Sveučilište u Zagrebu
PMF – Matematički odjel

Objektno programiranje (C++)

Vježbe 03 – Operatori

Vinko Petričević

Što je to izraz?

- Izraz se sastoji od jednog ili više operanada, te operacija koje su na njima primjenjene
- Primjeri:

```
x + y
points[size/2] * delta
ime + " " + prezime
ptr != 0 && *ptr != 0
```

Aritmetički operatori

Operator	Upotreba
*	expr * expr
/	expr / expr
%	expr % expr
+	expr + expr
-	expr – expr

Relacijski i logički operatori

Operator	Upotreba
!	!expr
<	expr < expr
<=	expr <= expr
>	expr > expr
>=	expr >= expr
==	expr == expr
!=	expr != expr
&&	expr && expr
||	expr || expr

Operatori za rad s bitovima

Operator	Upotreba
~	~expr
<<	expr << expr
>>	expr >> expr
&	expr & expr
^	expr ^ expr
|	expr | expr
&=	expr &= expr
^=	expr ^= expr
|=	expr |= expr

Operatori pridruživanja

- Složeni operatori pridruživanja:

```
+=    -=    *=    /=    %=
<<=   >>=   &=    ^=    |=
```

- `a op= b;`
  je ekvivalentno
  `a = a op b;`
- Primjer:

```
int sum = 0;
for (int i = 0; i < sz; ++i)
    sum += a[i];
```

Inkrement i dekrement

- Postfix verzija
  - inkrement: `x++`
  - dekrement: `x--`
- Prefix verzija
  - inkrement: `++x`
  - dekrement: `--x`

Uvjetni operator

- Sintaktički oblik:

```
expr1 ? expr2 : expr3
```

- Primjer:

```
int i = 10, j = 20, k = 30;
cout << "Veca od vrijednosti "
     << i << " i " << j << " je "
     << ( i > j ? i : j ) << endl;
```

- Zadatak: Za dane varijable `i`, `j` i `k` što računa slijedeći izraz?

```
((i > j) ? (( i > k) ? i : k)
         : ( j > k ) ? j : k)
```

sizeof operator

- Sintaktički oblici:
  - `sizeof (ime_tipa);`
  - `sizeof (ime_objekta);`
  - `sizeof ime_objekta;`
- Zadatak: Neka je zadano polje `a`:

  ```
  int a[] = {0, 1, 2};
  ```

  Koja je vrijednost izraza `sizeof(a)`?
  Kako biste odredili broj elemenata polja `a`?

sizeof operator

- Zadatak: Neka je zadano:

```
string st1("foobar");
string st2("racunarski praktikum");
string *ps = &st1;
```

Koliko iznosi:

```
sizeof(st1)      sizeof(st2)
sizeof(ps)       sizeof(*ps)
```

Koje su vrijednosti od:

```
sizeof(short)    sizeof(short*)
sizeof(short&)   sizeof(short[3])
```

Prioriteti operatora

- Zadatak: Identificirajte redoslijed evaluacije slijedećih složenih izraza:

```
(a)  ! ptr == ptr->next
(b)  ~ uc ^ 0377 & ui << 4
(c)  ch = buf[bp++] != '\n'
```

- Zadatak: Slijedeća dva izraza sadrže sintaktičke greške uslijed nepažnje na prioritet operatora. Napravite potrebne korekcije.

```
(a) int i = doSomething(), 0;
(b) cout << ival % 2 ? "odd" : "even";
```

Prioriteti 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
- `,`

Klase – predefiniranje operatora

- promotrimo 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
```

Klase – 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 {...
public: ...
  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;
  }
};
```

Klase – 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; }
};
```

Klase – predefiniranje operatora

- Sada bez problema radi kôd:

```
{
  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
}
```

Zadatak

- 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.

Klase - 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;
}
```

Klase – 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) ...
```

cast

- Promotrimo kôd:

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

- kompajler to shvaća kao:

```
double f = (double)a;
```

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

```
class razlomak {
  operator double() const {
        return (double)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

- 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 ->

- Pogledajmo strukturu 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

- 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;
}
```

Klase 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)

Zadatak

- Napišite klasu STRING sa sljedećim funkcijama i operatorima:
  - konstruktor bez parametara – kreira prazan string
  - konstruktor s jednim parametrom tipa const char* - kreira kopiju stringa
  - destruktor, copy-konstruktor i pridruživanje
  - 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.

Implicitne konverzije

- Implicitne konverzije tipova
  - Kod aritmetičkih izraza:
```
int ival = 3;
double dval = 3.14159;
ival + dval;
```
  - Prilikom pridruživanja:
```
int* pi = 0;
ival = dval;
```
  - Prilikom proslijeđivanja argumenata kod poziva funkcije:
```
extern double sqrt(double);
sqrt(2);
```
  - Prilikom vraćanja povratne vrijednosti iz funkcije:
```
double f(int x, int y) {
    return x + y;
}
```

Aritmetičke konverzije

- Ukoliko dođe do potrebe tipovi se uvijek promoviraju na širi tip
- Svi aritmetički izrazi koji uključuju integralne tipove manje od `int`-a promoviraju se u `int`
- Primjeri:
  - ```
    char cval;
    long lval;
    cval + 1024 + lval;
    ```
  - ```
    char cval;
    int ival;
    float fval;
    cval + fval + ival;
    ```

Implicitne konverzije

- Zadatak: Neka su zadane deklaracije:

```
char cval;   int ival;
float fval;  double dval;
unsigned int ui;
```

Identificirajte implicitne konverzije tipova u slijedećim izrazima:

```
(a) cval = 'a' + 3;
(b) fval = ui - ival * 1.0;
(c) dval = ui * fval;
(d) cval = ival + fval + dval;
```

Eksplicitne konverzije – static_cast

- `static_cast` operator sličan je običnom castanju
  - ```
    double pi = 3.14;
    int i = d; // i = (int) d;
               // i = static_cast<int>(d);
               // i = 3;
    int ival = 1024;
    int* pi = &ival;
    void* pv;
    const char* pc;
    pv = pi;
    pc = static_cast<char*>(pv);
    ```
  - ```
    double dval = 3.14159;
    int ival = 3;
    val += static_cast<int>(dval);
    ```
- pointere može castati samo iz izvedenih u bazne klase i obratno. (Obratno može biti i opasno)

Eksplicitne konverzije

- `const_cast` operator uklanja const i volatile atribute

```
extern char* string_copy(char*);
const char* str;
char* p =
string_copy(const_cast<char*>(str));
```

- `reinterpret_cast` binarno kopira pointerske tipove, ali ne radi sa običnim tipovima
  -
```
complex<double>* pcom;
char* pc =
reinterpret_cast<char*>(pcom);
```

- `implicit_cast` binarno kopira sadržaj – ne radi u VS

Eksplicitne konverzije – reinterpret_cast

- ```
  struct X {
     int a;
  };
  struct Z {
     double d;
  };
  ```
- ```
     X x;
     x.a = 5;
     Z *z1 = (Z*)&x;
     Z *z2 = reinterpret_cast<Z*>(&x);
     // Z *z3 = static_cast<Z*>(&x);
     // Z *z4 = &x;
  ```

Eksplicitne konverzije – static_cast

- pametno se ponaša prilikom višestruktog nasljeđivanja, ali može izazvati greške:

```
struct X {
    double a;
};
struct Y {
    int b;
};
struct Z : public X, public Y { };
    Z z;
    X* px = static_cast<X*>(&z); // = &z;
    Y* py = static_cast<Y*>(&z); // = &z;
    cout<<px<<endl;
    cout<<py<<endl;
    cout<<(long)py-(long)px<<endl; // sizeof(X)
// px pokazuje na X dio klase, a py na y dio
    Y y;
    Z* pz = static_cast<Z*>(&y); // = (Z*)&y;
    cout<<(long)pz-(long)&y<<endl; // -sizeof(X)
// pz pokazuje na dio prije varijable y
```

Eksplicitne konverzije – static_cast

```
struct X {
    int a;
};
struct Y : public X {
    int b;
};
struct Z : public X, public Y {
};
```

```
    Z z;
    z.b = 3;

    X* x = &z; x->a = 1;
    Y* y = &z; y->a = 2;

    cout<<x->a<<endl;
    cout<<y->a<<endl;
    cout<<z.b<<endl;
```

Eksplicitne konverzije – static_cast

- ```
  struct X {
     int a;
  };
  struct Y : public X {
     int b;
  };
  ```
- ```
     Y y;
     X x;
     X *xp = &y;
     // Y *yp = &x;
     Y *yp = static_cast<Y*>(&x);
  ```

Eksplicitne konverzije – dynamic_cast

- `static_cast` je donekle pametan (int-double)
- dobro se ponaša prilikom polimorfizma, ali može uzrokovati probleme ako pokušamo castati nasljeđenu klasu iz bazne
- `dynamic_cast` je rješenje – vraća null ako nije moguće izvesti konverziju (o tome ćemo više govoriti kada budemo radili nasljeđivanje)

Eksplicitne konverzije – static_cast

- ```
  struct X {
      int a;
  };

  struct Y : public X {
      int b;
  };
  ```
- ```
  void f(X* x) {
      Y* y = static_cast<Y*>(x); // y=(Y*)x;
      if (y) cout<< y->a <<" "<< y->b <<endl;
  }
  ```
- ```
      X x(1);
      Y y; y.a=2; y.b=3;
      f(&y);
      f(&x);
  ```
  ```

Eksplicitne konverzije – dynamic_cast

- ```
  struct X {
      int a;
      virtual ~X(){}
  };
  struct Y : public X {
      int b;
  };
  ```
- ```
  void f(X* x) {
      Y* y = dynamic_cast<Y*>(x);
      if (y) cout<< y->a <<" "<< y->b <<endl;
  }
  ```
- ```
      X x; x.a = 1;
      Y y; y.a=2; y.b=3;
      f(&y);
      f(&x);
  ```
  ```