Programozás alapjai II.
(3. ea) **C++**

konstruktor és értékadás, dinamikus szerkezetek

Szeberényi Imre
BME IIT
<szebi@iit.bme.hu>

Hol tartunk ?

- C → C++ javítások
- OO paradigmák, objektum fogalma

A C++ csupán eszköz:

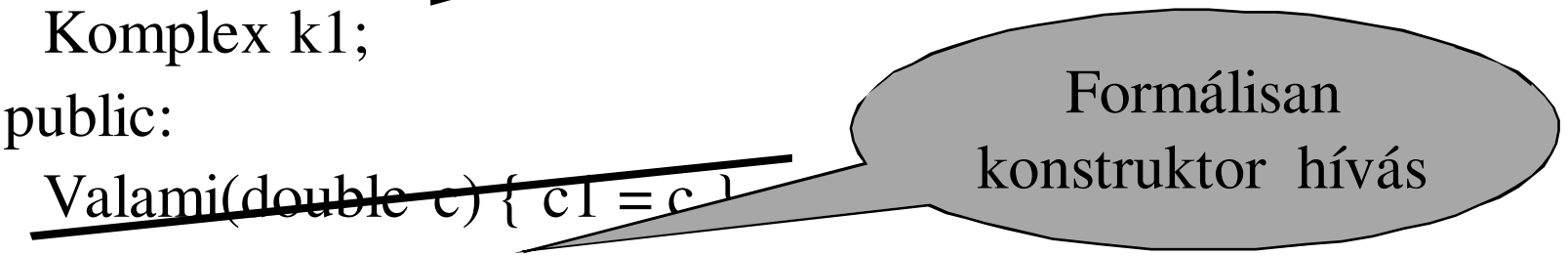
- objektum megvalósítása
  - osztály (egységbe zár, és elszigetel),
  - konstruktor, destruktor, tagfüggvények
  - alapértelmezett operátorok, és tagfüggvények
  - operátor átdefiniálás (függvény átdefiniálás)
- Elegendő eszköz van már a kezünkben?

Konstr: létrehoz+inic. (ism.)

```
class Komplex {
    double re, im;
  public:
    Komplex() { re = 0; im = 0; }
    Komplex(double r) { re = r; im = 0; }
    Komplex(double r, double i) { re = r; im = i; }
    double abs() { return sqrt(re*re+im*im); }
...
};
Komplex k;              // paraméter nélküli (default)
Komplex k1(1);          // 1 paraméteres
Komplex k2(1, 1);       // 2 paraméteres
```

Inicializáló lista

```
class Valami {
   const double c1 = 3.14;  // inicializálni kell, de hogyan?
   Komplex k1;
 public:
   Valami(double c) { c1 = c; }
   Valami(double c) :c1(c) { }
   Valami(double c, Komplex k) :c1(c), k1(k) { }
};
```



Konstans tag, és referencia tag, csak inicializáló listával inicializálható. Célszerű a tagváltozókat is inicializáló listával inicializálni (felesleges műveletek elkerülése).

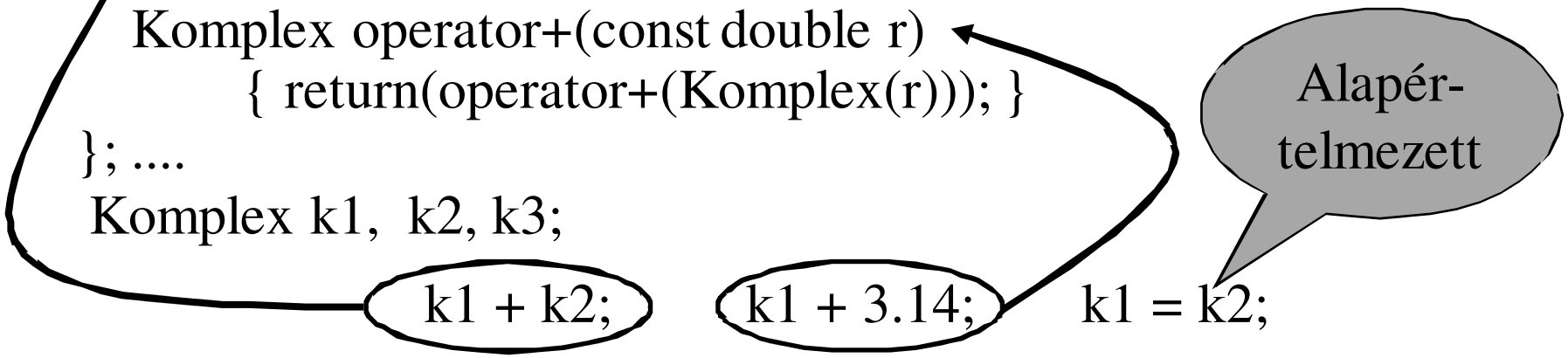
Destruktor: megszüntet (ism.)

```
class Dinamikus {
        int *pData;             // pointer valamilyen adatara
        double value;           // valamilyen másik adat
public:
        Dinamikus(int len) {
            pData = new int[len]; // terület lefoglalása
          ....
        }
        ~Dinamikus() { delete[] pData; } // terület felszabadítása
};
```

A pData és a value megszüntetése automatikus, ahogy egy lokális változó is megszűnik. A new-val foglalt dinamikus terület felszabadítása azonban a mi feladatunk, ahogyan C-ben is fel kell szabadítani a dinamikusan foglalt területet.

Műveletekkel bővített Komplex (ism.)

```
class Komplex {
  double re, im;
public:    ....
   Komplex operator+(const Komplex& k)
         { Komplex sum(k.re + re, k.im + im); return(sum); }
   Komplex operator+(const double r)
         { return(operator+(Komplex(r))); }
}; ....
 Komplex k1,  k2, k3;
```




```
3.14 + k1;    // bal oldal nem osztály !
              // Ezért globális függvény  kell !
```

double + Komplex (ism.)

```
class Komplex { ..... };
Globális fv., nem tagfüggvény:
Komplex operator+(const double r, const Komplex& k) {
      return(Komplex(k.re + r,    k.im));
}
```

Baj van! Nem férünk hozzá, mivel privát adat!

1. megoldás: privát adat elérése pub. fv. használatával:

```
Komplex operator+(const double r, const Komplex& k) {
       return(Komplex(k.getRe() + r,   k.getIm()));
}
```

Publikus lekérdező függvény

2. megoldás: védelem enyhítése

- Szükséges lehet a privát adatok elérése egy globális, függvényből, vagy egy másik osztály tagfüggvényéből.
- Az ún. barát függvények hozzáférhetnek az osztály privát adataihoz. Rontja az áttekinthetőséget, ezért nem kedveljük.

```
class Komplex { ..... public:
// FONTOS! Ez nem tagfüggvény, csak így jelöli, hogy barát
friend Komplex operator+(const double r, const Komplex& k);
};
```

```
Komplex operator+(const double r, const Komplex& k) {
        k.re ..... k.im.... // hozzáfér a privát adatohoz
}
```

Alapértelmezett tagfüggvények (ism.)

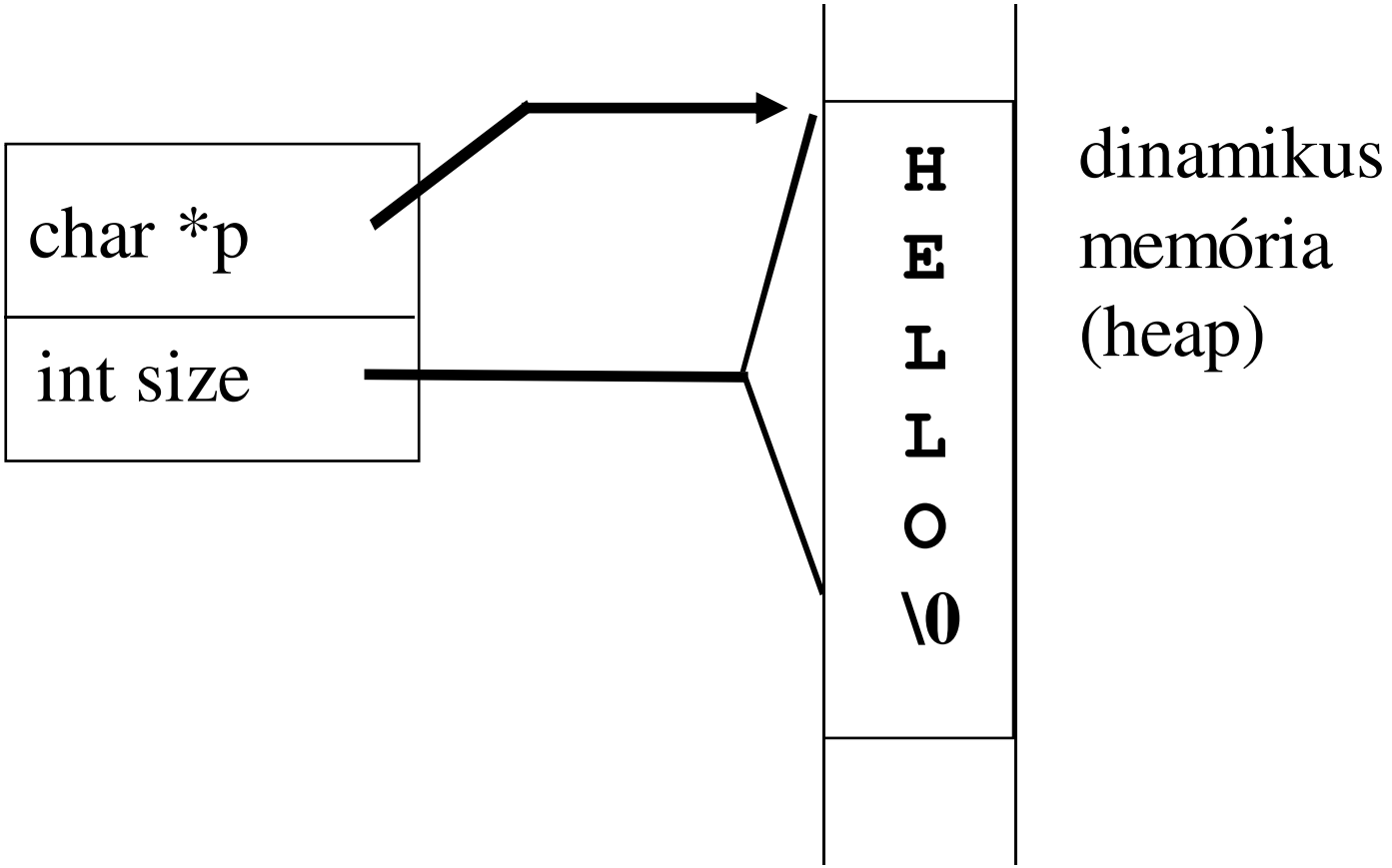
- Konstruktor
  - default: X() // nincs paramétere
  - másoló: X(const X&) // referencia paraméter
- Destruktor
- operator=(const X&) // értékadó
- operator&() // címképző
- operator*() // dereferáló
- operator->() // tag elérése pointerrel
- operator,(const X&) // vessző

A másoló konstruktor és az értékadó operátor alapértelmezés szerint meghívja az adattagok megfelelő tagfüggvényét.
Alaptípus esetén (bitenként) másol!

Példa: Intelligens string

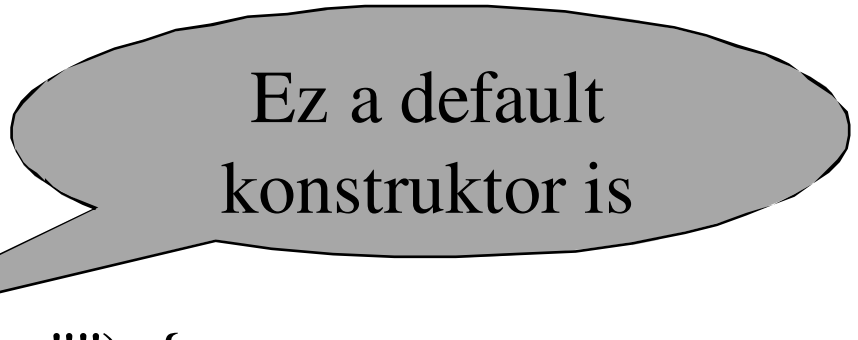
- String tárolására alkalmas objektum, ami csak annyi helyet foglal a memóriában, amennyi feltétlenül szükséges. → dinamikus adatszerkezet
- Műveletei:
  - létrehozás, megszüntetés
  - indexelés: []
  - másolás: =
  - összehasonlítás: ==
  - összefűzés: (String + String), (String + char) (char + String)
  - kiírás: cout <<
  - beolvasás: cin >>

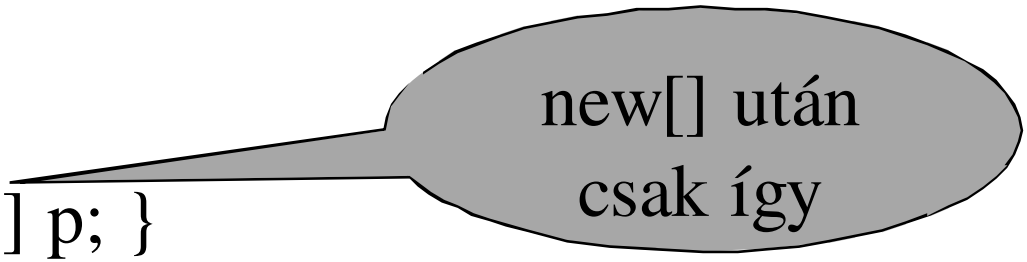
String adatszekezete



String osztály

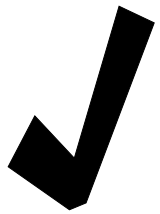
```
class String {
        char *p;
        unsigned int size;
public:
        String(const char *s = "") {
            p = new char[(size = strlen(s)) + 1];
            strcpy(p, s);
        }
        ~String( ) { delete[] p; }
        char& operator[] (int i) { return p[i]; }
        const char& operator[] (int i) const { return p[i]; }
};
```





Függvényhívás mint balérték

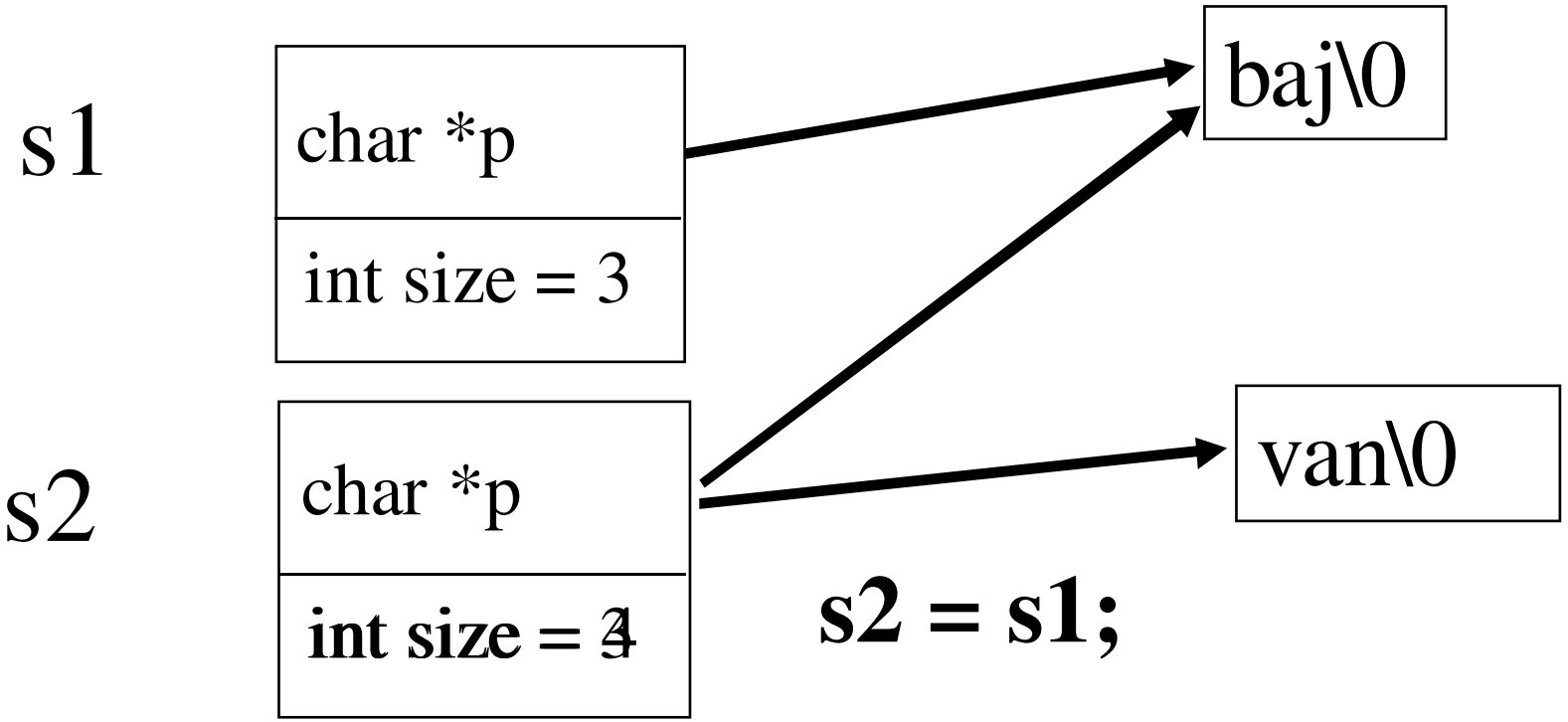
```
main ( ) {
        String s("Hello");  const String cs("Konstans");

        char c = s[3]; // c = s.operator[](3);  ➔ c=p[3];

         c = cs[4]; // c = cs.operator[](4) const; ➔ c=p[4];

        s[1]='u';       // s.operator[](1) =' u'; ➔ p[1]='u';
}                       // destruktor: delete[] p
```



} // destruktor cs-re, majd az s-re

Értékadás problémája / 2

{ String s1("baj"); String s2("van!");



} // destruktor “baj”-ra 2x, “van”-ra 0x

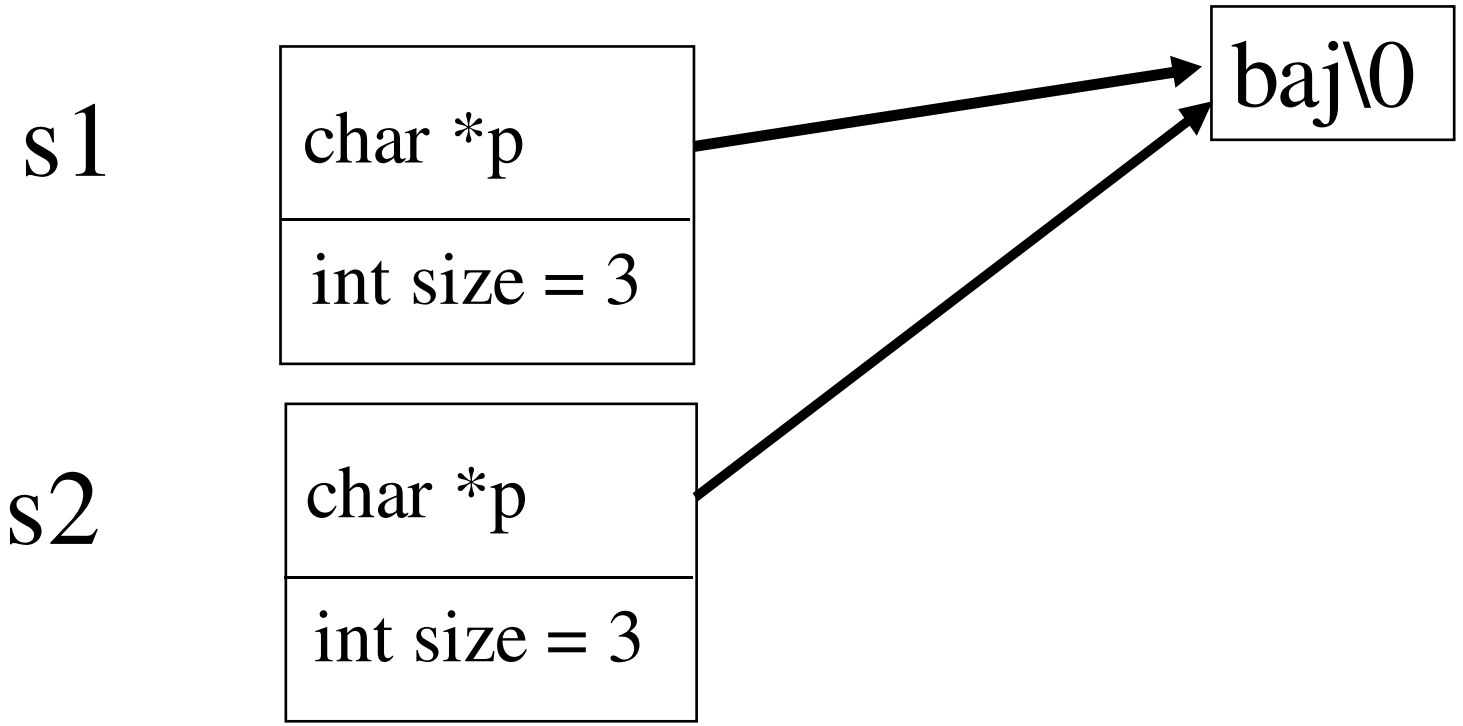
Megoldás: operátor= átdefiniálása

Paraméterként kapja azt, amit értékül kell adni egy **létező** objektumnak.

```
class String {
  ....
  String& operator=(const String& s) { // s1=s2=s3 miatt
        if (this != &s ) {                  // s = s miatt
              delete[] p;
              p = new char[(size = s.size) + 1];
              strcpy(p, s.p);
        }
        return *this;                       // visszaadja saját magát
    }
};
```

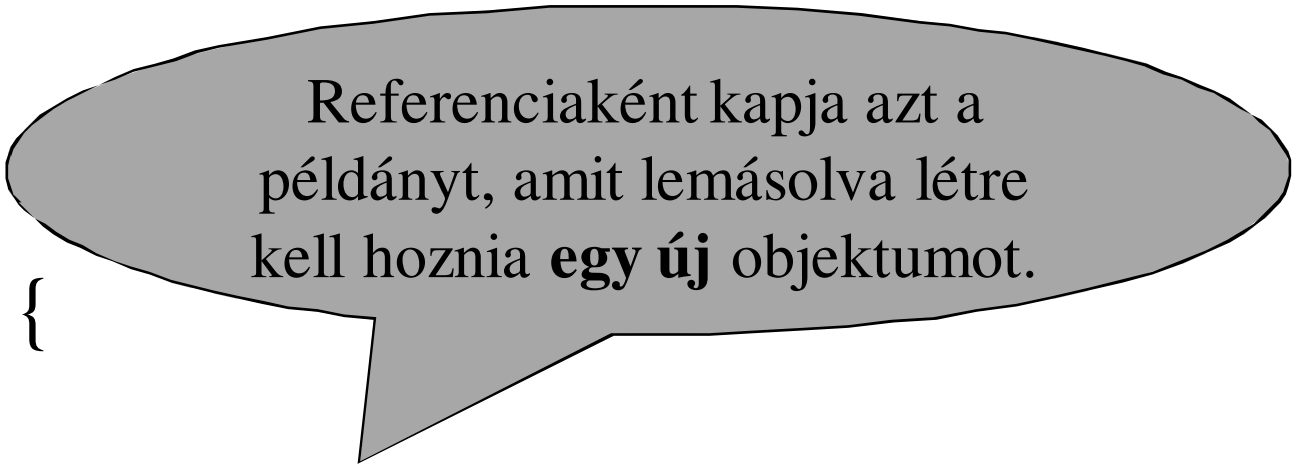
Kezdeti értékadás problémája

{ String s1("baj"); String s2 = s1;



} // destruktor “baj”-ra 2x

Megoldás: másoló konstruktor

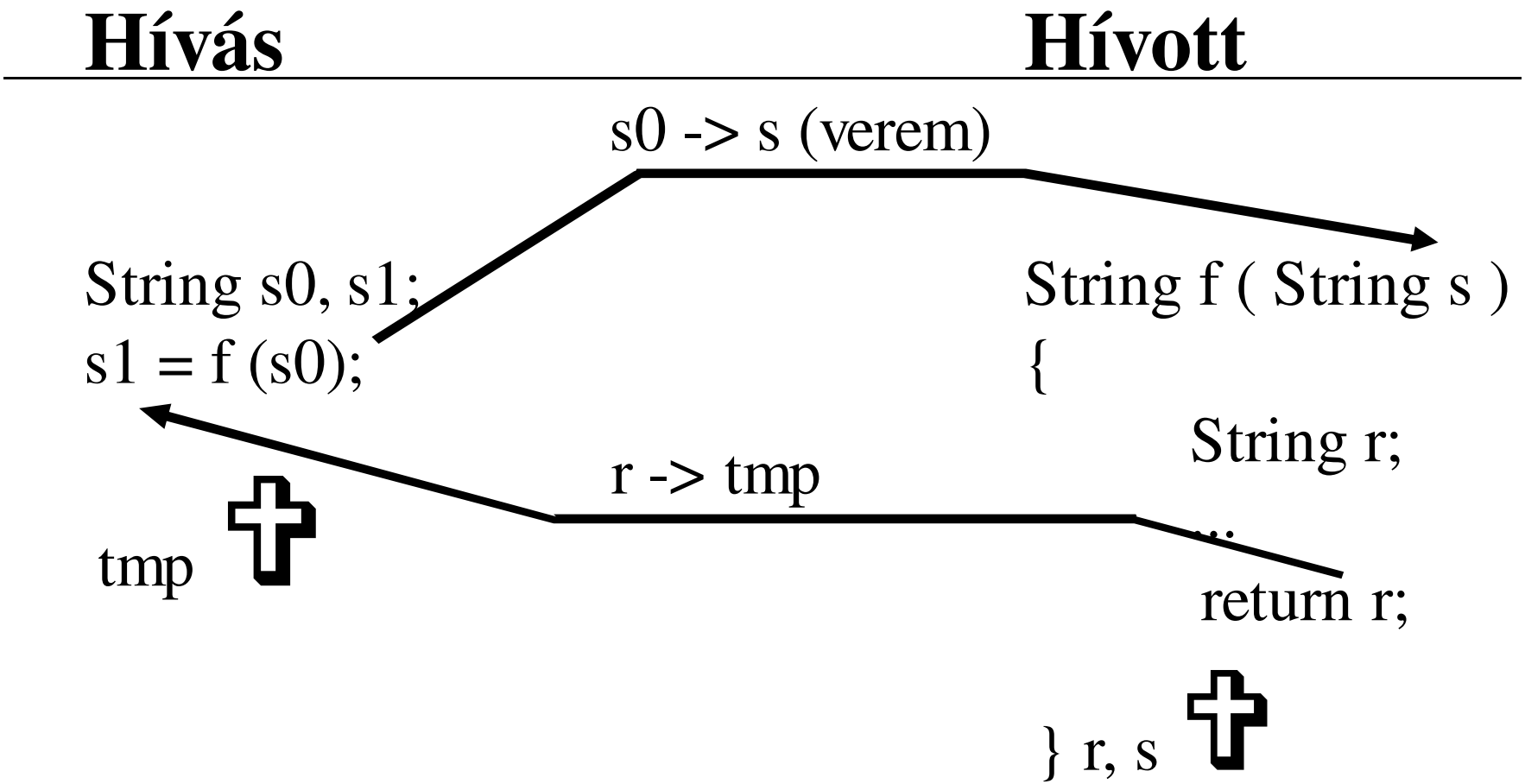



```
class String {
   ....
        String(const String& s) {
             p = new char[(size = s.size) + 1];
             strcpy(p, s.p);
        }
}
```

Miért más mint az értékadás?

- A kezdeti értékadáskor még inicializálatlan a változó (nem létezik), ezért nem lehet a másolással azonos módon kezelni.
- Mikor hívódik a másoló konstruktor?
  - inicializáláskor (azonos típussal inicializálunk)
  - függvény paraméterének átadásakor
  - függvény visszatérési értékének átvételekor
  - ideiglenes változók összetett kifejezésekben
  - kivétel átadásakor

Függvényhívás és visszatérés



Összetett algebrai kifejezés

```
String s, s0, s1, s2;
s = s0 + s1 + s2;
```

1. lépés: tmp1=s0+s1

2. lépés: tmp2=tmp1+s2

3. lépés: s = tmp2

4. lépés: tmp1, tmp2 megszüntetése destruktor hívással

String rejtvény

```
class String {
   char *p;
   int    size;
public:
   String( );                          // 1
   String(char *);                     // 2
   String(String&);                    // 3
   ~String( );                         // 4
   String  operator+(String&);         // 5
   char&  operator[](int);             // 6
   String& operator=(String&);// 7
};
```

```
main( ) {
      String s1("rejtvény");  2
      String s2;              1
      String s3 = s2;         3

      char c = s3[3];         6
      s2 = s3;                7
      s2 = s3 + s2 + s1;      5,3,5,3,
                              (3),7,4,4,(4)
} // destr.                   4,4,4
```

String rejtvény/2

```
class String {
   char *p;
   int    size;
public:
   String( );                          // 1
   String(char *);                     // 2
   String(String&);                    // 3
   ~String( );                         // 4
   String  operator+(String&);         // 5
   char&  operator[](int);             // 6
   String& operator=(String);          // 7
};
```

```
main( ) {
      String s1("rejtvény");   2
      String s2;               1
      String s3 = s2;          3

      char c = s3[3];          6
      s2 = s3;                 3,7,4
      s2 = s3 + s2 + s1;       5,3,5,3,
                          (3),3,7,4,4,4,(4)
} // destr.                    4,4,4
```

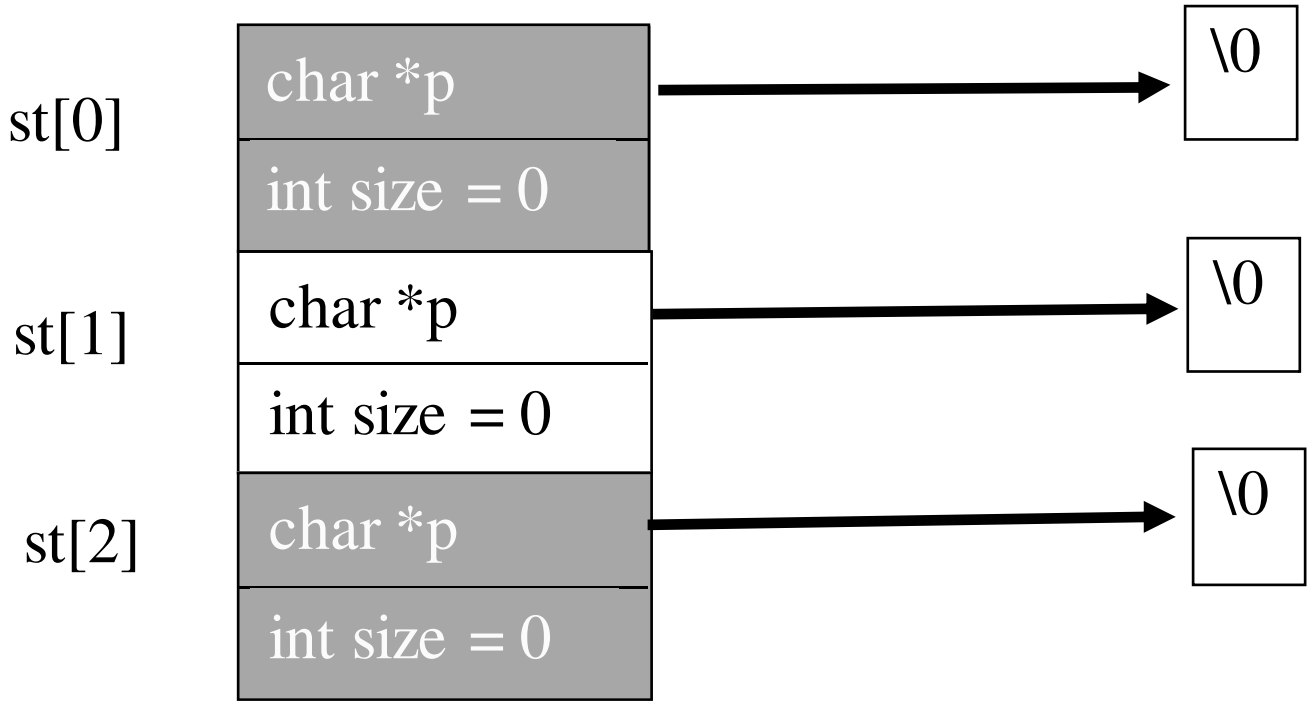
Miért referencia ?

Miért kell referencia a másoló konstruktorhoz?

- A paraméterátadás definíció szerint másoló konstruktort hív.
- Ha a másoló konstruktor nem referenciát, hanem értéket kapna, akkor végtelen ciklus lenne.

Miért fontos a delete[ ] ?

String *st = new String[3];



A delete st hatására csak a *st, azaz az st[0] destruktora hívódik meg! Az st[1] és az st[2] által foglalt memória nem szabadul fel! A delete[] meghívja minden elem destruktorát.

String +

Védelem enyhítése

Nem tagfüggvény!

```
class String {
    ....
    String operator+(const String& s);
    String operator+(char c);
    friend String operator+(char c, const
                                    String& s);
};
String operator+(char c, const String& s) {
    char *p = new char[s.size + 2];
    *p = c; strcpy (p+1, s.p);
    String ret(p); delete[] p;
    return ret;
}
```

Keletkezett-e += ?

- Az alaptípusokra meghatározott műveletek közötti logikai összefüggések nem érvényesek a származtatott típusokra.
- Azaz az operator= és az operator+ meglétéből nem következik az operator +=
- Ha szükség van rá, definiálni kell.

Változtatható viselkedés

- Feladat: "Varázsütésre" az összes String csupa nagybetűvel íródjon ki!
- Megoldás: viselkedést befolyásoló jelző, de hol?
  - objektum állapota (adata) – csak az adott példányra van hatása.
  - globális változó – elég ronda megoldás !
  - az osztályhoz rendelt állapot: statikus tag ill. tagfüggvény.

Statikus tag

- Az osztályban statikusan deklarált tag nem példányosodik.
- Pontosan egy példány létezik, amit explicit módon definiálni kell (létre kell hozni).
- Minden objektum ugyanazt a tagot éri el.
- Nem szükséges objetummal hivatkozni rá.

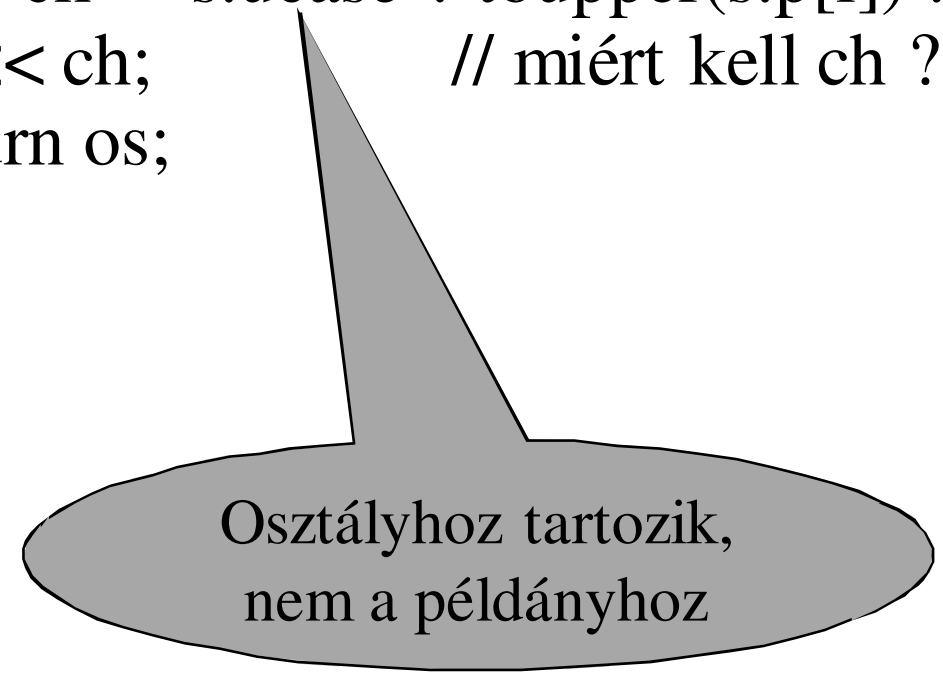
  pl: String**::**SetUcase(true);
- Statikus tagként az osztály tartalmazhatja önmagát.
- Felhasználás: globális változók elrejtése

String statikus taggal

```
class String {
        char *p;   int   size;
   static bool ucase;          // statikus tag deklarálása
public:
   ....
   static bool SetUcase(bool b = true) {
        bool prev = ucase; ucase = b; return (prev);
   }
   friend ostream& operator<<(ostream& os, String& s);
};
bool String::ucase = false; // FONTOS: létre kell hozni !!
```

String statikus taggal /2

```
ostream& operator<<(ostream& os, String& s) {
      for (i = 0; i < s.size; i++) {
         char ch  = s.ucase ? toupper(s.p[i]) : s.p[i];
         os << ch;                  // miért kell ch ?
       } return os;
}
```



Komplex példa újból

- Olvassunk be adott/tetszőleges számú komplex számot és írjuk ki a számokat és abszolút értéküket fordított sorrendben!
- Objektumok:
  - Komplex,
  - KomplexTar
    - konstruktorban adott méret (a, változat)
    - igény szerint változtatja a méretét (b, változat)
  - Mindkét megoldás dinamikus memóriakezelést igényel. Ügyelni kell a helyes felszabadításra, foglalásra.

KomplexTar osztály

```
class KomplexTar {
   Komplex *t;          // pointer a dinamikusan foglalt tömbre
   int db;              // tömb mérete (elemek száma)
public:
   class Tar_Hiba { }; // osztály az osztályban a hibakezeléshez
   KomplexTar(int m = 10) :db(m) {
         t = new Komplex[m]; }          // konstruktor (def = 10)
   KomplexTar(const KomplexTar& kt);// másoló konstruktor
   Komplex& operator[](int i);          // indexelés
   KomplexTar& operator=(const KomplexTar& kt); // értékadás
   ~KomplexTar() { delete[] t;}         // felszabadítás
};
```

KomplexTar osztály/2

```
KomplexTar::KomplexTar(const KomplexTar& kt) {//másoló konst.
    t = new Komplex[db = kt.db];
    for (int i = 0; i < db; i++)  t[i] = kt.t[i]; // miért nem memcpy ?
}
```

A memcpy nem hívná meg a konstruktort

```
KomplexTar& KomplexTar::operator=(const KomplexTar& kt) {// =
    if (this != &kt) {
        delete[] t; t = new Komplex[db = kt.db];
        for (int i = 0; i < db; i++)  t[i] = kt.t[i]; // miért nem memcpy ?
    }
    return *this;
}
```

Visszavezettük értékadásra

```
KomplexTar::KomplexTar(const KomplexTar& kt){//másoló 2.vált.
        t = NULL;  *this = kt;  // trükkös, de rendben van !
}
```

Indexelés és a főprogram (a,)

```
Komplex& KomplexTar::operator[](int i) {
   if (i >= db || i < 0) throw Tar_Hiba();  return t[i];
}

int main() {
   KomplexTar t(5);              // a tárolóban 5 elemünk van
   try {
      for (int i = 0; i < 20; i++)   cin >> t[i];   // beolvasás
      KomplexTar t2 = t1;       // másoló konstruktor
      for (i = 19; i >= 0; i--)
         cout << t[i] ' ' << (double)t[i] << endl; // kiírás
   } catch (KomplexTar::Tar_hiba) {
         cerr << "Indexelesi hiba\n";    // hibakezelés
    }
   return(0);
}
```

Változó méretű KomplexTar (b,)

```
// Indexelés hatására növekszik a méret, ha kell
Komplex& KomplexTar::operator[](int i)
{
  if (i < 0) throw Tar_Hiba();   // hibás indexelés
  if (i >= db) {   // növekednie kell, célszerű kvantumokban
     Komplex *tmp = new Komplex[i+10]; // legyen nagyobb
     for (int j = 0; j < db; j++)  tmp[j] = t[j]; // átmásol
     delete[] t;    // régi törlése
     t = tmp;       // pointer az új területre
     db = i + 10; // megnövelt méret
  }
  return t[i];      // referencia vissza
}
```

Összefoglalás /1

- INICIALIZÁLÁS != ÉRTÉKADÁS
- Inicializáló lista szerepe.
- Alapértelmezett tagfüggvények.
- Dinamikus szerkezeteknél nagyon fontos a másoló konstruktor és az értékadás felüldefiniálása. (nem maradhat alapért.)
- Default konstruktornak fontos szerepe van a tömböknél.

Összefoglalás /2

- Konstans tagfüggvények nem változtatják az objektum állapotát.
- Statikus tag és tagfüggvény az osztályhoz tartozik.
- Védelem enyhítése: friend
- Létrehozás, megsemmisítés feladatait a konstruktor és destruktor látja el.

Létrehozás, megsemmisítés

- Konstruktor
  - default: X() // nincs paramétere
    automatikusan létrejön, ha nincs másik konst.
  - másoló: X(const X&) // referencia paramétere van,
    automatikusan létrejön: meghívja az adattagok másoló konst.-rát, ha objektumok, egyébként bitenként másol.
- Destruktor
  - delete[ ] // [ ] nélkül csak a 0. tömbelemre!!
  - automatikusan létrejön: meghívja az adattagok destr.
- operator=(const X&) // értékadó operátor
  automatikusan létrejön: meghívja az adattagok értékadó operátorát, ha objektumok, egyébként bitenként másol.

Milyen furcsa kommentek!

- A kommentekből automatikusan generál dokumentációt a Doxygen program. (html, latex, rtf, man, ... formátumban)
- Csak jó kommentből lesz jó dokumentáció!

Speciális kezdet

Rövid leírás ponttal zárva

Részletesebb leírás

```
/**
 * Komplex osztály.
 * Komplex viselkedést megvalósító osztály.
 * Csak a feladat megoldásához szükséges műveleteket definiáltuk.
 */
class Komplex {
```

Milyen furcsa kommentek! /2

Speciális kezdet

Rövid leírás ponttal zárva

```
/**
 * Komplex osztály.
 * Komplex viselkedést megvalósító osztály.
 * Csak a feladat megoldásához szükséges műveleteket definiáltuk.
 */
class Komplex {
```

Részletesebb leírás

Részletes leírás

Komplex osztály.

Komplex viselkedést megvalósító osztály. Csak a feladat megoldásához szükséges műveleteket definiáltuk.

Definíció a(z) **komplex_oo2.cpp** fájl **28**. sorában.

Ez a dokumentáció az osztályról a következő fájl alapján készült:

- **komplex_oo2.cpp**

Projekt: komplex Készült: Fri Feb 24 00:14:43 2006 Készítette: doxygen *1.4.6-NO*

Milyen furcsa kommentek! /3

```
class Komplex {
    ....
/**
  * Konstruktor nulla, egy és két paraméterrel
  * @param r - valós rész (alapértelmezése 0)
  * @param i - képzetes rész (alapértelmezése 0)
  */
  Komplex(double r = 0, double i = 0) :re(r), im(i) { }
  operator double()     { return sqrt(re*re + im*im); }  ///< abszolút érték
  friend istream& operator>>(istream& s, Komplex& k) ; ///< Komplex beolvasás
  friend ostream& operator<<(ostream& s, const Komplex k); ///< Komplex kiírás
```

Paraméterek dokumentálása

Speciális kezdet

Rövid leírás

Konstruktorok és destruktorok dokumentációja

Komplex::Komplex (double *r* = 0,
double *i* = 0
) [inline]

Konstruktor nulla, egy és két paraméterrel.

Paraméterek:

r - valós rész (alapértelezése 0)
i - képzetes rélsz (alapértelmezése 0)

A tagok teljes listája

Publikus tagfüggvények

	Komplex (double r=0, double i=0) *Konstruktor nulla, egy és két paraméterrel.*
	operator double () *abszolút érték*

Barátok

istream &	operator>> (istream &s, Komplex &k) *Komplex beolvasás.*
ostream &	operator<< (ostream &s, const Komplex k) *Komplex kiírás.*

Tesztelési követelmények

- Legyen független, és megismételhető
- Legyen áttekinthető és tükrözze a tesztelt kód struktúráját.
- Legyen hordozható és újrafelhasználható.
- Segítsen a teszt írójának a problémára koncentrálni.
- Legyen gyors és automatizálható.
- Gyakran a tárolóba (svn) való betétel feltétele az ún. unit teszt sikeressége.

Google Test

- Kis méretű, forráskódban elérhető
  http://code.google.com/p/googletest/
- Platformfüggetlen (WinX, MAC, LINUX, Windows Mobile, MinGW, …)
- Assertion – alapú
  - success, nonfatal, fatal
- Teszt program:
  - teszt esetek
    - tesztek

Assertion

- Hasonlító függvényeket hívnak
  - Hiba esetén kiírják a hiba helyét, kódját
- ASSERT_*
  - fatális hiba – a program megáll
- EXPECT_*
  - nem fatális hiba – tovább fut

```
ASSERT_EQ(2*2, 4) << "2*2 hiba";
for (int i = 0; i < 10; i++) {
   EXPECT_LT(i-1, 2*i) << "i nem kisebb mint 2*i? i=" << i;
}
```

Egyszerű feltételek

Utasítás	Teljesülnie kell
ASSERT_EQ(*expected*, *actual*)	*expected* == *actual*
ASSERT_NE(*val1*, *val2*)	*val1* != *val2*
ASSERT_LT(*val1*, *val2*)	*val1* < *val2*
ASSERT_LE(*val1*, *val2*)	*val1* <= *val2*
ASSERT_GT(*val1*, *val2*)	*val1* > *val2*
ASSERT_GE(*val1*, *val2*)	*val1* >= *val2*
ASSERT_STREQ(exp_str, act_str)	*a két C string azonos*
ASSERT_STRNE(str1, str2);	*a két C string nem azonos*
ASSERT_STRCASEEQ(exp, act);	*a két C string azonos (kis/nagy betű az.)*
ASSERT_STRCASENE(str1, str2)	*a két C string nem azonos (kis/nagy b.)*

Egyszerű példa

```
#include "gtest/gtest.h"
#include "komplex.h"

TEST(KomplexTeszt,  ValosReszVizsgalata)  {
    const Komplex k1(3, 1);
    EXPECT_EQ(3, k1.getRe ())  << "Valos rész nem OK!";
}

TEST(KomplexTeszt,  KepzetesReszVizsgalata)  {
   const Komplex k1(3, 2);
   ASSERT_EQ(2,  k1.getIm ()) << "Képzetes rész nem OK!";
}
```

Problémák a HSZK-ban

- VS ingyenes változatával nem fordul (tuple par.)
- CB-vel rendben, van de érzékeny a jó lib-re.
  - include
  - win32
    - gtest.lib, gtest_main-mdd.lib (Visual Studio-hoz)
    - libgtestd.a, libgtest_maind.a (CodeBloks-hoz)
- További beállításokra lehet szükség:
  - Projektfájlban a megfelelő útnév beállítása:
    - CB: Project ->Build options ->Linker settings, Search directories
    - VS: Project-> Properties -> C++ -> General (additional dir),
      Project-> Properties -> linker -> Input (Additional dep.)

gtest_lite

- A prog2 tárgyhoz készült.
- Lényegében a gtest fontosabb lehetőségeit valósítja meg kompatibilis módon.
- Ronda makrókkal és statikus objektummal operál.
- HF-ben használható ill. használandó.
- Csak egyszerű teszteket támogat

gtest_lite példa

```
#include "gtest_lite.h"
#include "komplex.h"

TEST(KomplexTeszt, ValosReszVizsgalata) {
   const Komplex k1(3, 1);
   EXPECT_EQ(3, k1.getRe ()) << "Valos rész nem OK!";
} END

TEST(KomplexTeszt, KepzetesReszVizsgalata) {
  const Komplex k1(3, 2);
  ASSERT_EQ(2, k1.getIm ()) << "Képzetes rész nem OK!";
} END
```

További ellenőrzések

Utasítás	Teljesülnie kell
ASSERT_THROW(*statement, excep_type*)	adott típust *dobnia kell*
ASSERT_ANY_THROW(*statement*)	*bármit kell dobnia*
ASSERT_NO_THROW(*statement*)	*nem dobhat*
ASSERT_PRED_FORMAT1(*pred, val1*)	*pred(val1) == true*
ASSERT_PRED_FORMAT2(*pred, val1, val2*)	*pred(val1, val2) == true*
ASSERT_FLOAT_EQ(*expected, actual*)	*expected ~= actual*
ASSERT_DOUBLE_EQ(*expected, actual*)	*expected ~= actual*
ASSERT_NEAR(*val1, val2, abs_error*)	abs(val1-val2) <= abs_error
SUCCEED()	siker
FAIL(); ADD_FAILURE()	végzetes; nem végzetes
ADD_FAILURE_AT("*file_path*", *line_number*);	nem végzetes