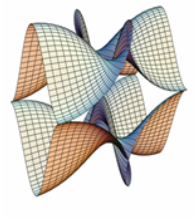

Prirodoslovno-matematički fakultet
Matematički odsjek
Sveučilište u Zagrebu

RAČUNARSKI PRAKTIKUM I

Vježbe 10 - friend, operatori

v2018/2019.

Sastavio: Zvonimir Bujanović

friend

Ako želimo da neka funkcija (koja nije članica) ili klasa ima pristup `private` ili `protected` elementima klase koju kreiramo, možemo joj to dopustiti tako da ju navedemo kao prijateljsku u definiciji klase.

```
class stack
{
    int element[100], size;
    ...

    friend void ispisi( const stack &s );
};

void ispisi( const stack &s )
{
    for( int i=0; i < s.size; ++i )
        cout << s.element[i] << " ";
    cout << endl;
}
```

friend

Kao prijateljska se može navesti cijela druga klasa.

- Sve funkcije klase `stack_manager` imaju pristup privatnim članovima klase `stack`:

```
class stack {
    ...
    friend class stack_manager;
};
```

Možemo označiti i samo jednu funkciju druge klase kao prijateljsku.

- Samo funkcija `velicina` klase `stack_manager` ima pristup privatnim članovima klase `stack`:

```
class stack {
    ...
    friend void stack_manager::velicina();
};
```

Operatori: ponavljanje

Odredite vrijednost ili komentirajte donje izraze.

```
int a = 2, b = 3, c = 4, d = 5;

if( a = b ) ...

a = b = c+a;
a += b += c;

if( a == b && c == d ) ...
if( a == b & c == d ) ...

a = b & c;

a << 2+3;
2+1 << 5
a << 1 << 2

a = 2, 3;
return 5, ++brojac;
```

Operatori: prioriteti

1. najveći prioritet ima `::`
2. `.`, `->`, `[]`, `()`, `++` i `--` (postfix). Asocijativni su slijeva nadesno.
3. `sizeof`, `++` i `--` (prefix), `~`, `!`, unarni `-` i `+`, `&` (referenciranje), `*` (dereferenciranje), `new`, `delete`, `()` – castanje. Asocijativni su zdesna nalijevo.
4. `*` (množenje), `/` i `%`
5. `+` (zbrajanje) i `-` (oduzimanje)
6. `<<` i `>>`
7. `<`, `>`, `<=` i `>=`
8. `==` i `!=`
9. `&` - bitovni i
10. `^` - bitovni isključivi ili
11. `|` - bitovni ili
12. `&&` - logički i
13. `||` - logički ili
14. `uvjet?izraz1:izraz2` - asocijativan zdesna nalijevo.
15. `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `|=`, `^=` - asoc. zdesna nalijevo.
16. najmanji prioritet ima `,`

Nadogradnja *(overloading)* operatora

Promotrimo sljedeću klasu:

```
class Razlomak {
    int brojnik, nazivnik;
    void skratiMe() { ... };

public:
    Razlomak( int b=0, int n=1 ) : brojnik(b), nazivnik(n)
    {
        skratiMe();
    }
};
```

Cilj je omogućiti ovakvu funkcionalnost:

```
Razlomak A( 2 ), B( 2, 7 ), C, D;

C = A + B; D = A * C; D *= B;
cout << C; // treba ispisati 16/7
```

Nadogradnja *(overloading)* operatora

Većinu operatora možemo nadograditi tako da znaju raditi s našim klasama. Ne mogu se nadograditi samo `.`, `.*`, `::`, `?:`, `#` i `##`.

Operatori su funkcije s imenom `operatorX`, gdje je `X` simbol operatora.

```
class Razlomak {
    int brojnik, nazivnik;
    void skratiMe() { ... };

public:
    Razlomak( int b=0, int n=1 ) { ... }
    friend Razlomak operator*( const Razlomak &a, const Razlomak &b );
};

Razlomak operator*( const Razlomak &a, const Razlomak &b )
{
    Razlomak rez;
    rez.brojnik = a.brojnik * b.brojnik;
    rez.nazivnik = a.nazivnik * b.nazivnik;
    rez.skratiMe();
```

Nadogradnja *(overloading)* operatora

Ovakav `operator*` se poziva u sljedećim situacijama:

```
Razlomak A(1, 2), B(1, 4), C, D;

// Oba parametra su tipa Razlomak.
C = A * B;

// Prvi ili drugi parametar je tipa int.
D = A * 3;
```

Kad bi postojao `operator*( const Razlomak &A, int x )`, u drugom primjeru bi se pozivao on.

- Kod nas ne postoji, pa se parametri pokušavaju konvertirati do nekog operatora koji postoji.
- `Razlomak` se ne može konvertirati u `int`.
- Broj 3 se može konvertirati u `Razlomak` jer `Razlomak` ima konstruktor koji prima `int`.
- Efekt je sljedeći: `D = A * Razlomak(3);`

Zadatak 1

- Nadogradite operatore zbrajanja, oduzimanja te unarnog minusa tako da funkcioniraju s parametrima tipa `Razlomak`.
- Obratite pažnju na skraćivanje razlomaka.
- Razlomke spremajte tako da im nazivnik nikada ne postane negativan.

Operatori kao članovi klase

Operatore možemo implementirati i tako da budu članovi klasa.
Tada se `this` podrazumijeva kao lijevi argument operatora.

```
class Razlomak {
    ...
    Razlomak operator*( const Razlomak &b ) const
    {
        Razlomak rez;
        rez.brojnik = brojnik * b.brojnik;
        rez.nazivnik = nazivnik * b.nazivnik;
        rez.skratiMe();
        return rez;
    }

    Razlomak operator-() const // unarni minus!
    {
        return Razlomak( -brojnik, nazivnik );
    }
};
```

Operatori kao članovi klase

Ako su operatori implementirani kao članovi klasa, onda se pozivaju u ovim situacijama:

```
Razlomak A(1, 2), B(1, 4), C, D, E, F;

// Isto kao: C = A.operator*( B );
C = A * B;
C = A.operator*(B); // ovo mozemo i eksplicitno napisati

// Isto kao: D = A.operator*( Razlomak(3) );
D = A * 3;

// Ovo NE radi (compile error)!!!
// Poziv E = 3.operator*( A ) nije ispravan.
E = 3 * A;

// Isto kao: F = A.operator-();
F = -A;
F = A.operator-(); // OK
```

Operatori kao članovi klase

Operatore `=`, `[]`, `()`, `->` možemo implementirati samo kao članove.

Obično tako implementiramo i sve operatore koji imaju pridruživanje: `+=`, `-=`, `*=`, ... Takvi operatori obično vraćaju referencu na `this` kako bi se mogli ulančavati.

```
class Razlomak {
    ...
    Razlomak &operator*=( const Razlomak &b ) // nije const!
    {
        brojnik *= b.brojnik;
        nazivnik *= b.nazivnik;
        skratiMe();
        return *this;
    }
};

Razlomak A(1, 2), B(1, 4), C(1, 3);
A *= B *= C; // Ulancavanje, efekt je: B = B * C; A = A * B;
```

Recikliranje operatora

Jedan operator možemo koristiti pri implementaciji drugog.

- Izbjegavamo dupliciranje koda.
- Smanjujemo mogućnost za pogrešku.

```
class Razlomak {
    ...
    Razlomak &operator*=( const Razlomak &b ) { ... }

    Razlomak &operator/=( const Razlomak &b )
    {
        // koristimo operator *=
        return *this *= b.inverz();
    }

    friend Razlomak operator*( const Razlomak &a, const Razlomak &b );
};

Razlomak operator*( const Razlomak &a, const Razlomak &b ) {
    Razlomak rez = a;
    rez *= b; // koristimo operator *= iz klase Razlomak
    return rez;
}
```

Operatori inkrementa i dekrementa

Prefiksne operatore (`--a` i `++a`) implementiramo ovako:

```
class Razlomak {
    ...
    Razlomak &operator++() {
        *this += 1;
        return *this;
    }
    // ili friend Razlomak &operator++( Razlomak &a );
    Razlomak &operator--() { ... }
```

Postfiksne operatore (`a++` i `a--`) implementiramo ovako:

```
class Razlomak {
    ...
    Razlomak operator++( int ) { // int da se razlikuje od prefiksnog!
        Razlomak ret( *this );
        *this += 1;
        return ret;
    }
    // ili friend Razlomak operator++( Razlomak &a, int );
    Razlomak operator--( int ) { ... }
```

Operatori uspoređivanja: <, <=, ==, !=, >, >=

Logična implementacija je izvan klase, s povratnom vrijednosti tipa `bool`.

```
class Razlomak {
    ...
    friend bool operator==( const Razlomak &a, const Razlomak &b );
};

bool operator==( const Razlomak &a, const Razlomak &b )
{
    if( a.brojnik == 0 && b.brojnik == 0 )
        return true; // 0/2 == 0/3

    return a.brojnik == b.brojnik && a.nazivnik == b.nazivnik;
}

Razlomak A(1, 2);
if( 3 == A ) { ... } // OK
```

Operatore možemo implementirati i "nelogično".

- Na primjer, zbroj dva razlomka može biti string, rezultat uspoređivanja razlomaka pomoću == može biti `Razlomak`, itd.

Operator <

Tek kada je na klasi definiran operator < možemo deklarirati skup i mapu objekata te klase.

```
set<Razlomak> S;
S.insert( Razlomak( 2, 5 ) );
S.insert( Razlomak( 3 ) );
set<Razlomak>::iterator si = S.find( 3 ); // ima ga
si = S.find( Razlomak( 4, 5 ) ); // nema ga

map<Razlomak, int> M;
M[Razlomak( 3, 4 )] = 7;
M[7] = 12; // isto kao: M[Razlomak( 7 )] = 12;
```

Operatori pretvorbe tipa (cast)

Konverzija iz `int` u `Razlomak` implementirana je konstruktorom.
Kako napraviti konverziju iz `Razlomka` u npr. `float`?

```
Razlomak a( 1, 2 );
float f = a; // želimo: f = 0.5
```

Kompajler ovdje poziva operator pretvorbe tipa (cast):

```
Razlomak a( 1, 2 );
float f = (float) a; // želimo: f = 0.5
```

Za svaki tip (ugrađen ili naš) možemo definirati operator pretvorbe.
Uoči: ovim operatorima ne pišemo povratni tip!

```
class Razlomak {
    ...
    operator float() const
    {
        return (float)brojnik / nazivnik;
    }
};
```

Operatori pretvorbe tipa (cast)

Konverzije možemo definirati i između tipova koje sami definiramo:

```
class Kune {
    int iznos;
    Kune( int iznos_ ) { iznos = iznos_; }

    operator Lipe() { return Lipe( 100*iznos ); }
};

Kune K( 5 );
Lipe L = K;
```

Ne treba pretjerivati s definiranjem operatora pretvorbe, kako ne bi došlo do dvosmislenosti: ako definiramo i `(int) Razlomak`, račun `a+2` vodi na compile error!

Tako u klasi `string` nije definirana konverzija u `const char *`, koja bi glasila:

```
operator const char *() const { ... }
```

Zadatak 2

1. Nadopunite klasu razlomak, tako na njoj rade operatori za uspoređivanje.

```
Razlomak a, b;
if( a == b ) ...
if( a <= b ) ...
if( a < b ) ...
if( a >= b ) ...
if( a > b ) ...
if( a != b ) ...
```

2. Implementirajte konverziju `Razlomka` u `bool` (vraća `true` ako je razlomak različit od nule), te unarni operator `!` (vraća `true` ako je razlomak jednak nuli).

```
if( a ) ...
if( !a ) ...
```

Operatori << i >>

U kontekstu cjelobrojnih tipova, `a << 2` je obično shiftanje bitova ulijevo, a `a >> 2` shiftanje bitova udesno.

U kontekstu *streamova*, `cout << a` ispisuje, a `cin >> a` učitava varijablu.

- `cout` je objekt klase `ostream`.
- `cin` je objekt klase `istream`.

```
class Razlomak {
    ...
    friend ostream &operator<<( ostream &f, const Razlomak &r );
};

ostream &operator<<( ostream &f, const Razlomak &r )
{
    f << r.brojnik << "/" << r.nazivnik;
    return f; // ulancavanje!
}
...
Razlomak a( 1, 2 ), b( 3, 4 );
cout << a << b << endl; // OK, ispis: 1/2 3/4
```

Nadogradnja *(overloading)* operatora

- Možemo promijeniti tipove podataka koje operatori vraćaju (npr. `operator==` vraća string, a ne bool).
- Operatore možemo i preopteretiti (isti operator se različito ponaša za različite tipove parametara).

```
Razlomak operator*( const Razlomak &a, const Razlomak &b ) { ... }
Razlomak operator*( const Razlomak &a, int b ) { ... }
Razlomak operator*( float a, const Razlomak &b ) { ... }
```

- Svi operatori osim operatora pridruživanja = se prenose i u naslijeđene klase.
- Operatori mogu u baznoj klasi biti označeni kao `virtual`.

Operatori * i ->

Prisjetimo se implementacije liste s iteratorom:

```
template <class Type>
struct list
{
    ...
    Type element[100];
    struct iterator
    {
        list *parent; int index;
        ...
        Type& data() { return parent->element[index]; }
        iterator next() { return iterator( parent, index+1 ); }
        int isEqual( iterator it ) { ... }
    };
};
```

- `isEqual` sada možemo implementirati kao `operator==`.
- `next` sada možemo implementirati kao `operator++`.
- Umjesto `li.data()` želimo pisati `*li`.
- Ako u listi čuvamo strukture s članom `nesto`, onda želimo pisati `li->nesto`.

Operatori * i ->

`operator*` vraća referencu na odgovarajući objekt.

`operator->` vraća pokazivač na neku strukturu, a kompajler će onda toj strukturi ponovno proslijediti `operator->`.

```
template <class Type>
struct list
{
    Type element[100];
    ...
    struct iterator
    {
        list *parent; int index;
        ...
        Type *operator->() { return &parent->element[index]; }
        Type &operator*() { return parent->element[index]; }
    };
};
```

Zadatak 3

Napišite parametriziranu strukturu `list` koja može pamtiti do 100 elemenata određenog tipa s naredbama `push_back`, `begin` i `end`.

- Napišite `operator[](int index)` koji vraća referencu na element u listi.
- Implementirajte podstrukturu `iterator` s operatorima `++` (i prefiksni i postfiksni), `==`, `!=`, `*` (dereferenciranje) i `->`.

```
int main()
{
    list<Razlomak> L;
    L.push_back( 1 ); L.push_back( Razlomak(2, 3) );

    cout << L[1] << endl; // 2/3

    for( list<Razlomak>::iterator li = L.begin(); li != L.end(); ++li )
    {
        cout << *li << endl;
        li->ispisi(); // ispisi je clanska funkcija u klasi Razlomak
    }

    return 0;
}
```

operatori i const

Kao i kod drugih funkcija članica, dozvoljeno je imati dva identična operatora kao člana klase: jedan koji radi s const objektima, a drugi koji radi s ne-const objektima.

Objasnite zbog čega prvi mora vraćati referencu na const!

```
template<class Type>
class list {
    Type element[100];
    ...
    const Type& operator[] ( int index ) const { return element[index]; }
    Type& operator[] ( int index ) { return element[index]; }
};

void f( const list<Razlomak> &L ) {
    cout << L[1];              // poziva const varijantu
    L[1] = Razlomak( 1, 2 ); // !!! compile error, L[1] je const Razlomak & !!!
}

list<Razlomak> L;
L.push_back( Razlomak( 2, 3 ) ); L.push_back( 5 );

cout << L[1];              // poziva ne-const varijantu
L[1] = Razlomak( 1, 2 ); // poziva ne-const varijantu
```

Strukture s pokazivačima

Ako imamo strukturu koja dinamički alocira memoriju, obično trebamo implementirati:

- destruktor kojim oslobađamo svu memoriju koju zauzima objekt koji nestaje;
- copy-konstruktor kojim alociramo memoriju za novu kopiju objekta, te kopiramo podatke iz postojeće u novu kopiju;
- `operator=` koji obavlja sličan posao kao copy-konstruktor, ali se poziva u drugim situacijama.

```
void f( MyString str ) { ... }
MyString g() { MyString ret("ret"); return ret; }

// copy-konstruktor se koristi kod inicijalizacije (a ne operator=)
// copy-konstruktor: A se kopira u B, A se kopira u C
MyString A( "a" ), B( A ), C = A, D, E;

f( A );  // copy-konstruktor: A se kopira u str iz funkcije f
D = A;   // operator=: A se kopira u D
E = g(); // operator=: ret iz funkcije g se kopira u E
```

Strukture s pokazivačima: destruktor

```
class MyString
{
    char *data;
    int size;

public:
    MyString() { data = nullptr; size = 0; }

    MyString( const char *s )
    {
        size = strlen( s );
        data = new char[size+1];
        strcpy( data, s );
    }

    ~MyString() { if( data != nullptr ) delete[] data; }
};
```

Strukture s pokazivačima: copy-konstruktor

```
class MyString
{
    ...
public:
    MyString( const Mystring &s )
    {
        size = s.size;
        if( s.data != nullptr )
        {
            data = new char[size+1];
            strcpy( data, s.data );
        }
        else
            data = nullptr;
    }
};

void f( Mystring str ) { ... }

MyString A( "a" );
f( A ); // copy-konstruktor: A se kopira u str iz funkcije f
```

Strukture s pokazivačima: operator=

```
class MyString {
    ...
public:
    MyString &operator=( const Mystring &s ) {
        if( this == &s ) // slucaj: A = A
            return *this;

        if( data != nullptr )
            delete[] data; // brisemo stari string ako postoji

        size = s.size;
        if( s.data != nullptr )
        {
            data = new char[size+1];
            strcpy( data, s.data );
        }
        else
            data = nullptr;

        return *this; // ulancavanje
    }
};
```

Zadatak 4

Implementirajte i sljedeće operatore za klasu `MyString`:

- zbrajanje stringova (`+`, `+=`);
- uspoređivanje (`==`, `!=`, `<`, `>`, `<=`, `>=`);
- ispisivanje na `ostream` (`<<`);
- `operator<<(int n)` koji briše prvih `n` znakova s početka stringa, te `operator>>(int n)` koji briše zadnjih `n` znakova;
- `operator[](int n)` vraća referencu na `n`-ti znak stringa;
- konverziju u `int` koja vraća duljinu stringa.