

# *Programozás alapjai II.*

## *(4. ea) C++*

*konstruktor és értékadás, dinamikus szerkezetek*

---

Szeberényi Imre, Somogyi Péter

BME IIT

<szebi@iit.bme.hu>



MŰEGYETEM 1782

# Hol tartunk ?

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

## A C++ csupán eszköz:

- objektum megvalósítása
  - osztály (egységbe zár, és elszigetel),
  - konstruktor, destruktork, tagfüggvények
  - alapértelmezett operátorok, és tagfüggvények
  - operátorok túlterhelése (függvény túlterhelés)
- Elegendő eszköz van már a kezünkben?

# Konstr: létrehoz+alapáll. (ism.)

A programozott törzs lefutása előtt számos feladata van. Pl.: létrehozza az adattagokat: hívja azok konstruktorát.

```
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() const { return sqrt(re*re+im*im); }
...
};
Komplex k;           // paraméter nélkül hívható (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) { }  
};
```

Inicializáló lista

Később pontosítjuk

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

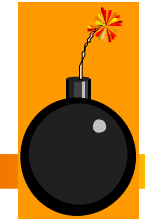
# *Destruktor: megszüntet (ism.)*

```
class String {  
    size_t len;           // tároljuk a hosszt  
    char *pData;         // pointer az adatara  
public:  
    String(size_t l) {  
        pData = new char[len = l]; // terület lefoglalása  
        ....  
    }  
    ~String() { delete[] pData; } // terület felszabadítása  
};
```

A programozott törzs lefutása után feladata van. Pl.:  
megszünteti az adattagokat: hívja azok destruktort

A pData és a len 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.

# Explicit destruktor hívás?



A programozott törzs lefutása után több dolga dolga van. Pl. megszünteti az adattagokat: hívja azok destruktorát, ha van. Ezért explicit módon hívni csak nagyon speciális esetben lehet. (placement new)

~~Ilyet NE!~~

```
{  
    String s1; s1.~String();  
    String* sp = new String, sp->~String();  
};
```



# 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;
```

$k1 + k2;$

$k1 + 3.14;$

$k1 = k2;$

Alapér-  
telmezett

```
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 átláthatóságot és gyengíti az egységbezárás elvét, 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) {  
    return(Komplex(k.re + r, k.im)); // hozzáfér a privát adathoz  
}
```

# Alapértelmezett tagfüggvények (ism.)

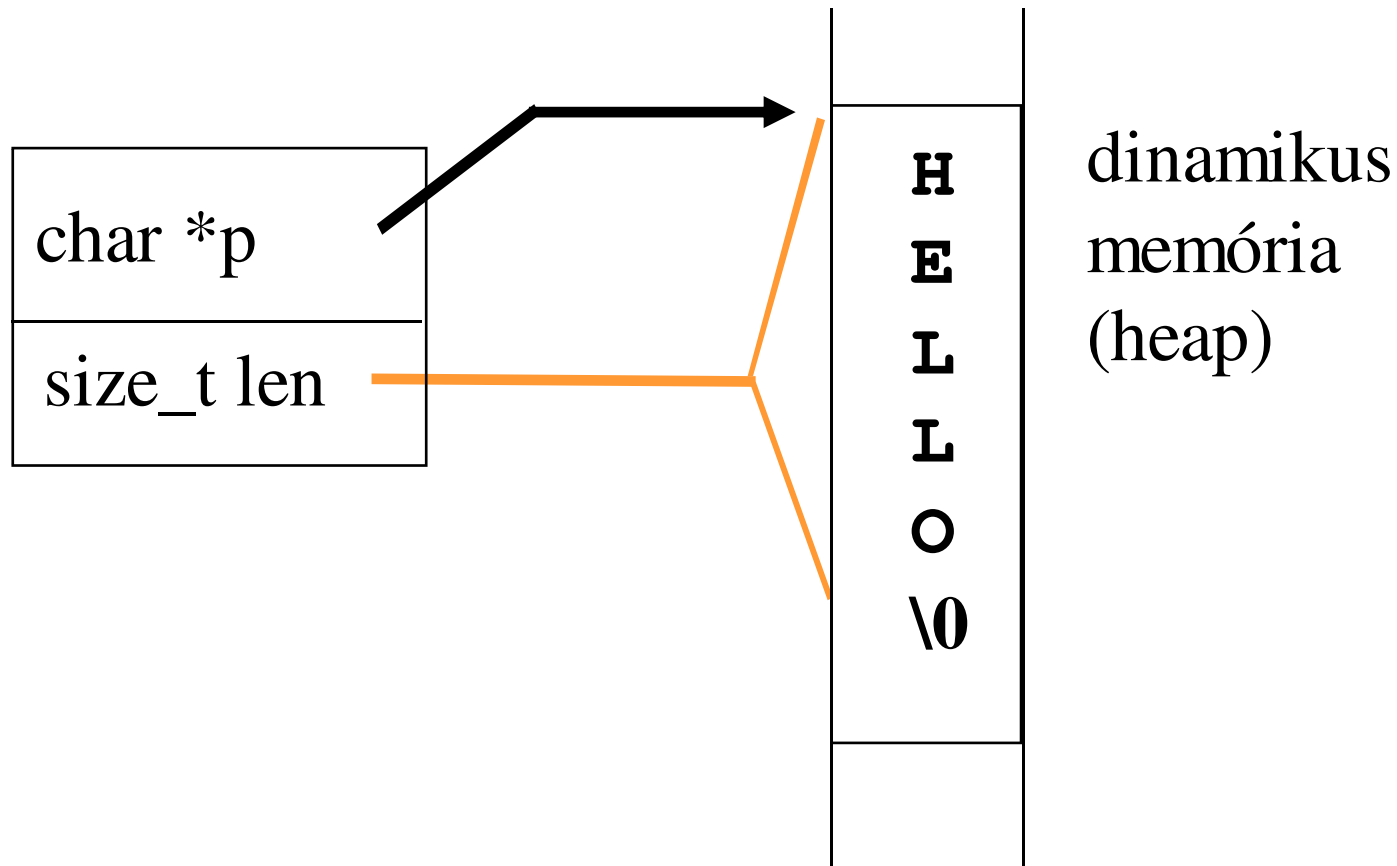
- Konstruktor
  - default: `X()` // nincs paramétere
  - másoló: `X(const X&)` // referencia paraméter
- Destruktor
- `operator=(const X&)` // értékadó
- `operator&()` // címképző
- `operator*()` // dereferáló
- `operator->()` // tag elérése pointerrel
- `operator,(const X&)` // vessző

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 adatszerkezete



# String osztály

```
class String {  
    char *p;  
    size_t len;  
public:  
    String(const char *s = "") {  
        p = new char[(len = strlen(s)) + 1];  
        strncpy(p, s, len); p[len] = 0;  
    }  
    ~String( ) { delete[] p; }  
    const char& operator[] (int i) const { return p[i]; }  
    char& operator[] (int i) { return p[i]; }  
};
```


Ez a default  
(paraméter nélkül hívható)  
konstruktor is

new[] után  
csak így!

referenciát ad, így  
balérték is lehet

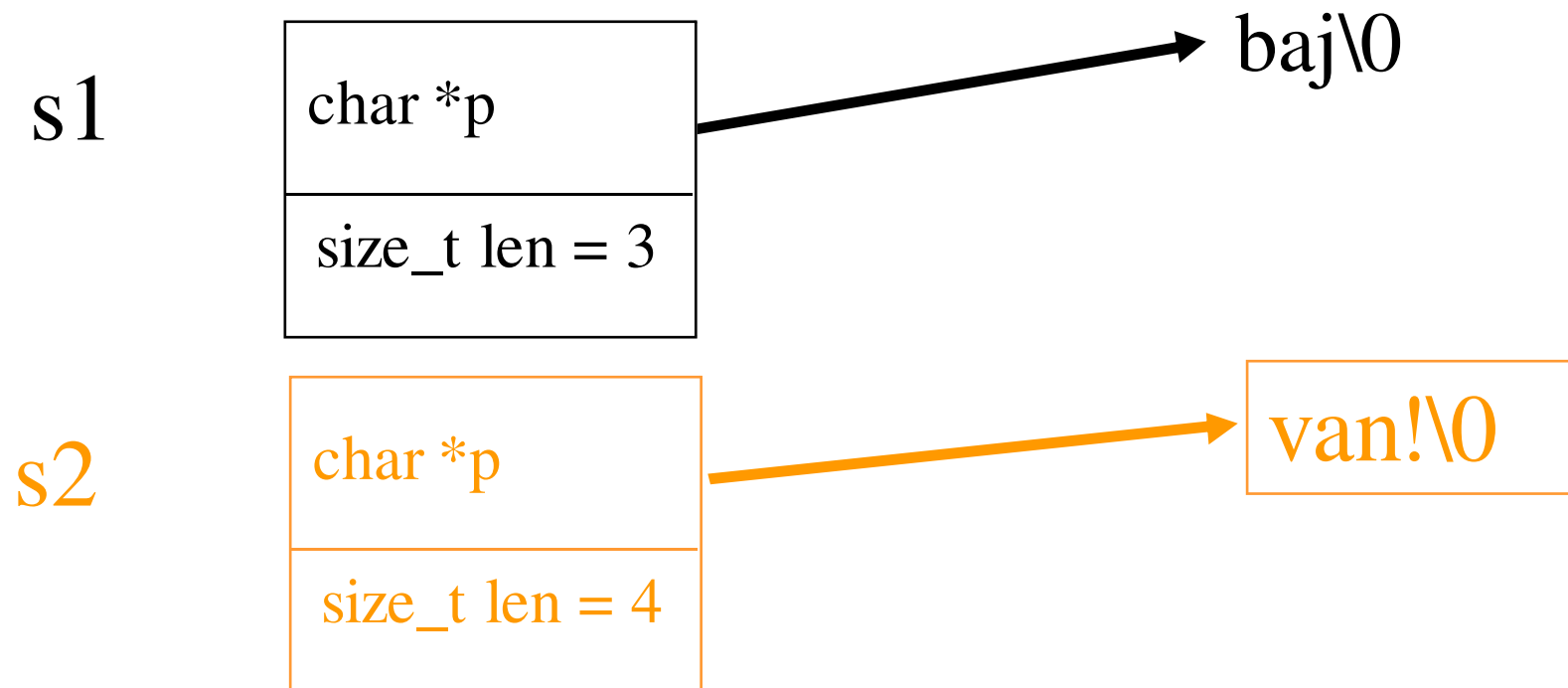
# Függvényhívás mint balérték

```
int main ( ) {  
    String s("Hello"); const String cs("Konstans");  
    // konstruktorok: s.p = new char[6] s.len = 5  
    //                  cs.p = new char[9] s.len = 8  
  
    char c = s[3]; // c = s.operator[](3); → c=s.p[3];  
  
    c = cs[4]; // c = cs.operator[](4) const; → c=cs.p[4];  
  
    s[1]='u'; // s.operator[](1) = 'u'; → s.p[1]='u';  
  
} // destruktorkok: delete[] cs.p,  
  //                delete[] s.p
```



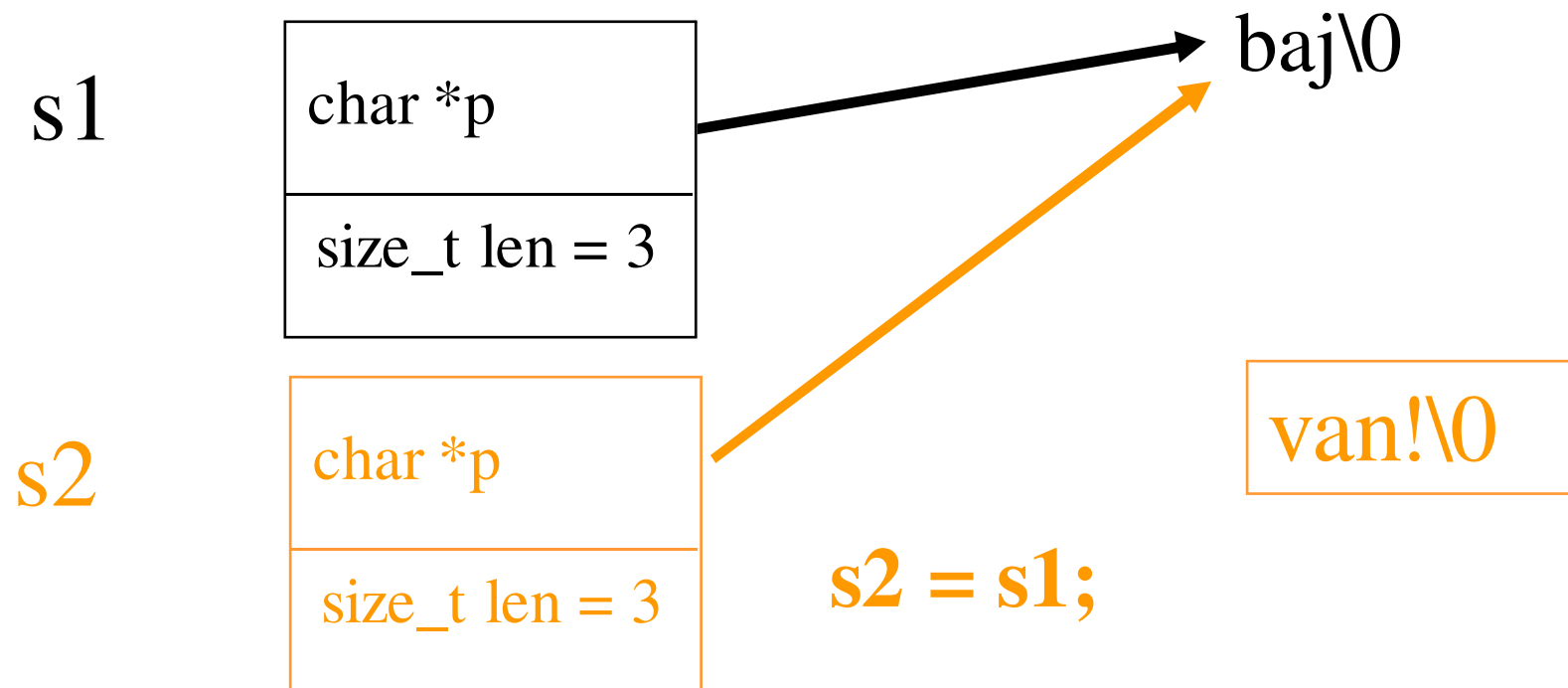
# Értékadás problémája

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



# Értékadás problémája

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



```
} // destruktorkor „baj”-ra 2x, „van”-ra 0x
```



## Megoldás: operátor= átdefiniálás

```
class String {
```

```
....
```

```
String& operator=(const String& s) { // s1=s2=s3 miatt
```

```
    if (this != &s ) { // s = s kivédésére
```

```
        delete[] p;
```

```
        p = new char[(len = s.len) + 1];
```

```
        strncpy(p, s.p, len); p[len] = 0;
```

```
    }
```

```
    return *this; // visszaadja saját magát
```

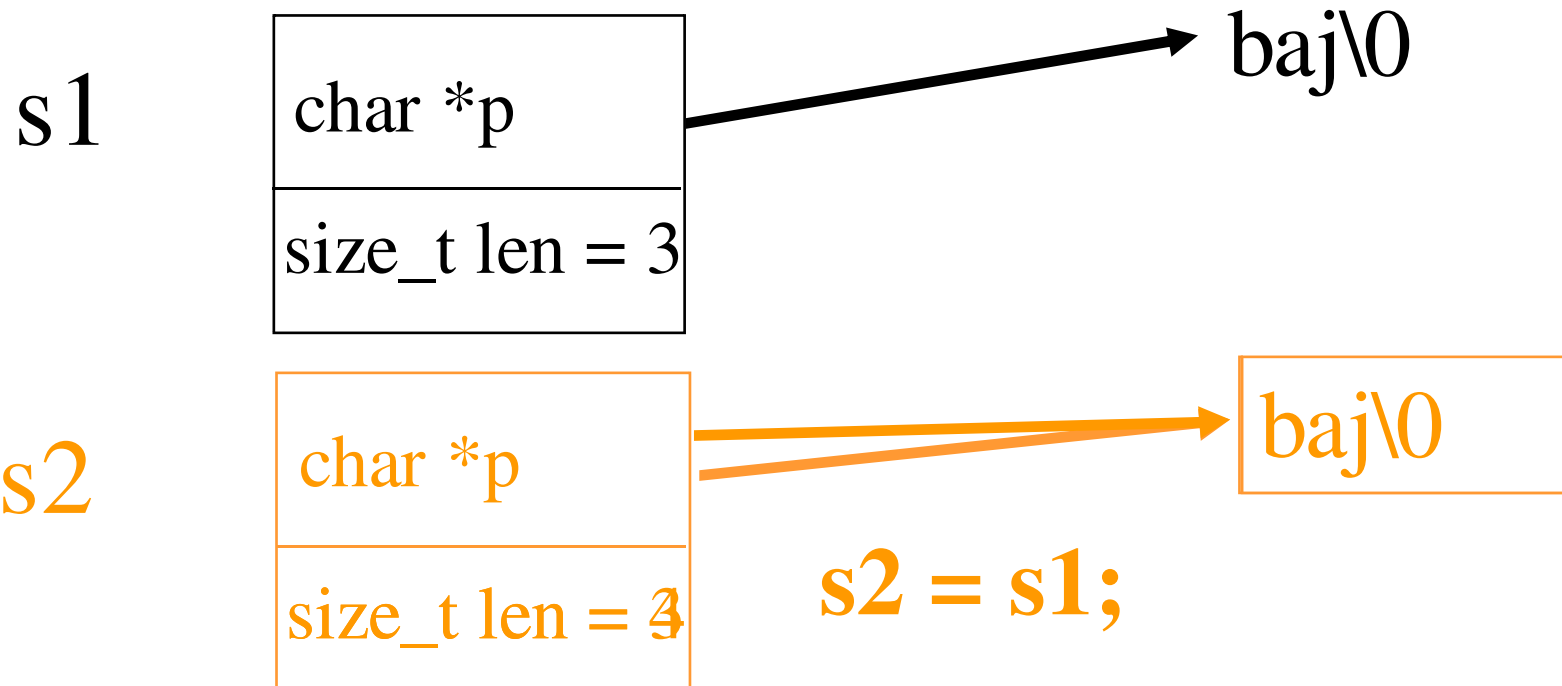
```
}
```

```
};
```

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

# *operator=-vel már nincs baj*

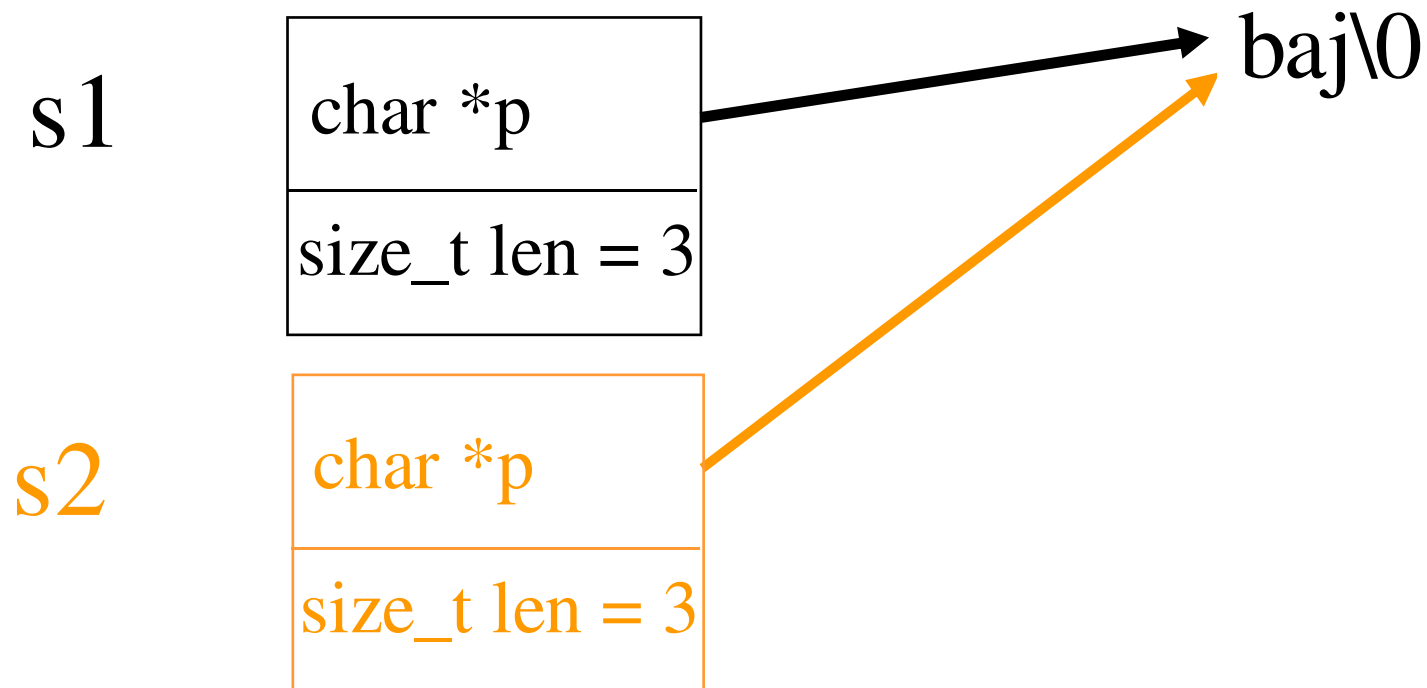
```
{ String s1("baj"); String s2("van!");
```



```
} // destruktorkok rendben
```

# Kezdeti értékadás problémája

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



```
} // destruktorkok „baj”-ra 2x
```

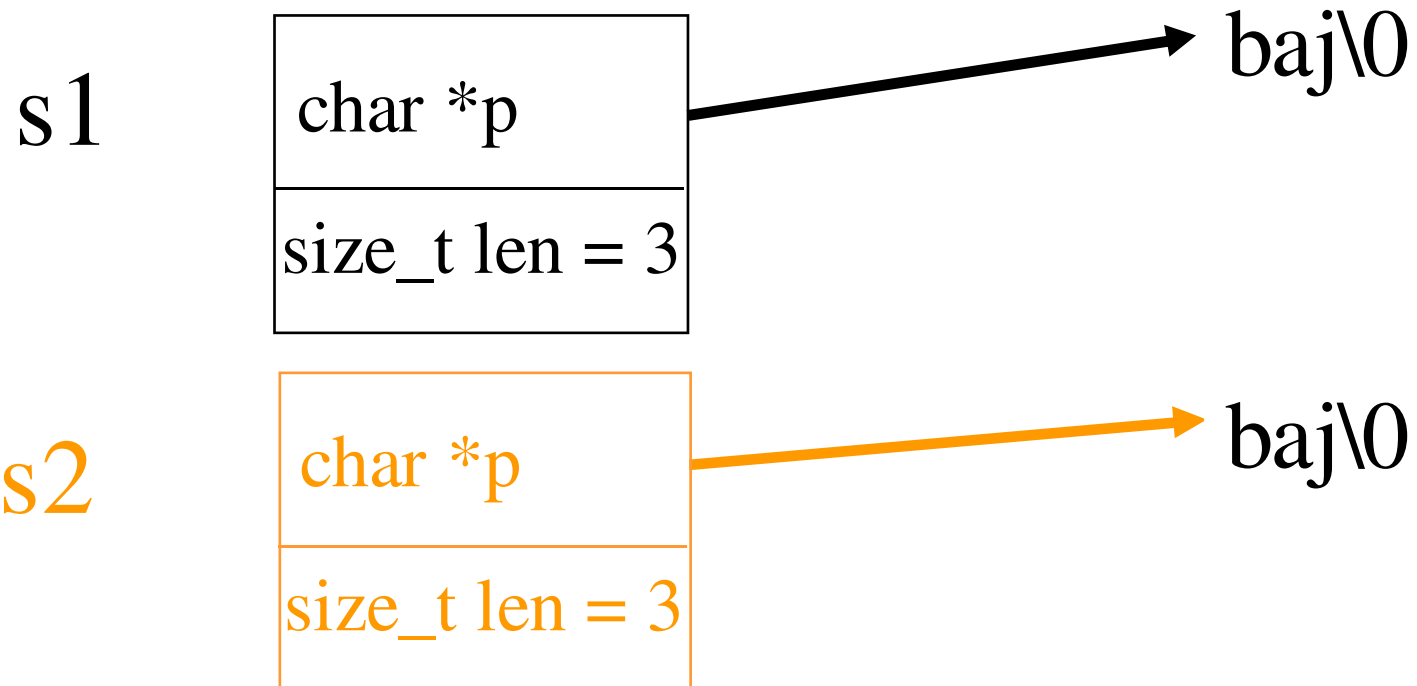
# Megoldás: másoló konstruktor

Referenciaként kapja azt a példányt, amit lemásolva létre kell hoznia egy új objektumot.

```
class String {  
    ....  
    String(const String& s) {  
        p = new char[(len = s.len) + 1];  
        strncpy(p, s.p, len); p[len] = 0;  
    }  
}
```

# Másoló konstruktorral már jó

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



```
} // destruktorkok rendben
```

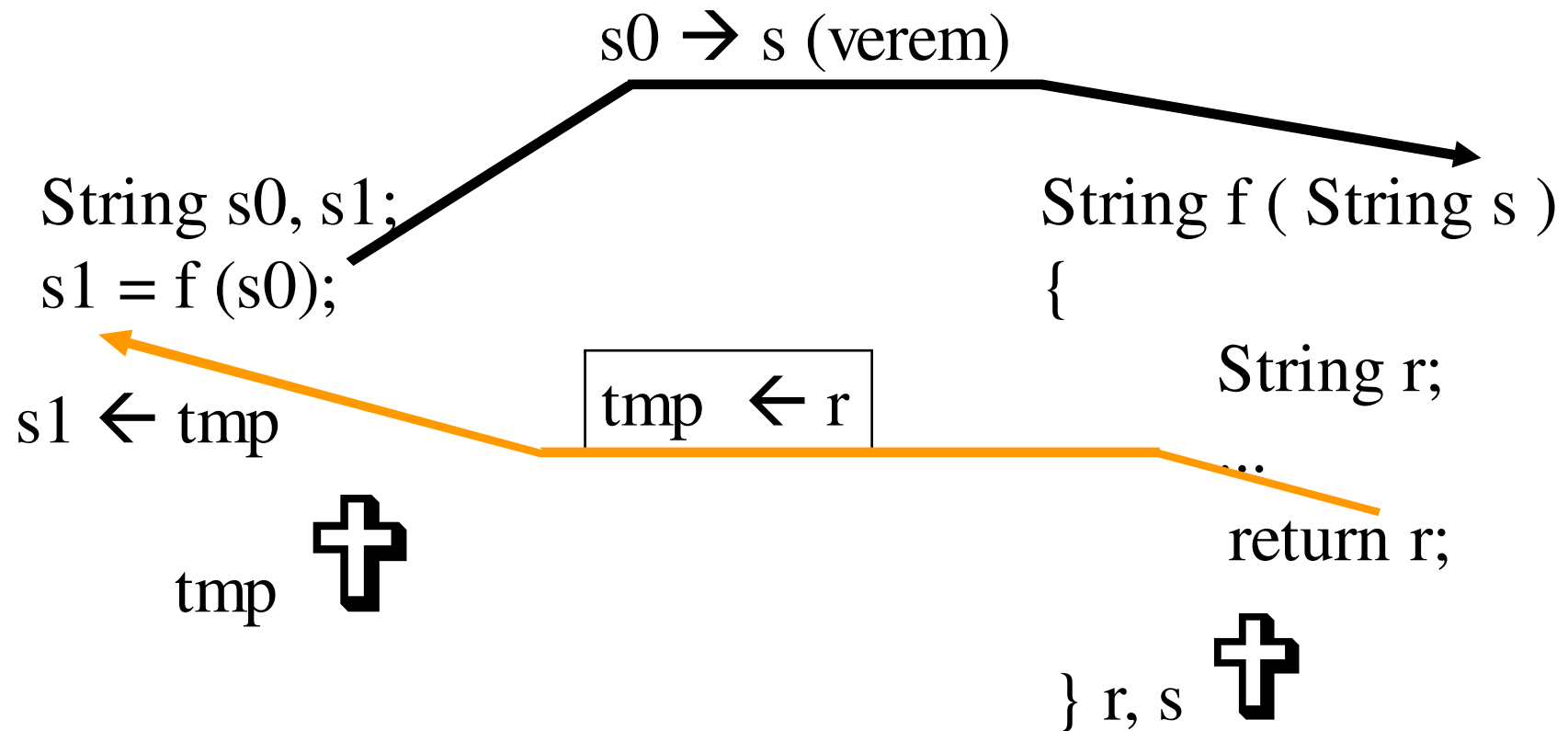
## *Miért más mint az értékadás?*

- 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 létrehozásakor
  - **kivétel** átadásakor

# Függvényhívás és visszatérés

**Hívás**

**Hívott**



# 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 destruktorkívással



# String rejtvény

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

```
int 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)
    return 0; // destr. 4,4,4
}
```

# String rejtvény/2

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

```
int 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)
    return 0; // destr. 4,4,4
}
```

# *Miért referencia ?*

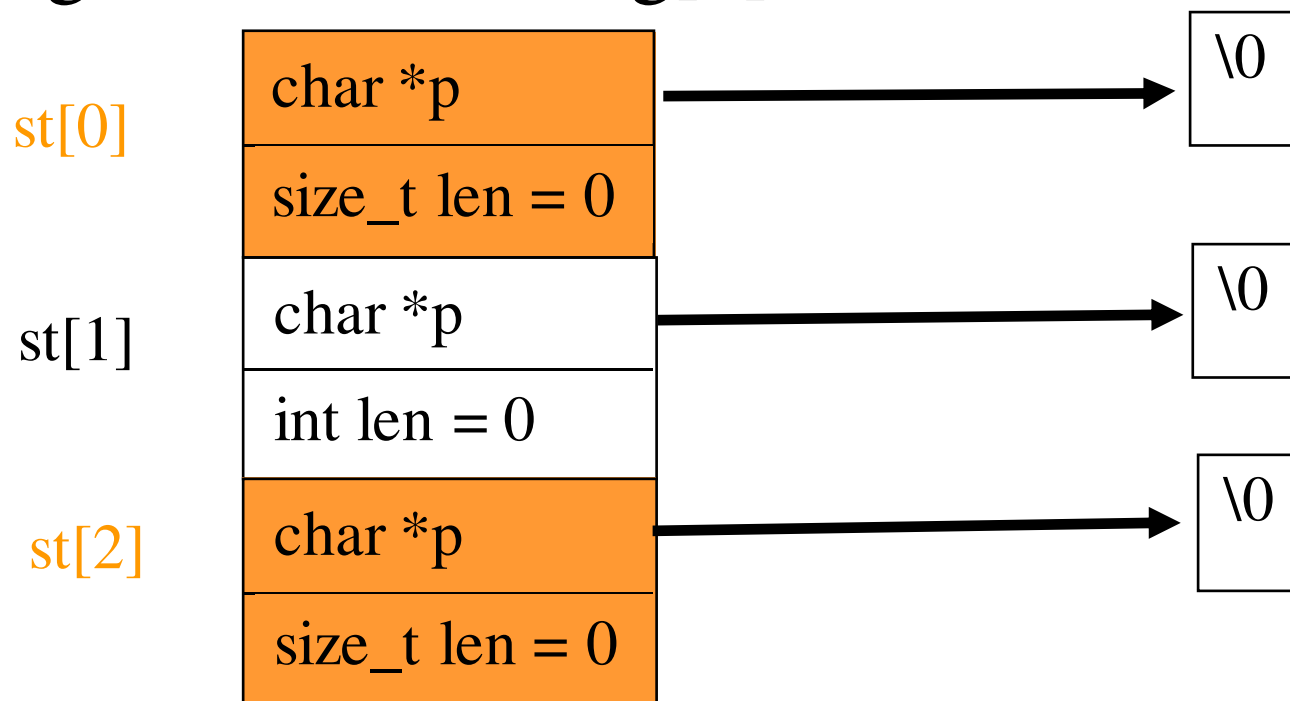
---

## 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 destruktort.

# String +

```
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.len + 2];
```

```
    *p = c; strncpy (p+1, s.p, s.len);
```

```
    p[s.len+1] = 0;
```

```
    String ret(p); delete[] p;
```

```
    return ret;
```

```
}
```

Védelem  
enyhítése

Nem  
tagfüggvény!

## *String + friend nélkül*

```
class String {  
    ....  
    String operator+(const String& s);  
    String operator+(char c);  
};
```

```
String operator+(char c, const String& s) {  
    return String(c) + s;  
}
```

Egyszerűbb és szebb is.

## *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**.
- Pontosán egy példány létezik, amit **explicit** módon **definiálni** kell (létre kell hozni).
- Minden objektum ugyanazt a tagot éri el.
- Nem szükséges objektummal hivatkozni rá.  
**pl: `String::SetUcase(true);`**
- Statikus tagként az osztály tartalmazhatja önmagát.
- Felhasználás: globális változók elrejtése

# String statikus taggal

```
class String {
    char *p; unsigned int len;
    static bool ucase; // statikus adattag deklarálása
public:
    ....
    static void Ucase(bool b) { ucase = b; } // beállít
    static bool Ucase() { return ucase; } // lekérdez
    friend ostream& operator<<(ostream& os, const String& s);
};
bool String::ucase = false; // Definíció FONTOS !!
```

Adattag definíciója

## String statikus taggal /2

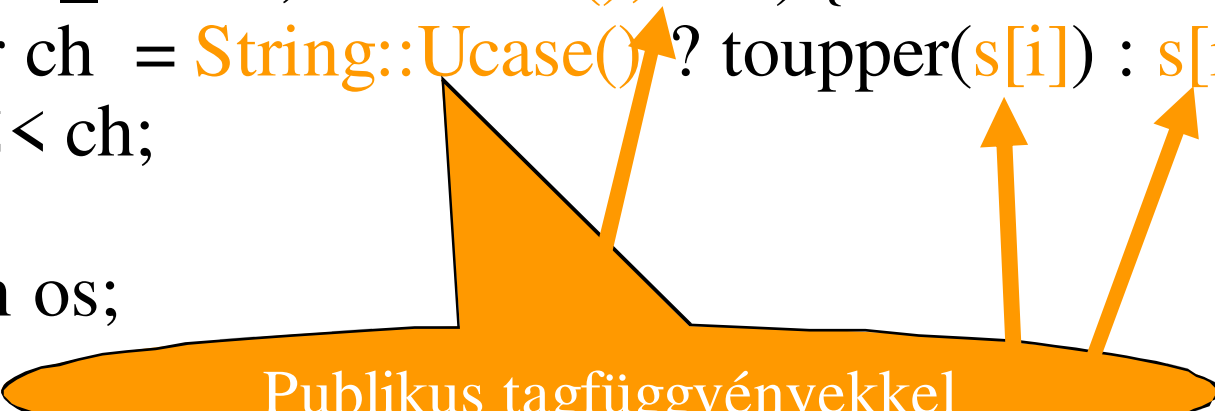
```
ostream& operator<<(ostream& os, const String& s) {  
    for (size_t i = 0; i < s.len; i++) {  
        char ch = String::ucase ? toupper(s.p[i]) : s.p[i];  
        os << ch; // miért kell ch ?  
    }  
    return os;  
}
```

Osztályhoz tartozik, nem a példányhoz.  
Lehetne `s::ucase` is

# String statikus taggal /3

**//Friend nélkül:**

```
ostream& operator<<(ostream& os, const String& s) {  
    for (size_t i = 0; i < s.size(); i++) {  
        char ch = String::Ucase()? toupper(s[i]) : s[i];  
        os << ch;  
    }  
    return os;  
}
```



Publikus tagfüggvényekkel

The diagram consists of an orange oval at the bottom containing the text 'Publikus tagfüggvényekkel'. Three orange arrows point upwards from this oval to the code elements 'String::Ucase()', 's[i]', and 's[i]' in the provided code block.

# Statikus adattag inicializálás

Ha a statikus adattag konstans és integrál- vagy enum típusú, akkor inicializálható az osztály deklarációjában is. Ez a definíciót is kiváltja.

```
struct A {  
    static const int a = 35; // definíció is  
    static enum { s1, s2} const st = s1;  
};
```

```
const int A::a = 35;
```

Nem kell/szabad

# Adattag inicializálás

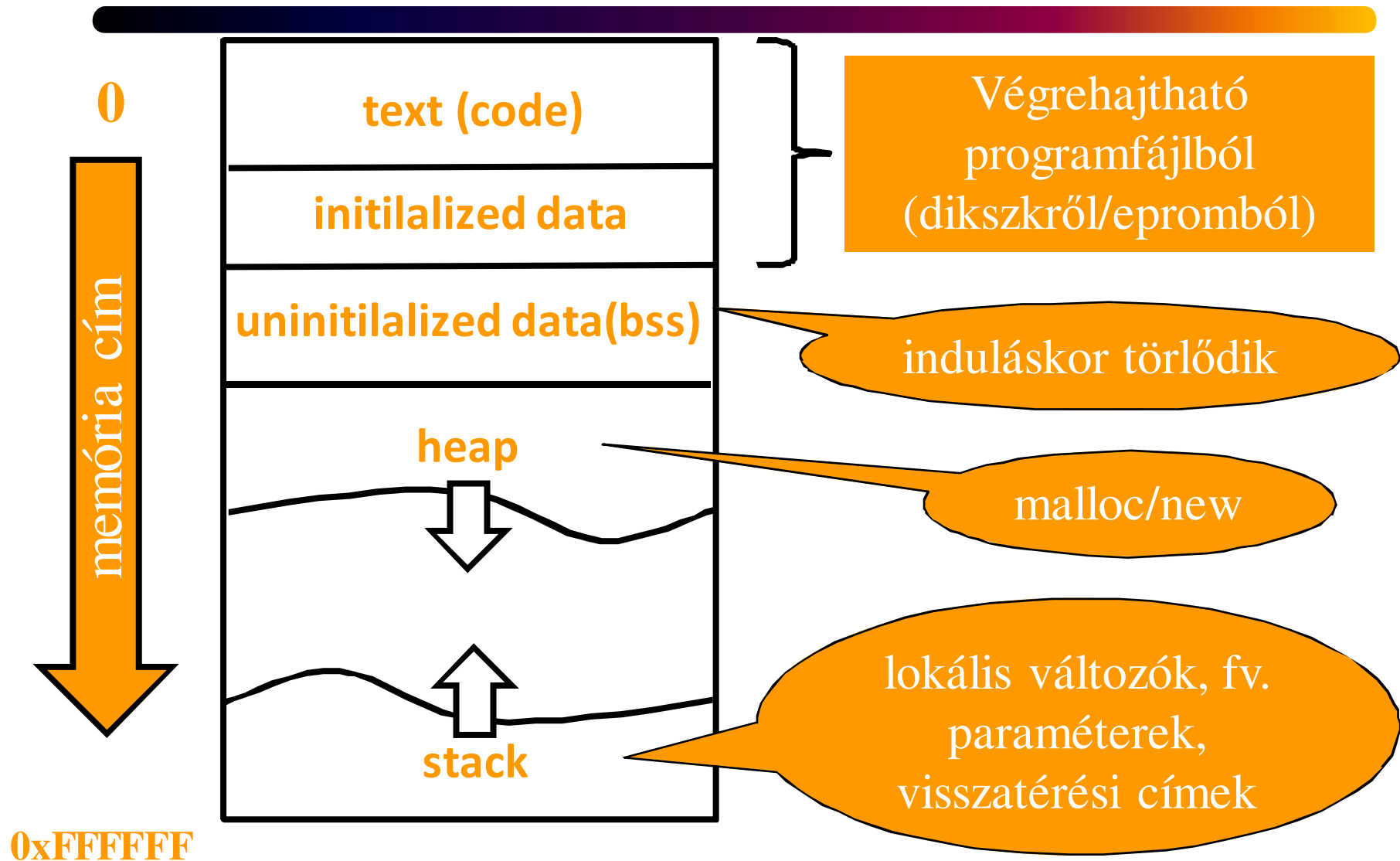
C++98: Csak konstruktorban

Csupán érdekesség (szorgalmi):

C++11-től: Az osztály deklarációjában is lehet. Ha mindkettőben van, akkor a deklarációnál megadott nem hajtódik végre.

```
struct B { B(int i = 0) {cout << i;} };  
struct A {  
    int a = 35;  
    B b0 = B();           // kiír: 0  
    B b1 = B();           // ez nem hajtódik végre  
    A() :b1(B(1)) {}// kiír: 1  
};
```

# Memóriakép (ism.)



# *Data szegmens*

---

- Text = a program (gépi) utasításai
  - modern környezetben írásvédett
- Data szegmens:
  - inicializált statikus és globális adat
  - konstansok (sztring) → RO data
- BSS (ASM direktíva 50-es évekből)
  - nem inicializált (0-val), statikus és globális adat



# Példa

```
cat maci.cc
```

```
char buf[100];           // 100 byte bss
char duma[] = "Hello !"; // 8 byte data
const char *pp = "C++";  // 8 byte data + 4 RO data
void f() {}              // ?? byte text
```

```
g++ -c maci.cc
```

```
size -A maci.o
```

section	size	addr
.text	6	0
.data	16	0
.bss	100	0
.rodata.str1.1	4	0
.comment	50	0



# Mi van a motorházban?

```
g++ -S maci.cc  
more maci.s
```

```
        .globl  buf  
        .bss  
buf:    .zero   100  
        .globl  дума  
        .data  
дума:  .string "Hello !"  
        .globl  pp  
        .section      .rodata  
.LC0:  .string "C++"  
        .data  
pp:    .quad   .LC0  
        .text  
        .type   _Z1fv, @function  
_Z1fv: pushq   %rbp  
      movq   %rsp, %rbp  
      popq   %rbp  
      ret
```



```
char buf[100];  
char дума[] = "Hello !";  
const char *pp = "C++";  
void f() {}
```

# *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 a méret: a) változat**
    - **igény szerint változtatja a méretét: b) változat**
  - Mindkét megoldás dinamikus memóriakezelést igényel. **Ügyelni kell a helyes felszabadításra, foglalásra.**

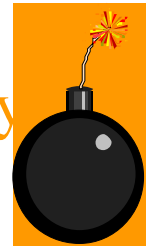
# KomplexTar osztály

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

## KomplexTar osztály/2

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

A memcopy nem hívná meg a konstruktort



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

Visszavezettük értékadásra

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

## a) Indexelés és a főprogram

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

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

## *b) Változó méretű KomplexTar*

*// Indexelés hatására növekszik a méret, ha kell*

```
Komplex& KomplexTar::operator[](unsigned int i)
{
    if (i >= db) { // növekednie kell, célszerű kvantumokban
        Komplex *tmp = new Komplex[i+10]; // legyen nagyobb
        for (size_t 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
}
// Konstans tároló nem tud növekedni
const Komplex& KomplexTar::operator[](size_t i) const {
    if (i >= db) throw Tar_Hiba(); return t[i];
}
```

## *c) Gyakorlatiasabb változat*

```
class KomplexTar {
    static const unsigned int nov = 3; // növekmény értéke
    Komplex *t;                       // pointer a dinamikusan foglalt adatra
    unsigned int db;                  // elemek száma
    unsigned int kap;                 // tömb kapacitása
public:
    KomplexTar(int m = 10) :db(m), kap(m+nov) {
        t = new Komplex[kap]; }
    KomplexTar(const KomplexTar&);
    unsigned int capacity() const { return kap; }
    unsigned int size() const { return db; }
    Komplex& operator[](unsigned int i);
    const Komplex& operator[](unsigned int i) const ;
    KomplexTar& operator=(const KomplexTar&);

```

...



## c) Gyakorlatiasabb változat /2

```
Komplex& KomplexTar::operator[](unsigned int i) {
    if (i >= kap) {
        Komplex *tmp = new Komplex[i+nov]; // legyen nagyobb
        for (size_t j = 0; j < db; j++) tmp[j] = t[j]; // átmásol
        delete[] t; // régi törlése
        t = tmp; // pointer az új területre
        kap = i + nov; // megnövelt kapacitás
    }
    if (i >= db) db = i+1; // megnövelt darab
    return t[i]; // referencia vissza
}
const Komplex& KomplexTar::operator[](unsigned int i) const {
    if (i >= db) throw Tar_Hiba(); return t[i];
}
```

[https://git.ik.bme.hu/Prog2/eloadas\\_peldak/ea\\_04/](https://git.ik.bme.hu/Prog2/eloadas_peldak/ea_04/)

# 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 konstr.
  - másoló: `X(const X&)` // referencia paramétere van,  
automatikusan létrejön: meghívja az adattagok másoló  
konstr.-át, 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

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

komplex Osztálylista

Az összes osztály, struktúra, unió és interfész listája rövid leírásokkal:

<b>Komplex</b>	<i>Komplex osztály</i>
<b>KomplexTar</b>	<i>KomplexTar osztály</i>

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

# 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

---

## 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); }  
friend istream& operator>>(istream& s, Komplex& k);  
friend ostream& operator<<(ostream& s, const Komplex k);
```

Paraméterek dokumentálása

Speciális kezdet

Rövid

abszolút érték

## Konstruktorok és destruktorkok dokumentációja

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

Konstruktor nulla, egy és két paraméterrel.

### Paraméterek:

*r* - valós rész (alapértelmezése 0)  
*i* - képzetes rész (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.