Bevezetés. Típusok és memóriakezelés alapjai.

C++ programozás – 1. óra
Széchenyi István Egyetem

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

2017

Bevezetés

- "*C++ is a general-purpose programming language providing a direct and efficient model of hardware combined with facilities for defining lightweight abstractions*" (B. Stroustrup, először 1982-ben)
- Kezdetben (többé-kevésbé) C nyelvet bővítő halmaz, melynek célja elsősorban rendszerprogramozás
- Rendszerprogramozás: direkt felhasználása a HW-nek, erőforrás-érzékenység, vagy interakció erőforrás-érzékeny kóddal:
  - Driverek, kommunikációs stack-ek, virtuális gépek, operációs rendszerek,...
- General-purpose: microcontrollertől hatalmas, elosztott üzleti alkalmazásokig

Bevezetés II.

- Két fontos elv:
  - C++-nál alacsonyabb absztrakciós szintű programozási nyelvnek ne nagyon legyen tere (kivéve persze ha a HW-t direktben programozzuk)
  - "*What you don't use you don't pay for*": semelyik nyelvi funkciót se lehessen szimulálni akárcsak egy kicsit jobb teljesítménnyel se:
    - $\rightarrow$ egyre egyszerűsödő, elegánsabb, hatékonyabb eszközök (pl. STL, boost)
- Ezzel együtt nagy szabadság, sokféle programozói stílus
  - Nehézség: a programozóra van bízva, hogy hatékony kombinációban alkalmazza ezeket
  - Adott feladat sokféleképpen megoldható! Könnyű rossz kódot írni!
  - Megoldás:
    - gyakorlás, gyakorlás, gyakorlás
    - elegáns kód tanulmányozása

Bevezetés – III.

- A C++ alapegysége az *osztály*, amely az alábbi tulajdonságokat hordozza:
  - Adatrejtés (data hiding)
  - Garantált inicializálás
  - Implicit típus-konverzió, dinamikus tipizálás
  - Felhasználó által kontrollálható memória menedzsment
  - Operátorok felüldefiniálása (operator overloading)
- Típusellenőrzés és modularitás megfogalmazása rugalmasabb C++-ban, mint C-ben
- De: típusellenőrzés szigorúbb is, mint C-ben. Pl.:

```
void *ptr;
int* i = ptr; //C-ben OK, C++-ban kasztolni kell pl int* i = (int *)ptr;
```

Bevezetés – IV.

- További javítások:
  - szimbolikus konstansok (#define, de kerüljük, mert globális névteret használ)
  - inline függvények
  - alapértelmezett fv.paraméterek
  - overload-olt (felültöltött) fv.nevek
  - free store management operátorok (new, delete malloc és free helyett)
  - referencia-típus
  - ...
- Közben megmarad a C azon képessége, hogy HW-szintű egységeket kezeljünk
- C++-ból felhasználhatunk C-könyvtárakat is
- Erős támogatottság a legtöbb platformon

Bevezetés V: Programozási stílusok

- Stroustrup papa jótanácsa: a nyelv funkcióira ne tekintsünk kész megoldásként! $\rightarrow$ inkább mint téglákra gondoljunk rájuk
  - Egyszerű megfogalmazása az ideálisnak:
  - Koncepciók, fogalmak direkt megfogalmazása kódban
  - Független koncepciók, fogalmak független megfogalmazása
  - Fogalmak közötti kapcsolatok direkt megfogalmazása
  - DE: csak akkor ha ennek van értelme
  - Egyszerűség, elegancia
- További célok:
  - Statikusan típus-ellenőrzött megoldások preferálása
  - Információ legyen lokális: pl. kerüljük a globális változókat
  - Ne absztraháljunk túl a kelleténél: osztályhierarchiák ne legyenek túl kidolgozottak
- Minden nyelv erre törekszik, de más módokon:
  - Más felhasználási környezet, más háttér, más kompromisszumok hatékonyság és elegancia között, ...

Bevezetés VI.: Programozási stílusok II.

- C++ háttere: rendszerszintű programozás (C, BCPL) és absztrahálás lehetősége (Simula)
- 4 fajta programozási stílus támogatása:
  - **Procedurális programozás**: processzálás, adatstruktúrák (mint C, Algol)
  - **Adat-absztrakció**: interfészek biztosítása (pl. absztrakt és valódi osztályok), saját adattípusok definiálása
  - **Objektum-orientált programozás**: osztály-hierarchia, polimorfizmus
  - **Generikus programozás**: template-ek (fordításidejű parametrikus polimorfizmus)
- Hangsúly: stílusok hatékony kombinálása. Stroustrup: szörnyű, amikor a C++-t "objektum-orientált" nyelvként jellemzik (ez az igazságnak csak egy szelete, a valóság összetettebb)

Bevezetés VII.: Programozási stílusok III.

- Ezek a stílusok már korán megvoltak:
  - Osztályok: mind a 4 stílus támogatása
  - Publikus/privát osztályváltozók és metódusok: interfész és implementáció megkülönböztetése
  - Tagfüggvények, konstruktorok, destruktorok, user-defined inicializálás (assignment): adat absztrahálásához, obj.orientált prog.hoz, és egységes nyelv generikus programozáshoz
  - Függvény-deklaráció: statikusan ellenőrzött interfészek
  - Generikus fv.-ek, parametrizált típusok (később template-ek)
  - ...
- Filizófia: régen rossz lenne, ha a nyelv funkcióinak korlátozásával akarnánk helyes kódhoz jutni.

Bevezetés – VIII.: Ki használja?

- Sok millió programozó, többmillió kódsornyi C++ van használatban
  - Cégek: Google, Facebook, Amazon, Bloomberg, Morgan Stanley, Amadeus, ...
  - Operációs rendszerek: oktatáshoz, RT, high-throughput I/O
  - Legelterjedtebb operációs rendszerek számos alrendszere, driverek
  - Telefonok, kommunikációs eszközök, beágyazott rendszerek
  - Programnyelvek implementációja: JVM, Javascript (Google V8), böngészők, alkalmazási keretrendszerek (.NET)
  - Robotika, autók, szenzorhálózatok
  - Numerikus számítások (NASA, CERN, ...)
  - Grafika, virtuális valóság
  - Nem-szabványos könyvtárak: Boost, OpenCV, keményebbeknek: Loki

Fejlesztési környezet beüzemelése

```
//hello.cpp

#include<iostream>
using namespace std; //nevterek hasznalata altalaban kotelezo
//egyretegu nevter eseten ne hasznaljunk using namespace-t
//tobbregetu eseten sem, hanem inkabb namespace ezc = ezt::middleware::common;
//mi tortenne pl ha ket using namespace van, es mindkettoben szerepel egy azonos fv?
//meg alattomosabb, ha kezdetben minden OK, aztan valaki letrehoz egy masik nevterben egy uj fv-t.
      Nagyon alattomos!

int main()
{
  cout << "Hello world!" << endl; //inkabb: std::cout es std::endl!
  char c;
  cin >> c; //inkabb std::cin!
  return 0;
}
```

Típusok fajtái és belső reprezentálásuk

- Típus: adatok és rajtok elvégezhető műveletek (operátorok)
- Több beépített típus, és vannak átfedések is (többféle egész, többféle előjel nélküli típus, többféle lebegőpontos szám)
- Azért, mert a nyelv nem akar túl sokat feltételezni a HW-ről!
  - Bool típus: bool (int-re konvertálva 0 vagy 1 értékű, fordítva 0-ból false, minden másból true)
  - Karakter típusok: char, wchar_t,char16_t, char32_t
  - Egész típusok: int, short, long, long long
  - Lebegőpontos típusok: double, long double
  - Információ hiánya: void
  - Ezekből deklarátor operátorokkal más típusokat is létrehozhatunk (mutató, tömb, referencia-típusok, struct, enum, class)

Típusok fajtái és belső reprezentálásuk II.

- Adott architektúrán adott típus byte-számát a sizeof(x) fv.-nyel kérdezhetjük le (**fontos**: *itt a byte nem 8 bit, hanem az arch. legkisebb, címezhető egysége*).
- Ami biztos:
  - C++ objektumok mérete a char típus méretének egész számú többszöröse!
  - Vagyis: $sizeof(char) == 1$
  - Az is biztos, hogy egy char legalább 8 biten tárolódik, továbbá:
    - $1 == sizeof(char) <= sizeof(short) <= sizeof(int)$
    - $sizeof(int) <= sizeof(long) <= sizeof(long\ long)$
    - $1 <= sizeof(bool) <= sizeof(long)$
    - $sizeof(char) <= sizeof(wchar\_t) <= sizeof(long)$
    - $sizeof(float) <= sizeof(double) <= sizeof(longdouble)$
    - $sizeof(N) == sizeof(signedN) == sizeof(unsignedN)$// ahol N char, short, int, long vagy long long

Típusok fajtái és belső reprezentálásuk III.

- Típusokhoz általában operátorok tartoznak.
- Példa: aritmetikai operátorok egész típusra:

```
int a = 7; //int tipusu valtozo, melynek neve a
//a valtozot egesz tipusu 7-es ertekre inicializaljuk

a = 9; //ertekadas (nem inicializalas): a erteket 9-re modositjuk

a = a+a; //megduplazzuk a erteket
a += 2; //...majd inkrementaljuk 2-vel
++a; //... majd inkrementaljuk (1-gyel)... vegyuk eszre: ++a es nem a++
//az utobbi is jo de tobb a tevedes lehetosege
```

Típusok fajtái és belső reprezentálásuk IV.

- A memóriában minden bitekből áll, a típus ad a biteknek értelmezést
  - a **01100001** bitsorozat 97-es intnek és ‘a’ karakternek felel meg
  - a **01000001** bitsorozat 65-ös intnek és ‘A’ karakternek felel meg
  - a **00110000** bitsorozat 48-as intnek és ‘0’ karakternek felel meg

```
char c = 'a';
std::cout << c; //a c nevu karakter erteket irjuk ki, ami 'a'
int i = c;
std::cout << i; //az i nevu int valtozo erteket irjuk ki, ami 97
```

- Ugyanúgy mint a valóságban: ok hogy “42”, de 42 mi?

Típusok fajtái és belső reprezentálásuk V.

- Egy kicsit összetettebb példa

```
//col es cm kozotti konverzio
int main()
{
  const double cm_per_inch = 2.54;
  int val;
  char unit;
  while(std::cin >> val >> unit)
  {//olvassunk be amig a felhasznalo szeretne
    if(unit=='i'){
      std::cout << val << "in == " << val*cm_per_inch << "cm" << std::endl;
    }
    else if(unit=='c'){
      std::cout << val << "cm == " << val/cm_per_inch << "in" << std::endl;
    }
    else{
      return 0; //"hibas" mertekegyseg (pl 'q') eseten kilepes
    }
  }
}
```

Típusbiztonság

- Típusbiztonság: mi az? (angolul type safety)
- Minden adatot csak a típusának megfelelően használhatunk
  - Változót csak inicializálást követően használunk
  - Csak a változó deklarált típusának megfelelő műveleteket végzünk rajta
  - A változóra definiált minden művelet érvényes értéket ad a változónak
- Ideál: statikus típusosság
  - Fordító minden esetben jelzi a turpisságot (C++ esetében nem mindig garantált – nem is létezik olyan általánosan használt nyelv, ami mindig garantálja)
- Ideál: dinamikus típusosság
  - Futáskor a környezet érzékeli azt a tényt, hogy a típusosság követelményeit felrúgtuk (C++ nem mindig garantálja, de van olyan nyelv amelyik igen)

Példa a típusosság felrúgására

- Implicit szűkítés: értékadáskor az új típus nem tudja “teljes felbontásban” az eredeti értéket tárolni:

```
int main()
{
  int a = 20000;
  char c = a;
  int b = c;

  if(a!=b){
    std::cout << "oops! " << a << "!=" << b << std::endl;
  }
  else{
    std::cout << "wow! we have large characters!" << std::endl;
  }
}
```

Példa a típusosság felrúgására II.

- Inicializálatlan változók: a fordító tipikusan szól, de a C++ szabvány nem köti ki.

```
int main()
{
  int x; //x "random" kezdoertekre inicializalodik
  char c; //ugyanugy c
  double d; //ugyanugy d, viszont: nem minden bitminta ervenyes lebegopontos szam!

  double dd=d; //ezert itt pl. baj lehet (nem masolhatunk invalid leb.pontos szamokat)
}
```

Függvényekről röviden

- Függvényre már láttunk egy-két példát. A modularizáció egyik alapvető egységét jelentik
- Fontos, hogy C++-ban a paraméterek **érték szerint kerülnek átadásra**, nem identitás (cím) szerint. (Később látni fogjuk, hogy "kivétel" a pointer, mert az önmagában is címet jelent, ugyanakkor a cím maga akkor is értékként kerül átadásra)

```
int f(int a, int b)
{
  a = b+1;
  b = a+2;
  return b;
}

int a = 2;
int b = 3;
int c = f(a,b); //a es b erteke nem valtozik, c erteke pedig 6
//pointer eseten is csak a mutatott erteket valtoztathatjuk fv-en belul
//magat a cimet nem mert az egy ertekkent kerul atadasra (masolatkent)...
//ha meg is valtoztatjuk, az eredeti pointerre nem lesz kihatassal!
```

Memóriakezelés

- Program memóriaterülete az alábbiakból áll:
  - *parancssori argumentumok* (argc, argv[])
  - *stack* (szabad tár felső része, lefele növekszik)
  - *heap – avagy halom* (szabad tár alsó része, felfele növekszik)
  - *inicializált adatszegmens*
  - *0-ra inicializált (inicializalatlan) adatszegmens (BSS)*
  - *kódszegmens* (programkód)
- Kicsit részletesebben:
  - **Kódszegmens**: futtatható kód (hozzáférhető, de nem módosítható dinamikusan)
  - **Adatszegmens**: két része van (inicializált és nem inicializált). Globális és statikus változókat tárol.
  - **Stack**: függvények adatainak tárolása (pl. paraméterek, lokális de nem statikus változók, stb.)
  - **Heap**: dinamikusan allokált adatok tárolása (ld. new és delete operátorok)

Memóriakezelés – II.

```
int initToZero1;
static float initToZero2;
int* initToZero3; //mindharom inicializatlan adatszegmens mert inicializalatlan globalis vagy
      statikus valtozok
double initializedD = 20.0; //inicializalt adatszegmens

int main()
{
  int stringLength; //stack, mert lokalis valtozo. A fv vegenel eltunik!
  static int initToZero4; //inicializalatlan adatszegmens, mert statikus valtozo
  static int initialized2 = 10; //inicializalt adatszegmens, mert statikus valtozo
  myFunction("something"); //fv-hivas: stack
  int* q = new int[5]; //heap
  delete[] q; //dinamikusan allokalt valtozokat torolni kell, meg mielott a scope-bol eltunnek
        kulonben memory leak!!
}
```

Memóriakezelés – III.

- Külön nézzük meg a stack-et. Függvényhíváskor ennek tartalma így változik:
  - Hívás utána instrukció címe (callback)
  - Visszatérés típusának megfelelő méretű hely lefoglalása
  - A stack tetejére mutat most az ún. stack frame. Minden ami ezután jön szigorúan a fv-hez tartozik (futás után törlődik!)
  - Függvényargumentumok tárolása. Futás közbn létrehozott változók értékei.
- Függvény visszatérésekor:
  - A visszatérés értéke bemásolódik a callback fölötti helyre (megj.: valójában alatta levő, mert stack lefele növekszik – mégis fordítva képzeljük el)
  - Minden törlődik, ami a stack frame pointer fölött helyezkedik el
  - Visszatérés értékének pop-olása, értékadás esetén a célváltozóba történő bemásolás
  - Callback cím pop-olása, és folytatás

Tömbökről röviden

- T típus esetén T[size] jelentése: “size darab T típusú elemből álló tömb”. Indexálás 0-tól (size-1)-ig

```
float v[3]; //v[0], v[1] es v[2]
```

- Index-túlcsordulás katasztrofális, és sokszor nincs futásidejű ellenőrzés... (nézzük meg, hogy Visual Studio-ban van-e)
- Tömb mérete csak konstans (fordításkor ismert) érték lehet. Ha változó méret kell, inkább használjunk STL vektort:

```
int myarr[n]; //nem OK
std::vector<int> myvec(n); //OK
```

- Ha inicializálólistát használunk, akkor a méret automatikus:

```
int myarr[] = {1, 2, 3, 4}; //4-elemu tomb
```

Tömbökről röviden – II.

- String-literálok is tömbnek számítanak. A végükön láthatatlan '\0' karaktert tartalmaznak:
  - "this is a string"
  - sizeof("Bohr")==5
  - "Bohr" típusa: const char[5]
- C-ben és régi C++-ban (C++11-es szabvány előtt) nem-const típushoz is lehetett string literált társítani. Ez ma már nem lehetséges.

```
char* p = "Platon"; //most mar hiba mert nem const!
char[] p = "Zeno"; p[0] = 'R'; //ha modositani szeretnenk, hasznaljunk tombot!

//stringek tarolasa statikus, ezert visszaadhatoak fv-bol (nem tunnek el a stackrol)...
pl. return "range_error"
```

Mutatók

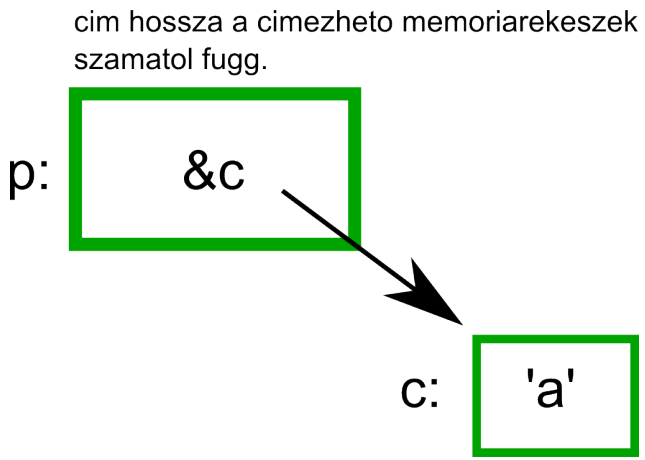
- Világos, hogy objektumra a neve alapján hivatkozhatunk.
- Ugyanakkor a C++-ban legtöbb objektum önálló identitással is rendelkezik. Vagyis két lvalue (értékadó operátor bal oldalán álló "bal oldali érték") akkor is megkülönböztethető, ha értékük u.az, de más mem.területen vannak.
- Ezért objektum hozzáférhető, ha két dolgot ismerünk:
  - az objektum címét
  - az objektum típusát
- Másra nincs is szükség. A mutató (pointer) objektumok címét jelöli.

```
T* ptr; //''T mutatoja'' tipusu ptr valtozo, amely T tipusu objektum memoriacimet tartalmazhatja
```

Mutatók II.

- Mutatókkal kapcsolatos néhány operátor:
  - referencia operátor (objektumból cím)
  - dereferencia operátor (címből objektum vagy érték)

```
char c = 'a';
char* p = = &c; //p c cimet tartalmazza, mert & az un. address-of operator (referencia operator)
char c2 = *p; //c2 == 'a' ; prefix * operator a dereferencia operator
//rvalue-ra alkalmazva erteket ad vissza. lvalue-ra objektumot:
*p = 'b'; //mostantol c erteke is 'b'
```



Mutatók III.

- Mutatók megvalósítása direkt módon függ a gép címzési módjától (byte-ot címez? 4 byte-ot egyszerre?).
- Kevés architektúra tud bitet címezni, ezért a legkisebb "felbontású" pointer char-ra mutat. Ennél kisebb méretű értékeket tárolhatunk:
  - bitenkénti logikai operátorok útján
  - struct-ok bit-fieldjeiben
  - vagy STL bitset objektumokban.

Mutatók IV.: void*

- Alacsonyszintű kódban előfordulhat, hogy úgy akarunk mem.címet átadni, hogy nem tudjuk, milyen típus van ott. Ehhez hasznos a void* mutatótípus.
- void* típusú változóhoz bármilyen pointer típust társíthatunk, kivéve függvényre, illetve osztály tagváltozójára mutató pointert.
- Két void*-ot tesztelhetünk egyenlőségre (mint a pointereknél általában: azonos címre mutatnak-e?), és adott void* pointert explicit módon konvertálhatunk más típusra.
- **Ezt tényleg csak akkor tegyük, ha tudjuk, hogy mit csinálunk!** A programozó felelőssége, hogy tényleg olyan típus legyen ott, amilyet mond.

Mutatók V.: void* II.

- Egyéb műveletek nem biztonságosak, mert a fordító nem tudhatja, mire mutat egy void*. Felhasználáshoz ezért explicit konverzió kell:

```
void f(int* pi)
{
  void * pv = pi; //OK, implicit konverzio mukodik
  *pv; //hiba, void* nem dereferencialhato!
  ++pv; //hiba, nem inkrementalhatjuk, mert nem ismerjuk a mutatott adat meretet

  int* pi2 = static_cast<int*>(pv); //statikus kasztolas rendben
  double* pd1 = pv; //hiba
  double* pd2 = pi; //hiba
  double* pd3 = static_cast<double*>(pv); //nem biztonsagos
  //pv alapbol int-re mutatott, es implementaciotol fugg, mit hogyan tarol
  //pl lehet hogy a double-ok 8 byte-os hatarokon kapnak helyet az int-ek meg nem
}
```

- void* segítségével fv-eknek úgy adhatunk át pointert, hogy semmit sem feltételezhetnek róluk. Alacsonyszintű (hw) használat esetén hasznos, de más esetben gyanakodjunk!

Mutatók VI.

- nullptr karaktersor null pointert jelenti: pointer, amely nem mutat semmire
- Bármilyen pointer típushoz társítható, de másfajta beépített típushoz nem:

```
int* pi = nullptr;
double *pd = nullptr;
int id = nullptr; //hiba, i nem mutato
```

- Gyakran inkább NULL makrót használják, de annak implementációtól függ a definíciója! nullptr használata tisztább.

Mutatók VII.: Mutatók és tömbök

```
int v[] = {1,2,3,4};
int* p1 = v; //p1 a tomb elso elemere mutat implicit konverzio utjan
int* p2 = &v[0]; //ez is u.az
int* p3 = v+4; //utolso utani elemre mutat
```

- Az utolsó sor furcsa, de vannak esetek amikor hasznos (pl. STL iterátorok *end* mutatója az utolsó utáni elemre mutat). Az ilyen mutató értékét sose írjuk vagy olvassuk, csak egyenlőségre tesztelünk.
- Fontos tanulság: tömb implicit módon mutatóvá konvertálódik. Fordítva nem igaz!
- Mutatóra alkalmazott aritmetikai operátor hatása függ attól, hogy milyen típusra mutat a pointer. Ha T* típusú pointerhez 1-et hozzáadunk, p+1 értéke p + sizeof(T) lesz!

Mutatók VIII.: Mutatók és tömbök II.

```
template<typename T>
int byte_diff(T* p, T*q)
{
  return reinterpret_cast<char*>(q) - reinterpret_cast<char*>(p);
  //4-fele explicit kasztolas, szandekosan rondan neznek ki hogy kiugorjanak a szemnek
  //a programozo is latja, hogy "itt vigyazni kell"
  //reinterpret_cast: bitmintazatok jelenteset valtoztatja meg
}

void diff_test()
{
  int vi[10];
  short vs[10];

  cout << vi << ' ' << &vi[1] << ' ' << &vi[1]-&vi[0] << ' ';
  cout << byte_diff(&vi[0], &vi[1]) << endl;
}
```

Mutatók IX.: Mutatók és tömbök III.

- Ha függvénynek tömböt adunk át, implicit módon az első elemre mutató pointerre konvertálódik:

```
int strlen(const char*);

void f()
{
  char v[] = "Annamari";
  int i = strlen(v);
  int j = strlen("Pisti");
}
```

- Ha a pointert dereferenciáljuk és módosítjuk a mutatott értéket, az eredeti tömb is módosul (mondhatjuk, hogy kivételesen pass by reference van, de valójában itt is érték szerinti átadás történik, csak egy hivatkozás értékét adjuk át).
- Persze ilyenkor a függvény nem ismeri a tömb méretét. Ez sok hiba forrása.

Mutatók X.: Mutatók és tömbök IV.

- C-stílusú karakterláncok '\0'-val végződnek, ezért hosszuk számítható (pl. az iménti példában)
- Más esetekben a tömb hosszát is át kell adni a paraméterben:

```
void computeSthg(int* vec_ptr, int vec_size);
```

- Ez azonban régies stílus. Ma már inkább konténereket használunk (vector, array, map stb.) amik önmagukban hordozzák az objektumok méretét, sőt, menedzselik annak memóriafelhasználását is.
- Végül ha mindenképpen tömböt adnánk át, nem pointert, akkor tömb-referenciát kell használni, de a tömb hossza is a paraméter része (referenciákról később lesz csak szó):

```
void f(int(&r)[4]);
```

Mutatók XI.

- Más objektumokra (tömbökre) mutató pointerek kivonása nem értelmezett eredményt ad.
- Pointerek nem adhatók össze (nincs rá értelmes indok se, és a szabvány is kiköti hogy nem lehet)
- Úgy ahogy az objektumoknak van címük, a függvénytörzshöz generált kódnak is van címe a kódszegmensben.
- Pointerek ezért függvényekre is definiálhatók, azonban a szintaxisnál oda kell figyelni a megfelelő zárójelezésre:

```
int* fp(char*) //fp fuggveny int-re mutato pointert ad vissza
int (*fp)(char*) //int-et visszaado fv-re mutato pointer
```

- Ezen kívül a kódot nem módosíthatjuk derferencia-operátorral (na vajon miért?). Ezért függvénypointer csak fv meghívására és címének másolására használható.

Mutatók XII.

```
void error(string s){/*...*/}
void (*efct)(string); //pointer olyan fv-re, mely stringet var es semmit sem ad vissza

void f(){
  efct = &error;
  efct("error"); //fv-hivashoz nincs szukseg dereferencia-operatorra, automatikus!
}
```

- Itt nem kötelező amúgy az error függvény címét sem használni, implicit konverzió miatt.
- Függvénypointer használatakor a függvény szignatúrája nem változik.

Tesztfeladat

- Mit csinál az alábbi kód?

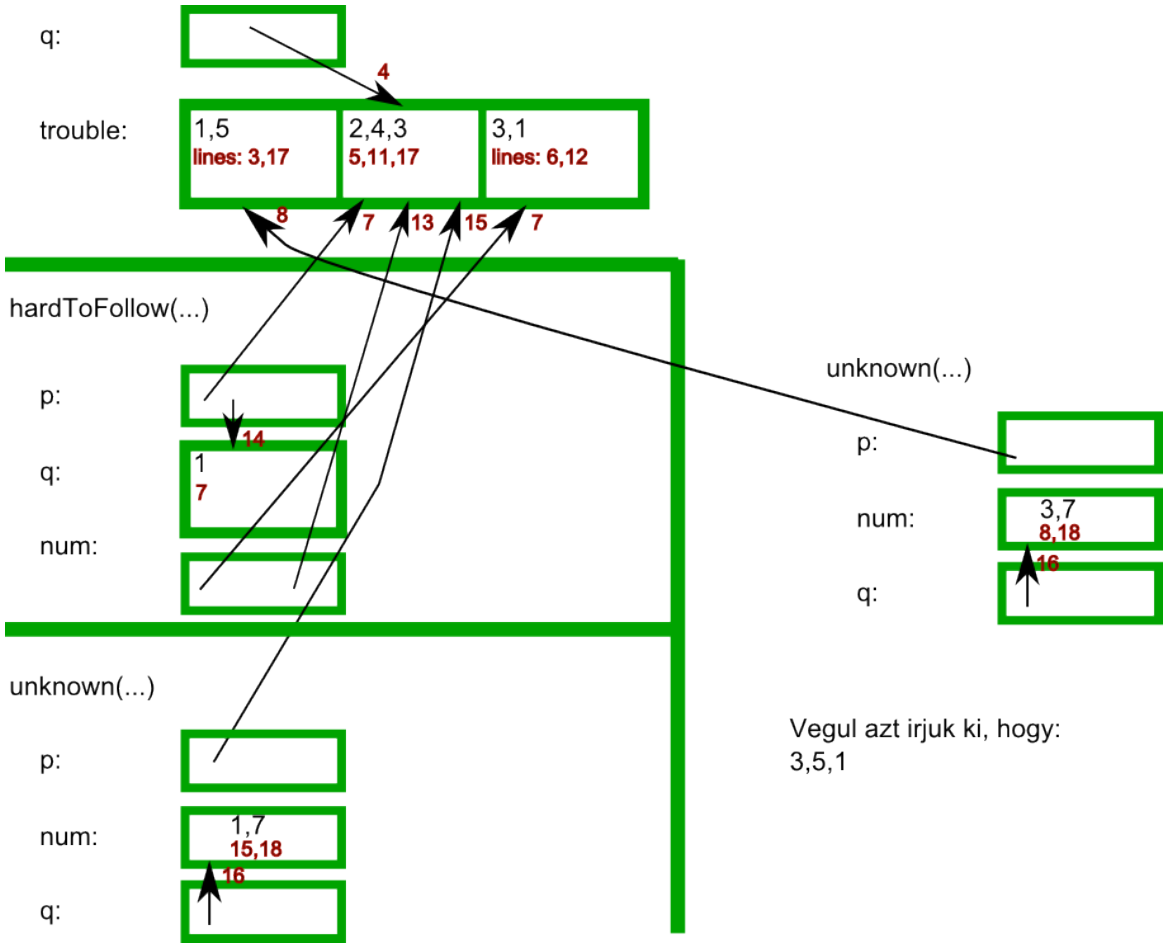
```
void unknown(int* p, int num)
{
  int* q = &num; //line 16
  *p = *q + 2; //line 17
  num = 7; //line 18
}

void hardToFollow(int* p, int q, int *num)
{
  *p = q + *num; //line 11
  *num = q; //line 12
  num = p; //line 13
  p = &q; //line 14
  unknown(num, *p); //line 15
}
```

```
main()
{
  int* q; //line 1
  int trouble[3]; //line 2
  trouble[0] = 1; //line 3
  q = &trouble[1]; //line4
  *q = 2; //line 5
  trouble[2] = 3; //line 6
  hardToFollow(q, trouble[0], &trouble[2]);
        //line 7
  unknown(&trouble[0], *q); //line 8

  std::cout << *q << " " << trouble[0]; //line 9
  std::cout << ", " << trouble[2] << std::endl;
        //line 10
}
```

A megoldás - ld. következő fóliát is!



A megoldás világosabban

id	változó neve	változó értéke (érték vagy id)	kódsor
1	q	id3	4
2	trouble0	1 5	3 17
3	trouble1	2 4 3	5 11 17
4	trouble2	3 1	6 12
5	p-hardToFollow	id3 id6	7 14
6	q-hardToFollow	1	7
7	num-hardToFollow	id4 id3	7 13
8	p-unknown	id3	15
9	num-unknown	1 7	15 18
10	q-unknown	id9	16
11	p-unknown2	id2	8
12	num-unknown2	3 7	8 18
13	q-unknown2	id12	16