

Klases

- n Klase reprezentē objektu kopu un operācijas, kas ļauj manipulēt ar šiem objektiem, radīt un likvidēt tos. Atvasinātās klases manto bāzes klases locekļus.
- n Vienkāršota klases deklarācijas sintakse:

```
class vārds [: bāzes_klase]
{
    klases_locekļu_saraksts
};
```

- n Piemērs:

```
class Alpha
{
    private: int a, b;
    public: void set(int, int);
};
```

```
Alpha a1, m[10], *pa = &m[3];
int k, n[100], *p;
```

Klases (turpinājums)

Klases locekļu sarakstā var būt:

- datu deklarācijas;
- funkciju deklarācijas un definīcijas;

Funkcijām var būt vienādi vārdi, ja ir atšķirīgi parametru saraksti – notiek t. s. funkcijas pārlāde (*overloading*).

```
class Beta
{
    private:        int temp;
                   float delta;
    public:         void set(int, float);
                   void set(int);
};
```

Nav iespējams atšķirt parametru pēc vērtības un pēc atsauces!

```
void setVal( int  x );
void setVal( int& x );
```

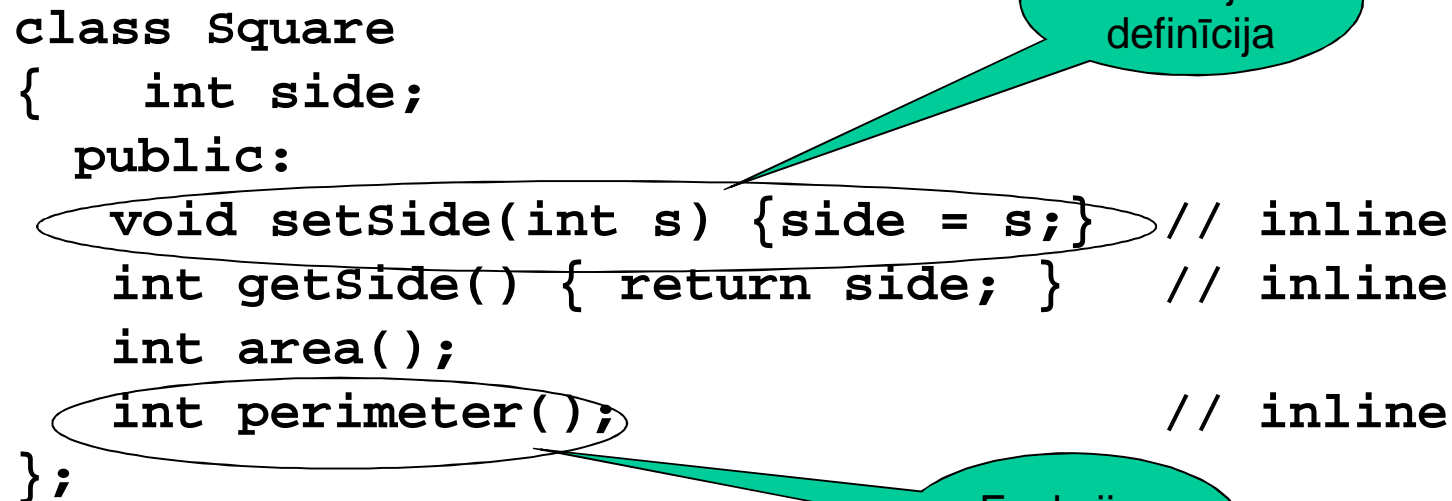
//setVal(z); - Kļūda! Nevar noteikt, kura funkcija jāizsauc.

Klases (turpinājums)

- n Ja funkcija ir definēta klases locekļu sarakstā, tad tā ir iebūvēta (*inline*) funkcija, kas kompilatoram iesaka, ka funkcijas izsaukums jāaizvieto ar tās definīciju.

Piemēram:

```
class Square
{
    int side;
    public:
    void setSide(int s) {side = s;} // inline
    int getSide() { return side; } // inline
    int area();
    int perimeter(); // inline
};
```



```
int Square::area()
{
    return side * side;
}
```

```
inline int Square::perimeter()
{
    return side * 4;
}
```

Klases locekļu pieejamība – iekapsulēšanas princips

n Klases locekļiem (mainīgajiem un funkcijām) iespējami trīs piekļuves veidi:

public – publisks

private – privāts (pēc noklusēšanas)

protected – aizsargāts

n Klases deklarācijas sintakse:

```
class klases_vārds [: bāzes_klase(s)]  
{  
    public:  
        publisko_klases_locekļu_saraksts  
    private:  
        privāto_klases_locekļu_saraksts  
    protected:  
        aizsargāto_klases_locekļu_saraksts  
};
```

Piemērs

```
class Triangle {  
    private:  
        int a, b, c;  
    public:  
        int setSides(int x, int y, int z);  
        void getSides(int *x, int *y, int *z);  
        float area();  
        int perimeter() { return (a + b + c); }  
};
```

Piemērs (turpinājums)

```
#include <math.h>
```

```
float Triangle::area()  
{ float p, s;  
  p = perimeter() / 2.0;  
  s = ( p*(p-a)*(p-b)*(p-c) );  
  if ( s >= 0 ) return( sqrt( s ) );  
  else return -1;  
}
```

```
int Triangle::setSides(int x, int y, int z)  
{ a = x; b = y; c = z;  
  if ( a + b < c || b + c < a || a + c < b ) return 0;  
  return 1;  
}
```

```
void Triangle:: getSides(int *x, int *y, int *z)  
{ *x = a;  
  *y = b;  
  *z = c;  
}
```

Nepilnā klases deklarācija

```
class klases_vārds;
```

n Ļauj atsaukties uz klases rādītāju, pirms klases pilnās deklarācijas.
Piemēram:

```
class Document; //nepilnā klases Document deklarācija
```

```
...
```

```
Document *d;
```

```
Document* docSearch(int key);
```

Klases konstruktori

- n Konstruktors ir speciāla klases funkcija, kas tiek automātiski izsaukta, kad tiek izveidots klases objekts
 - § Lokālajam objektam (auto) konstruktors tiek izsaukts, kad blokā izveido mainīgo
 - § Dinamiskiem objektiem, konstruktors tiek izsaukts, kad izpilda new.
 - § Globāliem un statiskiem mainīgajiem, konstruktors tiek izsaukts pirms funkcijas mai n() izsaukuma.

```
#include "triang.h" // Triangle deklarācijas iekļaušana
```

```
Triangle t1; // konstruktoru izsauc pirms funkcijas main()
```

```
void main()
```

```
{
```

```
    Triangle t2;           // konstruktora izsaukums
```

```
    Triangle* pt3;
```

```
    pt3 = new Triangle; // konstruktora izsaukums
```

```
    ...
```

```
    delete pt3;
```

```
}
```


Klases konstruktori (turpinājums)

- n Konstruktors ir klases funkcija, kuras vārds sakrīt ar klases vārdu
- n Konstruktoram nav atgriežamās vērtības tipa

```
class X
{
    int a;
    char b;

public:
    X();
    X(int);
    X(int, char);
    //X(X); // KĻŪDA!
    X(X&); // kopijas konstruktors

};
```

- n Kopijas konstruktoram vienīgais parametrs ir atsauce uz šīs klases objektu
- n Kopijas konstruktors rada dotā objekta kopiju

Klases konstruktori (turpinājums)

- n Konstruktori parasti tiek izmantoti, lai inicializētu klases mainīgos (atribūtus) un sagatavotu klases objektu

```
X::X()  
{  
    a = 0;  
    b = '\0';  
}
```

```
X::X(int n)  
{  
    a = n;  
    b = '\0';  
}
```

```
X::X(int a, char b)  
{  
    this->a = a;  
    this->b = b;  
}
```

```
X::X(X& x)  
{  
    a = x.a; //atļauta piekļuve klases privātajiem atribūtiem!  
    b = x.b; // -- " --  
}
```

Klases konstruktori (turpinājums)

- n Ja klasei nav deklarēts neviens konstruktors, tad kompilators automātiski ģenerē konstruktoru bez parametriem
- n Ja klasei nepieciešams kopijas konstruktors, bet tas nav deklarēts, tad kompilators tādu ģenerē pats

```
class Y
{
    int alpha;
public:
    Y();
    Y(int);
    Y(Y&);
};

class Z
{
    int beta;
    char gamma;
};
```

```
void main()
{
    Y a;          // Y()
    Y b(10);      // Y(int)
    Y c(a);       // Y(Y&)
    Y d = a;      // Y(Y&)
    Y e = 15;     // Y(int)
    Z k;          // Z()
    Z n = k;      // Z(Z&)
    Z m = 15;     // KĻŪDA!
}
```

Klases destruktori

- n Destruktors ir speciāla klases funkcija, kas tiek automātiski izsaukta, kad objekts tiek likvidēts.
 - § Klasei var būt tikai viens destruktors
 - § Destruktoram nevar būt parametri
 - § Destruktoram nav atgriežamās vērtības tipa
 - § Destruktora vārds sakrīt ar klases vārdu un tildes (~) simbolu tā sākumā.
- n Ja klasei nav deklarēts destruktors, tad kompilators to ģenerē pats.

```
class R
{
    int nn;
public:
    R(int); // konstruktors
    ~R();   // destruktors
};
```

Cik konstrukturu ir klasei R?

Klases destruktori (turpinājums)

n Destruktors tiek automātiski izsaukts:

- § Lokālajiem objektiem (auto) – kad programmas izpilde iziet no bloka, kurā objekti aprakstīti
- § Dinamiskiem objektiem – kad izpilda **delete**
- § Globāliem un statiskiem mainīgajiem – pēc funkcijas main() beigām

Konstruktori un destruktori funkcijās

```
#include <iostream.h>
class A
{   private: int a;
    public:

    A()
    { a = 0;
      cout << "Default constructor";
    }

    A(int v)
    { a = v;
      cout << "INT constructor";
    }

    A(const A& aa)
    { a = aa.a;
      cout << "Copy constructor";
    }

    ~A()
    { cout << "Destructor"; }

    int getA() { return a; }
};
```

```
void printA(A o) //parametrs ir objekts
{
    cout << o.getA() << endl;
}
```

```
void main()
{
    cout << "main begin" << endl;
    A f(4);
    printA(f);
    cout << "main end" << endl;
}
```

Programmas darbības rezultāts:

```
main begin
INT constructor
Copy constructor
4
Destructor
main end
Destructor
```

Konstruktori un destruktori funkcijās

```
#include <iostream.h>
class A
{   private: int a;
    public:
    A()
    { a = 0;
      cout << "Default constructor";
    }
    A(int v)
    { a = v;
      cout << "INT constructor";
    }
    A(const A& aa)
    { a = aa.a;
      cout << "Copy constructor";
    }
    ~A()
    { cout << "Destructor";
    }

    int getA() { return a; }
};
```

```
void printA(A& o) //parametrs ir atsauce
{
    cout << o.getA() << endl;
}
```

```
void main()
{
    cout << "main begin" << endl;
    A f(4);
    printA(f);
    cout << "main end" << endl;
}
```

Programmas darbības rezultāts:

```
main begin
INT constructor
4
main end
Destructor
```

Vienkārša simbolu virknes klase

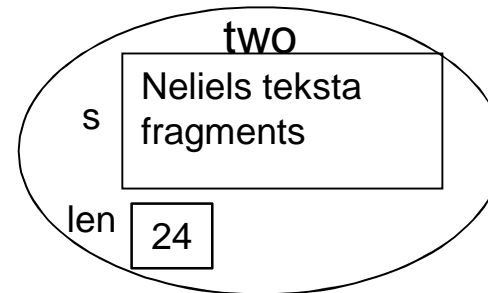
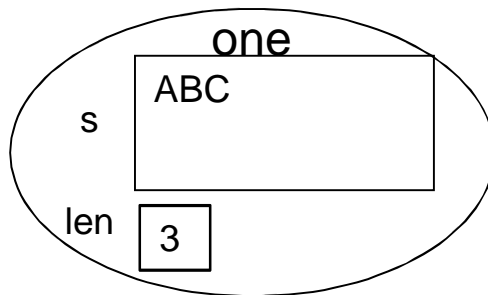
```
class MyString{
private:
    char s[256];
    int len;
public:
    void assign(char *str);
    int length() {return len;};
    void print();
};

// Assign string value
void MyString::assign(char *str)
{
    strcpy(s, str);           //copy string
    len = strlen(str); //assign length
}

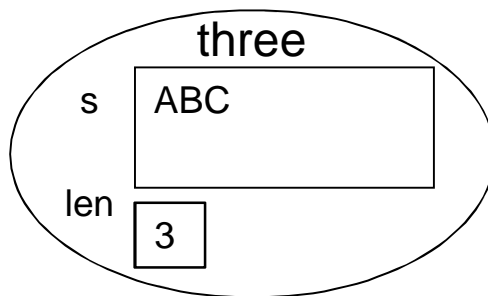
// Output of string value with newline
void MyString::print()
{
    cout << s << "\n";
}
```


Vienkārša simbolu virknes klase

```
MyString one, two; // Izsauc konstruktoru MyString() 2 reizes  
one.assign("ABC");  
two.assign("Neliels teksta fragments");
```



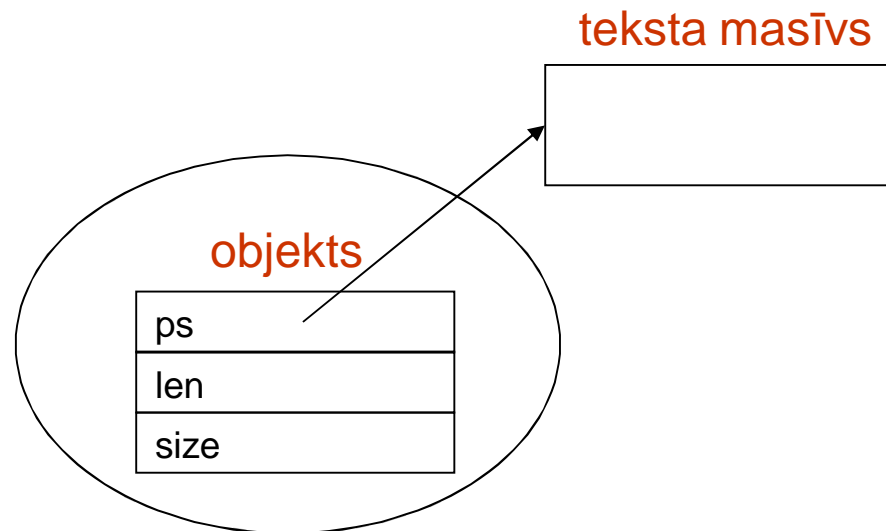
```
MyString three = one; // Kopijas konstruktors
```



`sizeof(three)` vērtība ir 260

Simbolu virknes klase ar konstruktoriem un destruktoru

```
class MyString{
private:
    char *ps;
    int size;
    int len;
public:
    MyString();
    MyString(int maxLength);
    MyString(char *str);
    ~MyString();
    void assign(char *str);
    int length() {return len;};
    void print();
};
```



Simbolu virknes klase ar konstruktoriem un destruktoru

```
MyString::MyString()  
{   ps = new char[256];  
    size = 256;  
    len = 0;  
}  
MyString::MyString( int maxLength )  
{   if (maxLength < 1) {  
        cout << "Illegal string size : " << maxLength;  
        exit(0);}  
    ps = new char[maxLength + 1];  
    size = maxLength + 1;  
    len = 0;  
}  
MyString::MyString( char *str )  
{   len = strlen( str );  
    ps = new char[len+1];  
    strcpy( ps, str );  
    size = len+1;  
}  
MyString::~MyString()  
{   delete ps; }
```

Simbolu virknes klase ar konstruktorem un destruktoru

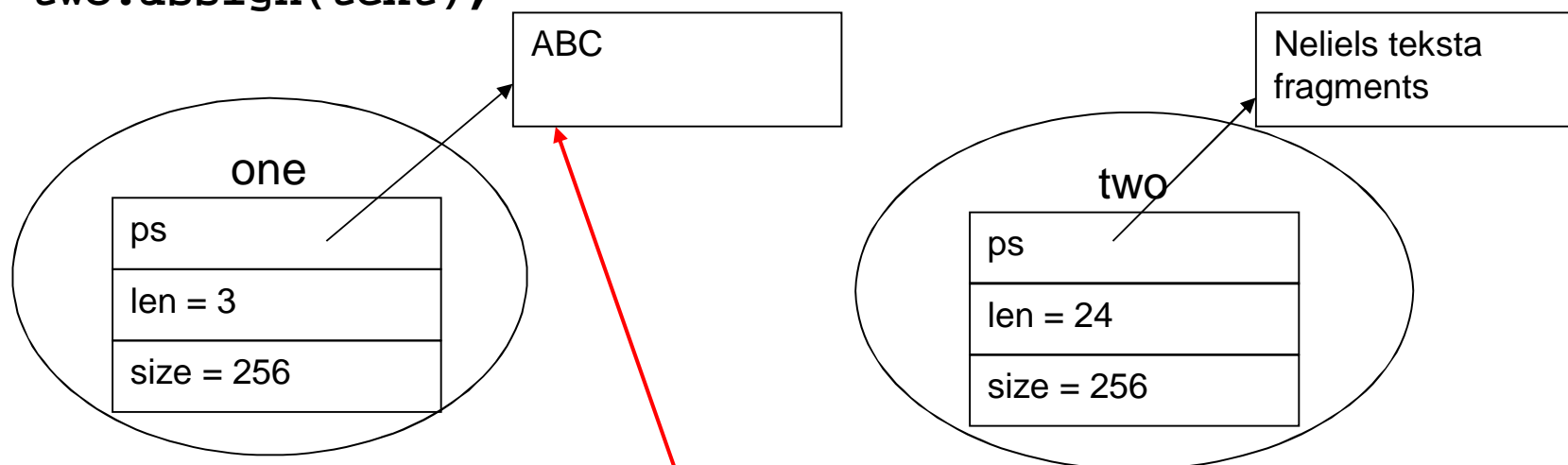
```
// Test MyString class
void main()
{
    char text[] = "Neliels teksta fragments";
    MyString a, b(10), c("ABC");

    a.assign("GAMMA");
    b.assign(text);
    a.print();
    b.print();
    c.print();

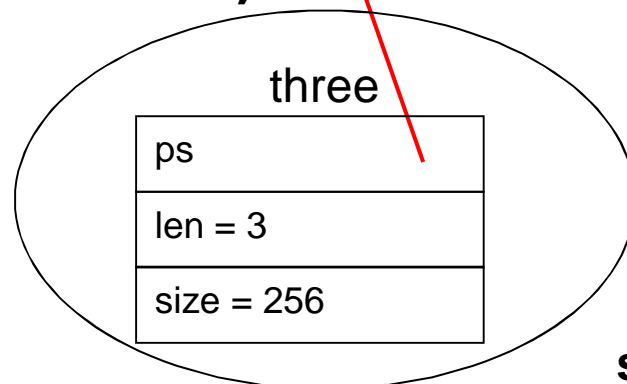
    MyString *q;
    q = new MyString(25);
    q->assign("DELTA");
    q->print();
}
```

Simbolu virknes klase ar konstruktoriem un destruktoru - PROBLĒMAS

```
MyString one, two;  
one.assign("ABC");  
two.assign(text);
```



```
MyString three = one;
```



Gan likvidējot objektu *one*, gan likvidējot objektu *three*, destruktorā izpildīs *delete* vienai un tai pašai atmiņai!

sizeof(three) vērtība ir 12

Simbolu virknes klase ar konstruktoriem un destruktoru

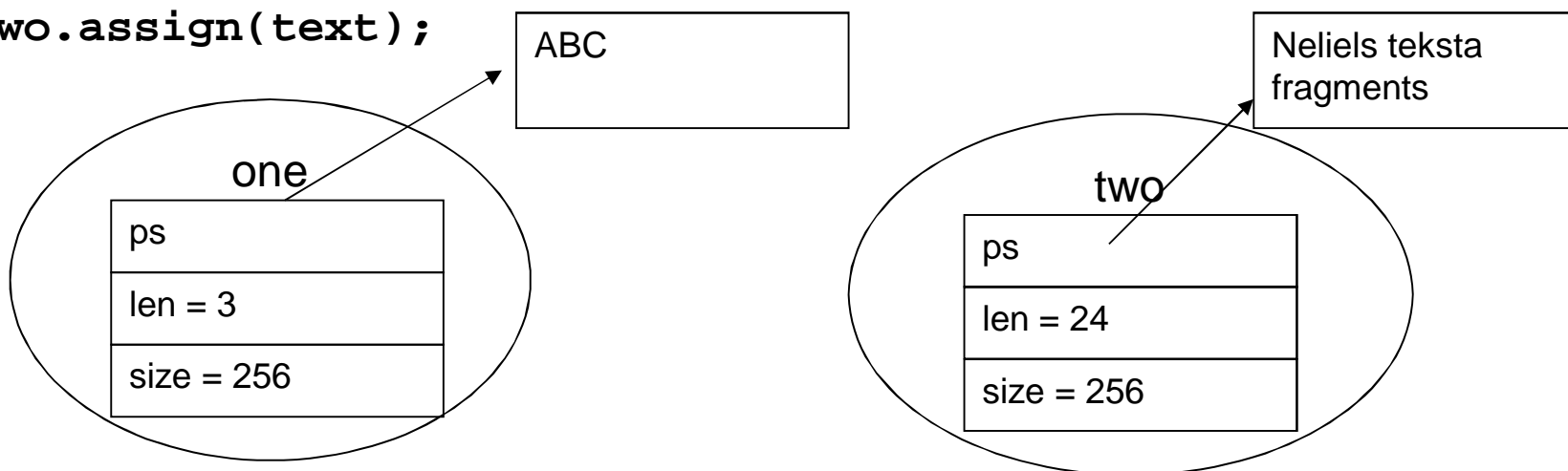
n Risinājums – jāraksta savs kopijas konstruktors:

```
class MyString {
private:
    char* ps;
    int len;
public:
    ...
    MyString(const MyString&);
    ...
};

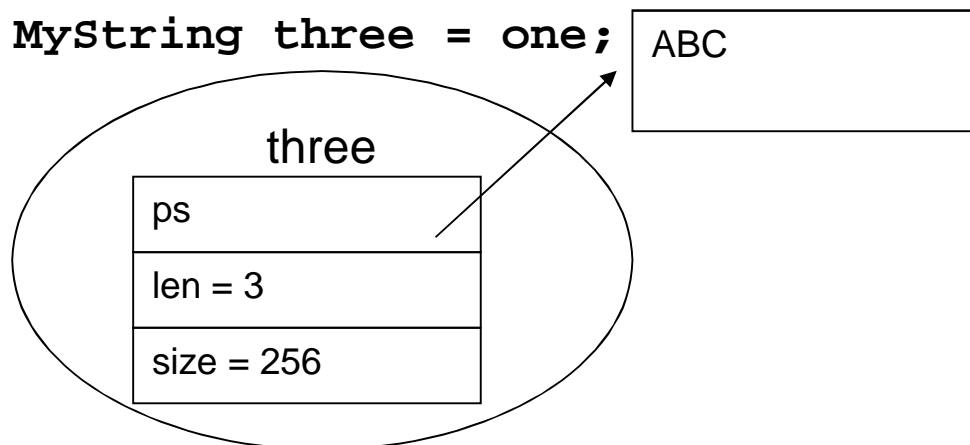
MyString:: MyString(const MyString& s)
{
    len = s.len;
    size = s.size;
    ps = new char [size];
    for (int i=0; i<=len; ++i) ps[i] = s.ps[i];
}
```

Simbolu virknes klase ar konstruktorem un destruktoru

```
MyString one, two;  
one.assign("ABC");  
two.assign(text);
```

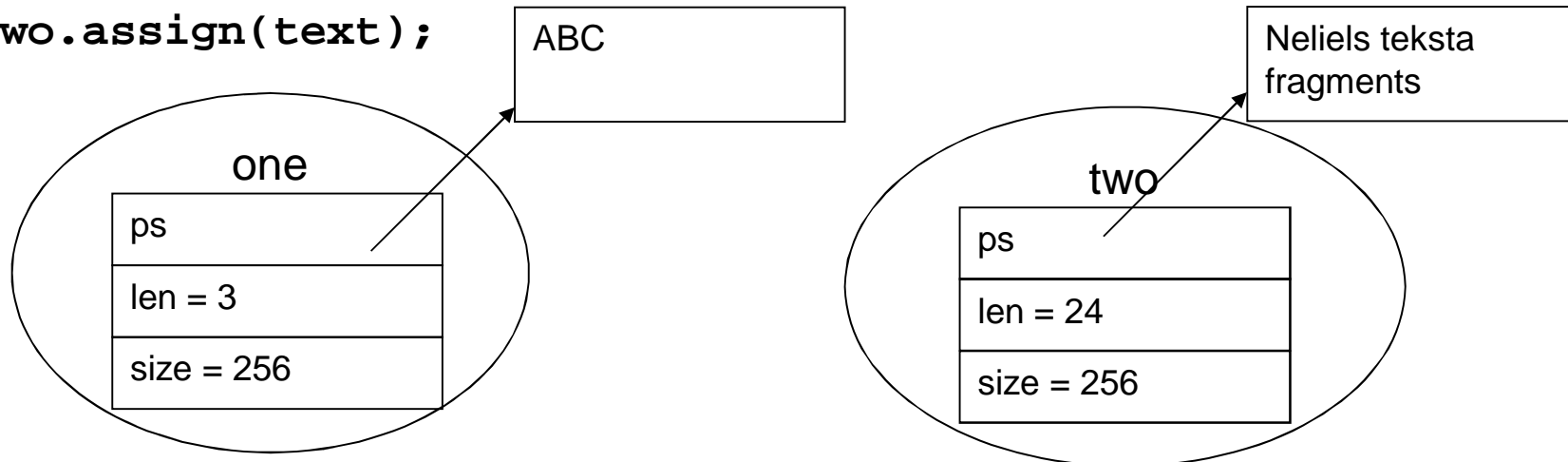


```
MyString three = one;
```

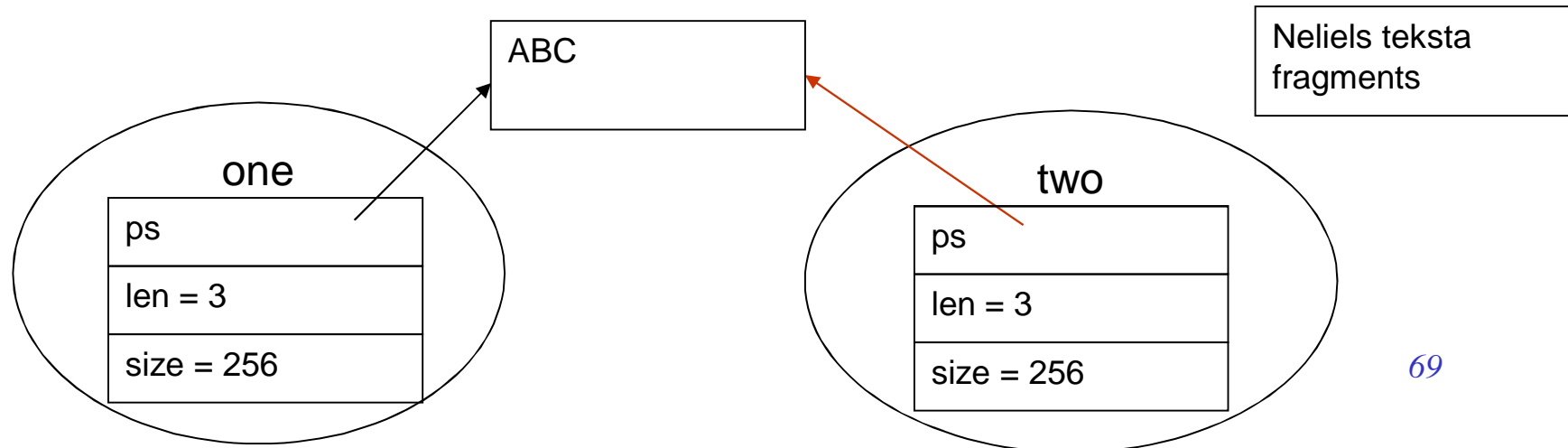


Simbolu virknes klase ar konstruktoriem un destruktoru

```
MyString one, two;  
one.assign("ABC");  
two.assign(text);
```



```
two = one;    // piešķire!
```



Simbolu virknes klase ar konstruktoriem un destruktoru

n Risinājums – jāraksta sava piešķires operācija:

```
class MyString {  
private:  
    char* ps;  
    int len;  
public:  
    ...  
    MyString(const MyString&);  
    MyString& operator=(const MyString&);  
    ...  
};
```

```
MyString& MyString::operator=(const MyString& s)  
{  
    len = s.len; size = len+1;  
    if (ps) delete[] ps;  
    ps = new char [size];  
    for (int i=0; i<=len; ++i) ps[i] = s.ps[i];  
    return *this;  
}
```

Konstruktori un destruktori

n Secinājums

Ja starp klases atribūtiem ir rādītāji uz dinamiski iedalītiem atmiņas apgabaliem, tad jāprogrammē savs:

- § kopijas konstruktors
- § piešķires operators
- § destruktors