year(1); //OK

  int j = cd.year(); //OK
  cd.add_year(1); //hiba: konstans valtozo erteke nem modosithato
  //(meg akkor sem, ha a fv amugy nem tenne ilyet!)
  //a fordito nem olyan okos, csak azt nezi hogy const tagfuggveny-e
}
```

- Beszélhetünk viszont olyanról is, hogy egy metódus *logikailag* const. A felhasználó számára úgy kell, hogy tűnjön, mintha const lenne... de belül mégiscsak muszáj az objektum állapotát módosítanunk.
  - Pl. ha egy objektum konstans ugyan, de valamilyen értéket a hatékony működéshez cache-elnie, és időnként frissítenie kell.

Konstans tagfüggvények (const és mutable) III

- Tfh a Date osztálynak van egy string reprezentációja, amit költséges mindig újra és újra kiszámítani. Ekkor:

```
class Date
{
public:
  //...
  string string_rep() const; //a reprezentacio lekerdezheto
private:
  bool cache_valid; //ervenyes-e a reprezentacio mostani erteke
  string cache; //maga a reprezentacio
  void compute_cache_value(); //frissitsuk a cache erteket
  //...
};
```

- *string_rep*() a felhasználó szempontjából nem módosítja az objektum állapotát, de cache értéke időnként változik. A probléma, hogy const objektum esetén elvileg a *compute_cache_value*() fv-t nem lehetne meghívni!

Konstans tagfüggvények (const és mutable) III

- Egyik megoldás: használjunk *const_cast* típusú kasztolást
- Elegánsabb megoldás: **mutable** kulcsszó használata. Jelentése: *ez a tagváltozó const objektum esetén is módosítható*!

```
class Date
{
public:
  //...
  string string_rep() const; //a reprezentacio lekerdezheto
private:
  mutable bool cache_valid; //ervenyes-e a reprezentacio mostani erteke
  mutable string cache; //maga a reprezentacio
  void compute_cache_value() const; //frissitsuk a cache erteket
  //...
};
```

Konstans tagfüggvények (const és mutable) IV

- Ezek után:

```
string Date::string_rep() const
{
  if(!cache_valid){
    compute_cache_value();
    cache_valid = true;
  }
  return cache;
}

void f(Date d, const Date cd)
{
  string s1 = d.string_rep();
  string s2 = cd.string_rep(); //OK!!
  //...
}
```

Konstans tagfüggvények (const és mutable) V

- Harmadik megoldás: ha egy objektumnak nemcsak egy kis részét kell módosíthatóvá tennünk, használhatjuk külön objektum indirekt elérését (a const ezekre tranzitívan ugyanis nem vonatkozik):

```
struct cache{
  bool valid;
  string rep;
};

class Date{
public:
  //...
  string string_rep() const;
private:
  cache* c;
  void compute_cache_value() const;
  //...
};
```

```
string Date::string_rep() const
{
  if(!c->valid){
    compute_cache_value();
    c->valid = true;
  }
  return c->rep;
}
```

Tagváltozók elérése

- "Rendes" tagváltozók és pointerek közötti különbségek (emlékezzünk vissza referencia bevezetésének okaira):

```
struct X{
  void f();
  int m;
};

void user(X x, X* px){
  m = 1; //error: nincs m a scope-ban
  x.m = 1; //OK
  x->m = 1; //error: x nem pointer
  px->m = 1; //OK
  px.m = 1; //error: px pointer
}
```

- A fordító elvileg . használata esetén is tudna külöbséget tenni pointer és nempointer között, de olvasni borzasztó lenne!

Tagváltozók elérése II – statikus tagváltozók

- Globális változók, mint Date osztályban a *today* használata veszélyes.
- Erre valók a statikus tagváltozók, amelyek az osztályhoz tartoznak:

```
class Date{
  int d,m,y;
  static Date default_date;
public:
  Date(int dd=0, int mm=0, int yy=0);
  //...
  //fv is lehet statikus, ha sosem objektumra, hanem az osztaly kontextusaban hivjuk meg
  static void set_default(int dd, int mm, int yy); //default beallitasahoz
};

Date::Date(int dd, int mm, int yy)
{
  d == dd ? dd : default_date.d;
  m == mm ? mm : default_date.m;
  y == yy ? yy : default_date.y;
}
```

- Figyelem: többszálú programnál versenyhelyzetet okozhatnak!

Önreferencia: this

- Az objektum állapotát megváltozató fv-ek esetén (mint pl. *add_year*(), *add_day*(), stb.) célszerű az objektumra mutató referenciát visszaadni.
  - Ilyen esetben ugyanis az alábbi szintaxissal összefűzhetnénk több hívást:

```
void f(Date &d)
{
  //...
  d.add_day(1).add_month(1).add_year(1);
  //...
}
```

- Ennek érdekében az alábbi módon deklaráljuk ezeket a fv-eket:

```
Date& add_year(int n);
Date& add_month(int n);
Date& add_day(int n);
```

Önreferencia: this II

- De hogyan valósítsuk meg őket?? Ilyenkor jó a **this** pointer:

```
Date& Date::add_year(int n)
{
  if(d==29 && m==2 && !leapyear(y+n))
  {
    d = 1;
    m = 3;
  }
  y += n;
  return *this;
}
```

Önreferencia: this III

- Minden nem-statikus fv tudja, milyen objektumra hívták meg!
- X típusú osztályban *this* típusa $X*$
- Const metódus esetén a típusa viszont *const X** lesz.
  - Nem $X*$ const, vagyis nem a pointer const, hanem a mutatott érték! (mindig jobbról kell olvasni, ld. korábban)
  - Így garantálható, hogy this-en keresztül a metódus nem módosíthatja az objektum állapotát
- Fontos, hogy this rvalue-nak számít, ezért nem lehet pl. a címét lekérdezni, és érteket sem adhatunk neki.

Önreferencia: this IV

- *this* használata legtöbbször implicit. Pl. az *add_year*() fv így is megvalósítható lett volna, de felesleges:

```
Date& Date::add_year(int n)
{
  if(this->d==29 && this->m==2 && !leapyear(this->y+n))
  {
    this->d = 1;
    this->m = 3;
  }
  this->y += n;
  return *this;
}
```

- Időnként azonban explicit módon használjuk fel. Erre példa a fenti esetben a visszaadott referencia, illetve a következő fólián levő, láncolt listás struktúra megvalósítása.

Önreferencia: this V

```
struct Link
{
  Link* pre;
  Link* suc;
  int data;

  Link* insert(int x)
  {
    return pre = new Link{pre, this, x};
  }

  //toroljuk this-t
  void remove()
  {
    if(pre) pre->suc = suc;
    if(suc) suc->pre = pre;
    delete this;
  }
};
```

Hasznos tanácsok Stroustruptól

- Koncepciókat osztályok segítségével írjunk le
- Az osztály interfészét különítsük el megvalósításától
- Csak akkor használjunk publikus adatokat (struct-okat), ha tényleg csak adatokról van szó és nincsenek invariancia-feltételek
- Használjunk konstruktort init helyett
- Egyetlen argumentummal rendelkező konstruktorokhoz általában használjuk az explicit kulcsszót!
- Ha tagfüggvény adatot nem módosít, használjuk a const kulcsszót!

Hasznos tanácsok Stroustruptól II

- A “konkrét típus” a legegyszerűbb osztály-típus. Ha tehetjük, ezeket preferáljuk a bonyolultabb osztályok illetve sima adatstruktúrák helyett!
- Függvény csak akkor legyen tagfv, ha tényleg szüksége van arra, hogy ismerje az objektum belső reprezentációját
- Használjunk névteret az osztályok és segédfv-ek közötti kapcsolat megvilágítására!
- Ha egy fv-nek ismernie kell az osztály belső reprezentációját, de nem szükséges specifikusan objektumokra meghívni, használjuk a static kulcsszót!