Računarski praktikum 1
Vježbe 9

Zvonimir Bujanović
Vinko Petričević

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 podaci[100], vrh;
   ...
   friend void ispisi( const stack& s );
};

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

friend

- Kao prijateljski objekt se može navesti cijela druga klasa, pa npr. sve funkcije klase stack_manager imaju pristup privatnim dijelovima klase stack

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

- ili samo neka funkcija druge klase

```
class stack { ...
    friend void stack_manager1::velicina();
```

Operatori

- Kolika je vrijednost donjih izraza?

```
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;
```

Prioritet operatora koje smo do sada susretali

- najveći prioritet ima `::`
- nakon toga dolaze `.`, `->`, `[]`, `()`, `++` i `--` (postfix). Asocijativni slijeva nadesno.
- `sizeof`, `++` i `--` (prefix), `~` i `!`, pa unarni `-` i `+`, pa `&` (referenciranje), `*` (pokazivač), `new`, `delete` pa `()` – castanje. Asocijativni sa desna nalijevo.
- `.*` i `->*`.
- `*` (množenje), `/` i `%`.
- `+` (zbrajanje) i `-` (oduzimanje).
- `<<` i `>>`
- `<`, `>`, `<=` i `>=`
- `==` i `!=`
- `&` - bitovni i
- `^` - bitovni isključivi ili
- `|` - bitovni ili
- `&&` - logički i
- `||` - logički ili
- `uvjet?izraz1:izraz2` - kondicional – sa desna nalijevo
- `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `|=`, `^=` - sa desna nalijevo
- `,`

Predefiniranje operatora

- promotrimo donju klasu:

```
class razlomak {
   int m_p, m_q; // brojnik, nazivnik
   static int gcd( int a, int b ) { ... };
   void skratiMe() { ... };
public:
   razlomak( int p=0, int q=1 ) : m_p(p), m_q(q)
   { skratiMe(); }
   ...
};
```

- želimo omogućiti sljedeću funkcionalnost:

```
razlomak A( 2 ), B( 2, 7 ), C, D;
C = A + B; D = A * C; D *= B;
cout << C; // treba ispisati 16/7
```

Predefiniranje operatora

- Većini operatora na nekoj klasi možemo (pre)definirati značenje (svi osim `.`, `.*`, `::`, `? :`, # i ##)
- Operatori su funkcije (nestatički elementi klasa ili globalne) koje se zovu `operatorx`, gdje je *x* simbol operatora.

```
class razlomak {
   int m_p, m_q;
   static int gcd( int a, int b ) { ... };
   void skratiMe() { ... };
public:
   razlomak( int p=0, int q=1 ) : m_p(p), m_q(q)
   { skratiMe(); }
   razlomak operator*( const razlomak& b ) const {
      razlomak rez;
      rez.m_p = m_p * b.m_p;
      rez.m_q = m_q * b.m_q;
      rez.skratiMe();
      return rez; }
};
```

Predefiniranje operatora

```
class razlomak {
   razlomak inverz() const {
      return razlomak( m_q, m_p );
   }
public:
   razlomak& operator/=( const razlomak& b ) {
      return *this *= b.inverz();
      // ili return operator*=( b.inverz() );
   }
   razlomak& operator*=( const razlomak& b ) {
      razlomak rez = *this * b; *this = rez;
      return *this;
   }
   razlomak operator/( const razlomak& a ) const
   { ... }

   void ispisi() { cout<<m_p<<"/"<<m_q<<endl; }
};
```

Predefiniranje operatora

- Sada bez problema radi kôd:

```
int main()
{
  razlomak a( 1 ), b( 5,10 );
  (a*=b).ispisi(); // 1/2
  razlomak c = a*b; // ili a.operator*(b);
  c.ispisi(); // 1/4
  razlomak d = c.operator/(razlomak(1,6));
  d.ispisi(); // 3/2
  return 0;
}
```

Zadatak 1

- Nadopunite klasu razlomak, tako da joj definirate operatore zbrajanja, oduzimanja, te unarnog minusa (negativni razlomak).
- Obratite pažnju na skraćivanje
- Također obratite pažnju da ne bi bilo dobro da nazivnik bude negativan. Neka se o tome brine funkcija skratiMe.

operatori kao friend funkcije

- Promotrimo kôd:

```
razlomak a( 1 ), b;
b = a * 2; // radi: implicitno se poziva razlomak(2)
b = 2 * a; // ne radi: int nema operator*(const raz &A)
```

- Ako operator* stavimo kao nečlansku funkciju, gornji kôd će raditi:

```
razlomak operator*(const razlomak& a, const razlomak& b) {
  razlomak ret( a );
  return ret *= b;
}
```

- Ako je operator nečlanska funkcija, bilo bi dobro da je u klasi naznačen kao `friend` (ako treba pristup privatnim dijelovima)

```
class razlomak {
  friend razlomak operator*(const razlomak& a, const razlomak& b);
  ...
};
```

operatori ++ i - -

- Prefiksni operator ++ i - - se zovu (- -a i ++a):

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

- Postfix operator ++ i - - se zovu (a- - i a++):

```
   razlomak operator++( int ) {
      razlomak ret( *this );
      *this += 1;
      return ret;
   }//ili friend razlomak operator++(razlomak& a,int );
   razlomak operator--( int ) ...
```

operatori uspoređivanja: ==, !=, <, <=, >, >=

- bilo bi logično da je rezultat bool
- na klasi razlomak ih je logičnije (zašto?) implementirati kao nečlansku funkciju:

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

bool operator==( const razlomak& a, const razlomak& b ) {
  if( !a.m_p && !b.m_p ) return true; // 0/2 == 0/3
  return a.m_p == b.m_p && a.m_q == b.m_q;
}
```

- Sada radi kôd:

```
razlomak a, b; ...
if( a==b ) ...
```

operator <

- ako je na klasi definiran operator<, onda možemo imati skup i mapu objekata te klase
- donji kod je tada ispravan:

```
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;
```

cast

- Promotrimo kôd:

```
razlomak a( 1, 2 );
float f = a;
```

- kompajler to shvaća kao:

```
float f = (float)a;
```

- pa bi bilo dobro napraviti operator cast-anja iz tipa razlomak u float:

```
class razlomak {
   operator float() const { return (float)m_p/m_q; }
   ...
```

- kod cast-a treba biti oprezan, da se ne bi javile greške sa dvosmislenošću (ako imamo `(int)razlomak`, što je izraz `a+2`?)
- u std::string npr. nije napravljeno kastanje u const char*, koje bi bilo:

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

Zadatak 2

- 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 ) ...
```

- Također neka radi i konvertiranje u bool (true ako je brojnik različit od nule), i operator ! (true ako je brojnik jednak nuli)

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

operatori << i >>

- u kontekstu int-ova su shift-anje bitova ulijevo i udesno, u kontekstu stream-ova su učitavanje/ispisivanje
- cout je objekt klase ostream, cin objekt klase istream

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

ostream& operator<<(ostream& f, const razlomak& r)
{
   f << r.m_p << "/" << r.m_q;
   return f;
}
...
razlomak a; cout << a << endl;
```

Predefiniranje operatora

- Možemo promijeniti tipove podataka koje vraćaju (npr. `operator==` vraća string, a ne bool)
- Operatore možemo i preopteretiti (isti operator se različito ponaša za različite tipove parametara).
- Svi osim `operator=` se prenose i u naslijeđene klase (on se uz to i ne može deklarirati kao nečlanska funkcija – npr. kao friend)
- Mogu biti definirani i kao virtual

operatori * i ->

- prisjetimo se strukture list sa iteratorom:

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

- isEqual možemo zamijeniti sa operator==, next sa operator++
- uoči: umjesto li.data() = 5 smo kod STL-liste pisali *li = 5
- slično, želimo omogućiti i li->nesto = 5 ako u listi čuvamo strukturu Type sa članom nesto

operatori * i ->

- operator* vraća referencu na odgovarajući objekt
- operator-> vraća pokazivač na neku strukturu, a kompajler će onda toj strukturi proslijediti `operator->`

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

Zadatak 3

- Napišite parametriziranu strukturu list koja može pamtiti do 100 elemenata određenog tipa sa naredbama push_back, begin i end
- Napišite `operator[](int index)` koji vraća referencu na tip
- Napravite podstrukturu iterator sa operatorima ++ (i prefiksni i postfiksni), ==, `!=`, * (dereferenciranje) i `->`

```
int main() {
  list<razlomak> p;
  p.push_back(1); p.push_back(razlomak(2,3));
  cout << p[1] << endl; // 2/3
  for( list<razlomak>::iterator i=p.begin();
           i != p.end(); ++i ) {
     cout << *i << endl;
     i->ispisi();
  }
  return 0; }
```

Strukture s pokazivačima

- Ako imamo strukturu koja ima pokazivače (dinamički alocira memoriju), ta struktura bi trebala imati definirane neke posebne funkcije
- destruktor – potreban je da bi se oslobodila memorija koju je zauzeo konstruktor, copy-konstruktor ili `operator=`
- copy-konstruktor – potreban je da se ne bi desila dva puta destrukcija pointera
- `operator=` – da se ne bi dva puta desila destrukcija nekog pointera i da ne bi neki dijelovi memorije ostali zauzeti (jer za svaku strukturu kompajler kreira ovaj operator koji samo prekopira vrijednosti)

destruktor

- na kraju korištenja klase, potrebno je osloboditi memoriju koju je ona zauzela

```
class STRING {
   char *data;
   int size;
public:
   STRING() { data = 0; size = 0; }
   STRING( const char *s ) {
      size = strlen( s );
      data = new char[size+1];
      strcpy( data, s );
   }

   ~STRING() { if( data ) delete data; }
};
```

Copy-konstruktor

- ima jedan parametar tipa `const ime_klase&`

```
struct klasa {
   klasa( const klasa& k );
};
```

- kompajler ga poziva kada se poziva funkcija koja kao parametar prima objekt tipa *klasa*

```
void f( klasa c ) { ... }
```

- i kada funkcija vraća objekt tipa *klasa*

```
klasa g() {
   klasa ret, ret1;
   ...
   return ret;
}
```

Copy-konstruktor

- pri inicijalizaciji se isto koristi copy-konstruktor (ili obični konstruktor), a ne operator=

```
STRING s = "ab"; // u stvari s( "ab" )
STRING s1 = s;   // u stvari s1( s )
```

- promotrimo strukturu STRING:

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

operator=

- pridruživanje, koristi se kada pišemo a = b

```
class STRING {
public:
   STRING& operator=( const STRING& s ) {
      if( this == &s ) return *this; // a=a

      if( data ) delete data;

      size = s.size;
      if( s.data != NULL ) {
         data = new char[size+1];
         strcpy( data, s.data );
      } else data = 0;

      return *this;
   }
};
```

Zadatak 4

- Napišite i ostale naredbe iz klase STRING. To su:
  - dodavanje (+, +=)
  - uspoređivanje (==, `!=`, <, >, <=, >=)
  - ispisivanje na stream (<< ,gdje je prvi parametar output-stream)
  - neka operator `<<(int n)` briše prvih *n* znakova s početka stringa, a operator `>>(int n)` zadnjih *n* znakova
  - `operator[](int n)` vraća referencu na *n*-ti znak stringa
  - cast-anje u `int` vraća duljinu stringa.