ФАКУЛТЕТ ЗА ЕЛЕКТРОТЕХНИКА И ИНФОРМАЦИСКИ ТЕХНОЛОГИИ

Преоптоварување на оператори

– Податочни структури и програмирање–

Пример од претходно

```
#include <iostream>
#include <cstring>
using namespace std;
class String
{
private:
        char * cptr;
public:
        String(const char * = "");
        ~String();
        void Show(void) const { cout <<
cptr; }
};
inline String::String(const char * a_cptr)
{
        cptr = new char[strlen(a_cptr) + 1];
        strcpy(cptr, a_cptr);
}
inline String::~String()
{
        delete[] cptr;
}
```

покажувач кон динамички алоцирана меморија во која ќе се чува стрингот

```
int main()
{
        const String b("World");
        String c, d("Hello");
        {
                String a(d);
                a.Show(); cout << ' ';
                b.Show(); cout << endl;
                c = a; d = b;
                c.Show(); cout << ' ';
                d.Show(); cout << endl;
        }
        c.Show(); cout << ' ';
        d.Show(); cout << endl;
        return(0);
}
```

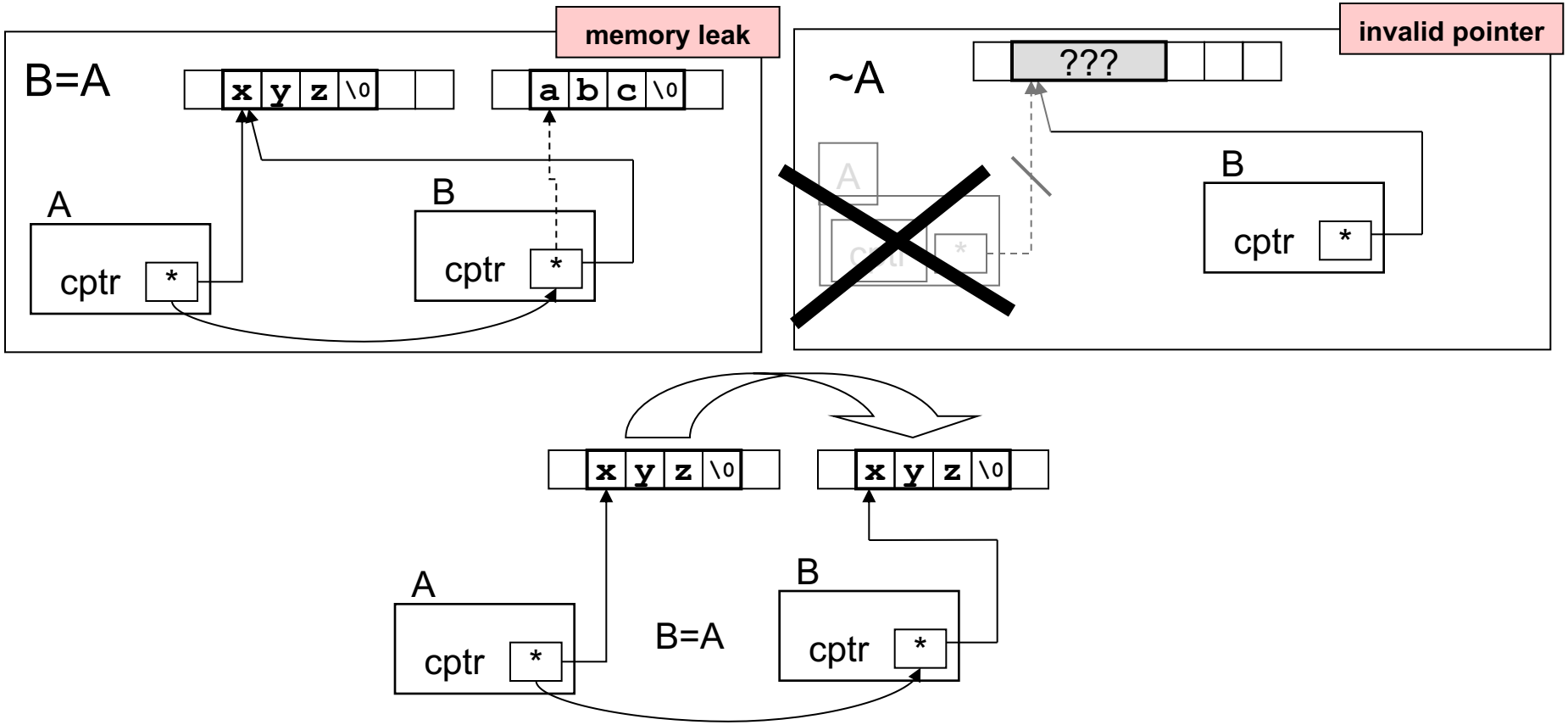
алоцирање на потребната меморија

копирање на низата знаци во неа

```
Hello World
Hello World
 World ???
```

Што се случува?

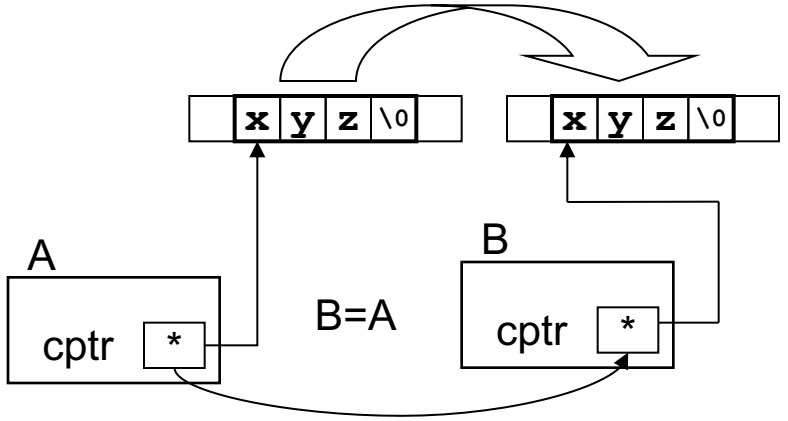
```
String A("xyz"), B("abc");
B = A; // ???
```



Преоптоварување на операторот за доделување = (assignment operator)

```
class String {
        . . .
public:
        String& operator=(const String &);

        . . .
};
String& String::operator =(const String &from)
 {
        delete[] cptr;
        cptr = new char[strlen(from.cptr) + 1];
        strcpy(cptr, from.cptr);
        return *this;
}
```



Но што ако: `A=A;`
или не толку очигледно:
`String *p=&A;`
`A=*p;`

this покажувач

```
#include <iostream>
using namespace std;

class point {
public: // place public members first
          void print() const { cout << "(" << x << "," << y << ")"; }
          void set(double u, double v) { x = u; y = v; }
          void plus(point c);
          point inverse() { x = -x; y = -y; return (*this); }
          point* where_am_I() { return this; }
private:
          double x, y;
};

void point::plus(point c) { x += c.x; y += c.y; }
int main() {
          point a, b;
          a.set(1.5, -2.5);
          a.print();
          cout << "\na is at " << a.where_am_I() << endl;
          b = a.inverse();
          b.print();
          cout << "\nb is at " << b.where_am_I() << endl;
}
```

Покажувач кој покажува на членовите на објект од класата

```
(1.5,-2.5)
a is at 0x0064fdd4
(-1.5,2.5)
b is at 0x0064fdc4
```

Copy конструктор - дополнување

```
class String
{
public:
        . . .
        String(const String &); //copy
        String &operator=(const String &);
};
String& String::operator =(const String &from)
{
        if (this == &from)
                return *this;
        else
        {
           delete[] cptr;
           cptr = new char[strlen(from.cptr) + 1];
           strcpy(cptr, from.cptr);
           return *this;
        }
}
inline String::String(const String &from)
{
        cptr = new char[strlen(from.cptr) + 1];
        strcpy(cptr, from.cptr);
}
```

за да се избегне самододелување

```
int main(){
const String b("World");
String c, d("Hello");
{
   const String a(d), *p = &d;
   a.Show(); cout << ' ';
   b.Show(); cout << endl;
   c = "HELLO"; // ke raboti
   d = *p;
   c.Show(); cout << ' ';
   d.Show(); cout << endl;
}
c.Show(); cout << ' ';
d.Show(); cout << endl;
return(0);
}
```

```
Hello World
HELLO Hello
HELLO Hello
```

Преоптоварување на оператор = (1)

- Ако се дефинира **оператор** = да **враќа** **void**, би работело **еднократно доделување** (x=y)

```
class Rational {
public:
        Rational(int n = 0, int d = 1) : num(n), den(d) {};       // default constructor
        Rational(const Rational& r) : num(r.num), den(r.den) {};  // copy constructor
        void operator=(const Rational&);                          // assignment operator
        // other declarations go here
private:
        int num;
        int den;
};
void Rational::operator=(const Rational& r) {
        num = r.num;
        den = r.den;
};
```

Може да се употреби на следниот начин

```
Rational z, x, y(3, 5);
x = y;
```

Но не и вака:

```
z = x = y;
```

Преоптоварување на оператор = (2)

- За **повеќекратно доделување** (x=y=…=z) потребно е да се дефинира **операторот да враќа `Rational&`**

```
class Rational {
public:
        Rational(int n = 0, int d = 1) : num(n), den(d) {};       // default constructor
        Rational(const Rational& r) : num(r.num), den(r.den) {};  // copy constructor
        Rational& operator=(const Rational&);                     // assignment operator
        // other declarations go here
private:
        int num;
        int den;
};
Rational& Rational::operator=(const Rational& r) {
        num = r.num;
        den = r.den;
        return *this;
};
int main() { Rational x, y, z; x=y=z=3.14; }
```

Може да се употреби на следниот начин

```
Rational z, x, y(3, 5);
z = x = y;
```

Преоптоварување на оператор *=

```
class Rational {
public:
        Rational(int = 0, int = 1);
        Rational& operator=(const Rational&);
        Rational& operator*=(const Rational&);
        // other declarations go here
private:
        int num, den;
        // other declarations go here
};
Rational& Rational::operator*=(const Rational& r)
{
        num = num*r.num;
        den = den*r.den;
        return *this;
}
```

- Може да се употреби на следниот начин

```
Rational z, x(1, 2), y(3, 5);
x *= y;
z *= x *= y
```

Преоптоварување на релациски оператори (1)

```
#include <iostream>
using namespace std;

class Rational {
public:
        Rational(int n = 0, int d = 1) : num(n), den(d) {};
        Rational(const Rational& r) : num(r.num), den(r.den) {};
        Rational& operator=(const Rational&);
        Rational& operator*=(const Rational&);
        bool operator==(const Rational&);
        void Show() const { cout << num << '/' << den << endl; }
private:
        int num, den;
};
```

Преоптоварување на релациски оператори (2)

```
Rational& Rational::operator=(const Rational& r) {
        num = r.num; den = r.den; return *this;
}

Rational& Rational::operator*=(const Rational& r) {
        num *= r.num; den *= r.den; return *this;
}

bool Rational::operator==(const Rational &t) {
        return(num * t.den == den * t.num);
}
```

Еднаквост на дропки

Сите бинарни оператори членови на класа имаат еден аргумент и се повикуваат како: a==b ↔ a.operator==(b)

```
int main() {
        Rational x(2, 7), y, z;
        y = 2;
        y *= x;
        x *= 7; //OK
        x.Show();
        y.Show();
        cout << ((x == y) ? "isti se" : "ne se isti") << endl;
        cout << ((x == 2) ? "isti se" : "ne se isti") << endl; // OK
//      cout << ((2==y)? "isti se" : "ne se isti") << endl; // Ne moze!!!
}
```

Преоптоварување на релациски оператори (3)

strcmp() – враќа 0 ако се исти

```
class String {
        . . .
public:
        bool operator==(const String &a) const { return !strcmp(cptr, a.cptr); }
        bool operator!=(const String &a) const { return strcmp(cptr, a.cptr); }
        . . .
};
int main() {
        const String b("World");
        String c, d("Hello");
        const String a(d), *p = &d;
        c = "HELLO"; d = *p;
        a.Show(); cout << " " << (a == b ? "==" : "=/=") << " "; b.Show(); cout << endl;
        c.Show(); cout << " " << (c != d ? "!=" : "ne e !=") << " "; d.Show(); cout << endl;
        if (c == "HELLO") // moze
        {
                c.Show(); cout << "==HELLO" << endl;
        }
        // if("HELLO"==c) // vaka NE moze!!!
        return(0);
}
```

Преоптоварување на релациските оператори (4)

- **Со вака дефиниран оператор** `==` (како член на класата) споредувањата **`a==b` се преведуваат како `a.operator==(b)`**
- Тоа значи дека **од левата страна секогаш мора да се најде објект од класата за која е преоптоварен операторот**
- За да се овозможи **споредување со друг тип објект** (како `char*`: `"HELLO"==c`) операторот `==` **треба да се преоптовари за конкретниот пар објекти**
- За истиот да може да пристапи до приватниот дел од класата `String` мора да биде **прогласен и за нејзин пријател**

```
class String {
public:
. . .
        friend bool operator==(const char *a, const String &b)
        { return!strcmp(a, b.cptr); }
. . .
};
```

Преоптоварување на cast операторот и << (1)

Со преоптоварувањето на cast операторот **char*,** се овозможува објект од класата String имплицитно (или експлицитно) да се претвори во char* и да се проследи во функции или изрази кои примаат char*.

```
class String
{
...
public:
        operator char * () { return cptr; }
        friend ostream &operator<<(ostream &output, const String & a)
                { return output << a.cptr; }
        friend bool operator==(const String &a, const String &b)
                { return !strcmp(a.cptr, b.cptr); }
        friend bool operator==(const char *a, const String &b)
                { return !strcmp(a, b.cptr); }
        friend bool operator==(const String &a, const char *b)
                { return !strcmp(a.cptr, b); }
};
int main() {
        String A("Hello");
        cout << "Dolzinata na \"" << A << "\" e " << strlen(A) << endl;
        return(0);
}
```

```
Dolzinata na "Hello" e 5
```

Преоптоварување на `cast` операторот и `<<` (2)

- **По воведувањето на cast операторот `char*`, нема да работи споредувањето на `String` со `char*` -** за оператор `==` преоптоварен како член на класата
  - Error: two overloads have same function
- Тогаш, за да се пресмета изразот (`c=="HELLO"`) ќе постојат **две алтернативи**:
  - `"HELLO"` да се претвори во `String` и потоа да се споредуваат стрингови со преоптовареният оператор `==` од класата `String` или
  - `c` да се претвори во `char*` со дефинираниот `cast` оператор и потоа да се споредуваат покажувачи со вградениот оператор `==`
- Затоа **мора експлицитно** да се преоптовари споредбата на `char * ` со `String` како во примерот

Преоптоварување на операторите + и += за конкатенација (1)

```
String A("Hello"), B("World"), C;
C = A + B;
A += B;
```

```
C = A + ” Hi";
```

```
class String
{
    ...
    public:
    friend String operator+(const String& a, const String& b){
          String tmp;
          delete[] tmp.cptr;
          tmp.cptr = new char[strlen(a.cptr) + strlen(b.cptr) + 1];
          strcpy(tmp.cptr, a.cptr);
          strcat(tmp.cptr, b.cptr);
          return tmp;
    }
    friend String operator +(const String & a, const char * b); . . .
    friend String operator +(const char * a, const String & b); . . .
    String& operator +=(const String &from) {
          char *tmp;
          tmp = new char[strlen(cptr) + strlen(from.cptr) + 1];
          strcpy(tmp, cptr);
          strcat(tmp, from.cptr);
          delete[] cptr;
          cptr = tmp;
          return *this;
    }
}
```

Операторите +=, -=, ... мораат да бидат членови на класата и да враќаат референца кон објектот.

Преоптоварување на операторите + и += за конкатенација (2)

```
int main()
{
        String A("Hello"), B("World"), C, D(A);
        C = A + B;
        cout << C << endl;
        C = A + " " + B;
        cout << C << endl;
        D += " ";
        D += A;
        cout << D << endl;
        return(0);
}
```

```
HelloWorld
Hello World
Hello Hello
```

Преоптоварување на операторот [ ]

```
int main() {
        String a("Hello");
        const String b("World");
        String c, d;
        cout << a[0] << endl; // OK
        a[0] = 'W'; // OK
//      cout << b[0] << endl; // Ova ne raboti, a treba
//      b[0]='Z'; // Ova ne smee da raboti : b e const
        cout << "a=" << a << endl;
        cout << "b=" << b << endl;
}
```

Потребна е нова верзија на операторот [ ] за константни објекти!

```
class String
{
    ...
public:
        char &operator [](int i)
        { return cptr[i]; } // за неконстантни објекти
        const char &operator [](int i) const
        { return cptr[i]; } // за константни објекти
};
```

```
H
a=Wello
b=World
```

По ова и `cout << b[0] << endl;` ќе работи, но не и `b[0]='Z';`

Преоптоварување на операторите ++ и -- (1)

```
class Par {
private:
        int a, b;
public:
        explicit Par(int x = 0) :a(x), b(x) {}
        Par(int x, int y) :a(x), b(y) {}
        int prv() const { return a; }
        int vtor() const { return b; }
        int prv(int x) { return a = x; }
        int vtor(int x) { return b = x; }
        Par operator+(const Par &x) const { return Par(a + x.a, b + x.b); }
        Par operator-(const Par &x) const { return Par(a - x.a, b - x.b); }
        Par &operator+=(const Par &x) { a += x.a; b += x.b; return *this; }
        Par &operator-=(const Par &x) { a -= x.a; b -= x.b; return *this; }
        bool operator==(const Par &x) const { return(a == x.a && b == x.b); }
        bool operator!=(const Par &x) const { return !(*this == x); }
        Par operator-() const { return Par(-a, -b); } // unaren minus
        Par &operator++() { a++; b++; return *this; } //Prefix increment operator (++x)
        Par &operator--() { a--; b--; return *this; }  //Prefix decrement operator (--x)
        Par operator++(int) { Par r(*this); a++; b++; return r; } //Postfix increment (x++)
        Par operator--(int) { Par r(*this); a--; b--; return r; } //Postfix decrement (x--)
        friend ostream &operator<<(ostream &o, const Par &x)
        { return o << '(' << x.a << ',' << x.b << ')'; }
        friend istream &operator>>(istream &i, Par &x)
        { return i >> x.a >> x.b; }
};
```

потсетување

```
b = 10;
a = b++ ;     // a = 10 , b = 11

b = 10;
a = ++b;      //   a = 11 ,  b  = 11
```

Преоптоварување на операторите ++ и – (2)

```
int main() {
        Par x(1, 2), y(3), z;
        cout << x << " " << y << " " << z << endl;
        z.prv(x.prv());  cout << z << endl;
        cin >> y;  z = x + y - Par(3, 1); z += -x;
        // z+=1 nema da raboti, poradi explicit
        cout << x << z << endl;
        x = --z; cout << x << z << endl;
        x = z++; cout << x << z << endl;//z.operator++(0);
        return(0);
}
```

explicit keyword prevents the compiler from using that constructor for implicit conversions (z(1))

```
(1,2) (3,3) (0,0)
(1,0)
5 6 ◄┘
(1,2)(2,5)
(1,4)(1,4)
(1,4)(2,5)
```