Objektno orijentisano programiranje 1

Preklapanje operatora

Pojam preklapanja operatora

- Ako su u programu potrebni kompleksni brojevi i operacije nad njima
  - pogodno je da se operacije mogu predstaviti standardnim operatorima
  - na primer: `Kompleksni c1(1,2),c2(3,4),c3; c3=c1+c2;`
- C++ dozvoljava preklapanje operatora
  - drugi termin: preopterećenje operatora (*operator overloading*)
  - definišu se nova značenja operatora za korisničke tipove (klase)
  - princip je sličan kao kod preklapanja imena funkcija
- Preklopljeni operatori za klasne tipove su specijalne operatorske funkcije
- Operatorske funkcije nose ime `operator@`
  - simbol @ predstavlja neki operator ugrađen u jezik
- Operatorske funkcije preklapaju standaradne operatore (`+`, `-`, `*`, `/`, ...)
- Pozivanje operatorskih funkcija u izrazima
  - može biti notaciono isto kao i korišćenje operatora nad ugrađenim tipovima
  - izraz `t1@t2` se tumači kao:
    - `operator@(t1,t2) // za operatorsku prijateljsku funkciju klase`
    - `t1.operator@(t2) // za operatorski metod klase`

Primer operatorskih funkcija

```
class Kompleksni {
public:
  Kompleksni(double,double);   /* konstruktor */
  friend Kompleksni operator+(Kompleksni, Kompleksni);  /* oparator + */
  friend Kompleksni operator-(Kompleksni, Kompleksni);  /* operator - */
private:
  double real, imag;
};
Kompleksni::Kompleksni(double r=0.0, double i=0.0): real(r), imag(i) {}
Kompleksni operator+ (Kompleksni c1, Kompleksni c2) {
  Kompleksni c;
  c.real=c1.real+c2.real;
  c.imag=c1.imag+c2.imag;
  return c;
}
Kompleksni operator-(Kompleksni c1, Kompleksni c2) {
  return Kompleksni(c1.real-c2.real, c1.imag-c2.imag);
}
...
Kompleksni c1(1.0,1.0),c2(2.0,2.0),c3,c4;
c3=c1+c2;      /* poziva se operator+(c1,c2) */
c4=c2-c3;      /* poziva se operator-(c2,c3) */
```

Ograničenja preklapanja operatora

- Postoje neka ograničenja u preklapanju operatora:
  - ne mogu da se preklope operatori:
    `.`, `.*`, `::`, `?:`, `sizeof`, `alignof`, `typeid` i `throw`
  - ne mogu da se menjaju značenja operatora za primitivne (standardne) tipove podataka
  - ne mogu da se uvode novi simboli za operatore
  - ne mogu da se menjaju osobine operatora: *n*-arnost, prioritet i asocijativnost
  - neki operatori imaju posebna ograničenja za preklapanje:
    `=`, `i++`, `i--`, `[]`, `()`, `->`, `(tip)`, `new`, `delete`

Pravila o preklapanju operatora

- Za korisničke tipove su unapred definisani sledeći operatori:
  - = (dodela vrednosti), & (uzimanje adrese), `.`(pristup članu), `,` (lančanje)
  - dok ih korisnik ne preklopi (osim `.`), oni imaju podrazumevano značenje
- Za izvedene (pokazivačke) tipove iz korisničkih tipova definisani su:
  - `*` (indirektno adresiranje), `->` (posredan pristup članu), `[]` (indeksiranje)
  - dok ih korisnik ne preklopi i oni imaju podrazumevano značenje
- Vraćene vrednosti operatorskih funkcija mogu da budu bilo kog tipa
  - može i `void`
- Ako se simbol operatora sastoji od slova (npr. `new`), mora se pisati odvojeno od ključne reči `operator`
- Operatorske f-je ne mogu da imaju podrazumevane vrednosti argumenata
- Operatorski metodi ne mogu biti statički (osim `new` i `delete`)
- Deo potpisa operatorskog metoda čine i modifikatori `const`, ...

Bočni efekti i veze između operatora

- Postojeći bočni efekti kod operatora za primitivne tipove ne podrazumevaju se za preklopljene operatore
  - bočni efekti postoje kod:
    - operatora ++ i `--` (prefiksnih i postfiksnih)
    - svih operatora dodele (=, +=, `-=`, `*=`, ...)
  - može se preklopiti svaki od njih tako da nema bočni efekat
    - to nije dobra praksa
  - može se napraviti bočni efekat kod operatora koji ga inače nema
    - ni to nije dobra praksa
- Veze koje postoje između operatora za primitivne tipove ne podrazumevaju se za preklopljene operatore
  - na primer, ako je definisan `operator+`
    - `a+=b` ne znači automatski `a=a+b`
    - ako je potreban, operator += mora posebno da se preklopi

Preporuke

- Preklopljeni operatori treba da imaju očekivano značenje (zbog čitljivosti programa)
  - na primer, ako su definisani i `operator+=` i `operator+`, dobro je da `a+=b` ima isti efekat kao i `a=a+b`
  - ako standardni operator proizvodi bočni efekat, treba i preklopljeni
    - operatori dodele i `++`/`--` treba da menjaju stanje (levog za dodele) operanda
  - ako standardni operator vraća *lvrednost*, treba i preklopljeni
    - operatori dodele i prefiksni `++`/`--` treba da vraćaju rezultat po referenci
- Kada se definišu operatori za klasu, treba težiti da njihov skup bude kompletan
  - na primer, ako su definisani `operator=` i `operator+`, treba definisati i `operator+=`
  - za definisan `operator==` treba definisati i `operator!=`

Operatorski metodi/glob. funkcije

- Operatorske funkcije mogu biti:
  - metodi ili
  - globalne (prijateljske) funkcije
- Kod globalne operatorske funkcije
  bar jedan parametar mora biti klasnog tipa
- Ako je @ binarni operator (na primer +), on može da se realizuje:
  - kao metod klase X: `<tip> operator@(X)`
    - poziv a@b se tumači kao: `a.operator@(b)`
  - kao globalna (prijateljska) funkcija: `<tip> operator@(X,X)`
    - poziv a@b se tumači kao: `operator@(a,b)`
- Nije dozvoljeno da se u programu nalaze obe ove funkcije

Ograničenja operatorskog metoda

- Ako levi operand binarne operacije treba da bude standardnog tipa ➔ mora se deklarisati globalna funkcija
  - razlog: kod operatorskog metoda levi operand je skriveni argument
    - skriveni argument je uvek tipa klase čiji je metod f-ja članica
    - standardni tipovi nisu klasni tipovi
  - primer: `Kompleksni operator-(double d, Kompleksni c)`
  - po pravilu, ovakva f-ja treba da bude prijatelj klasi drugog parametra (operanda)
- Operatorski metod ne dozvoljava konverziju levog operanda

Primer metoda/globalne funkcije

```
class Kompleksni {
  double real, imag;
public:
  Kompleksni(double r=0, double i=0) : real(r), imag(i) {}
  Kompleksni operator+(Kompleksni c)
    { return Kompleksni(real+c.real, imag+c.imag; }
};

class Kompleksni {                        // alternativno
  double real, imag;
public:
  Kompleksni(double r=0, double i=0) : real(r), imag(i) {}
  friend Kompleksni operator+(Kompleksni,Kompleksni);
};
Kompleksni operator+ (Kompleksni c1, Kompleksni c2) {
  return Kompleksni(c1.real+c2.real, c1.imag+c2.imag);
}

void main () {
  Kompleksni c1(2,3),c2(3.4);
  Kompleksni c3=c1+c2;  // c1.operator+(c2) ili operator+(c1,c2)
}
```

Unarni i binarni operatori

- Unarni operator ima samo jedan operand, pa se može realizovati:
  - kao metod bez parametara:
    ```
    tip operator@ ()
    ```
  - kao globalna funkcija sa jednim parametrom:
    ```
    tip operator@ (X x)
    ```
- Binarni operator ima dva operanda, pa se može realizovati
  - kao metod sa jednim parametrom:
    ```
    tip operator@ (X xdesni)
    ```
  - kao globalna funkcija sa dva parametra:
    ```
    tip operator@ (X xlevi, X xdesni)
    ```

Primer unarnih i binarnih operatora

```
class Kompleksni {
  //...
public:
  // metod unarni operator!
  Kompleksni operator!();

  // globalna f-ja, unarni operator~
  friend Kompleksni operator~(Kompleksni);

  // metod binarni operator-
  Kompleksni operator-(Kompleksni);

  // globalna f-ja, binarni operator+
  friend Kompleksni operator+(Kompleksni, Kompleksni);
};
```

Inicijalizacija i dodela vrednosti (1)

- Inicijalizacija objekta pri kreiranju i dodela vrednosti se razlikuju
  - inicijalizacija podrazumeva da objekat još ne postoji
  - dodela podrazumeva da objekat sa leve strane operatora već postoji
- Inicijalizacija se vrši uvek kada se kreira objekat:
  - statički, automatski, dinamički, privremeni i klasni član
- Inicijalizacija poziva konstruktor, a ne operator dodele
  - konstruktor se poziva čak iako je notacija za inicijalizaciju simbol =
  - ako je izraz sa desne strane simbola =
    - istog tipa kao i objekat koji se kreira,
      poziva se konstruktor kopije ili premeštanja
    - različitog tipa u odnosu na objekat koji se kreira,
      poziva se konstruktor konverzije
    - u oba slučaja može biti pozvan i konstruktor sa više parametara,
      ako ostali parametri imaju podrazumevane vrednosti

Inicijalizacija i dodela vrednosti (2)

- Dodelom se naziva izvršavanje izraza sa operatorom dodele =
  - operator dodele se može preklopiti pisanjem operatorske funkcije `operator=`
- Podrazumevano značenje operatora = je kopiranje objekta član po član
  - pri kopiranju atributa tipa klase
    - pozivaju se operatori = odgovarajućih klasa atributa
  - pri kopiranju člana klase tipa pokazivača
    - kopiraće se samo taj pokazivač, a ne i pokazivana vrednost/objekat
    - kada treba kopirati i pokazani objekat treba da se preklopi `operator=`

Preklapanje operatora dodele

- Operatorska funkcija `operator=` mora biti nestatički metod
- Ugrađena varijanta operatora dodele vrši plitko kopiranje
- Kada plitka kopija nije zadovoljavajuća, treba napisati metod `operator=`
  - to je situacija kada objekat sadrži po pokazivaču ili referenci neke delove
- Pošto je kod ugrađene varijante rezultat *lvrednost*, preporuka je da se rezultat vraća po referenci
- Najčešća realizacija:
  - prvo se ispita da nije slučaj poziva a=a; ako jeste, ništa se ne radi
  - zatim se uništavaju stari delovi levog operanda (nije neophodno, pogotovo ako su novi delovi iste veličine)
  - zatim se kopiraju (ili premeštaju) delovi desnog operanda
- Obrazac:

```
X& X::operator=(const X &x){
  if (&x != this) {/* unistavaju se delovi starog *this;
                      formiraju se novi delovi;
                      kopira se sadrzaj iz x */}
  return *this;
}
```

Varijante operatora =

- Postoji kopirajuća i premeštajuća varijanta operatora dodele
- Premeštajuća se primenjuje na *nvrednosti*, odlučuje prevodilac
- Nema razlike u semantici, samo u efikasnosti
- Razlika u tipu parametra:
  - kopirajuća varijanta: `X& X::operator=(const X& x);`
  - premeštajuća varijanta: `X& X::operator=(X&& x);`
- Obe varijante imaju ugrađene (implicitne) definicije
  - atributi primitivnog tipa se prosto kopiraju
  - atributi klasnog tipa se formiraju kopirajućim/premeštajućim operatorom =
- Ugrađena kopirajuća dodela se briše
  - ako se u klasi definiše premeštajući konstruktor ili premeštajuća dodela
  - može se restaurirati sa =**default**
- Ugrađena premeštajuća dodela se briše
  - ako se u klasi definiše kop./prem. konstruktor, destruktor ili kop. dodela
- Ako u klasi ne postoji premeštajuća dodela, koristi se kopirajuća dodela

Preporuke

- Ako se za klasu piše destruktor, konstruktor kopije ili `operator=`
  - sva je prilika da treba da se napišu sve te funkcije
- Ako se za klasu piše `operator=`
  - sva je prilika da treba pisati obe varijante – kopirajuću i premeštajuću
- Preporuka za implementaciju:
  - privatni metodi `kopraj(x)`, `premesti(x)`, `brisi()`
  - pozivaju se iz
    - kopirajućeg konstruktora
    - premeštajućeg konstruktora
    - kopirajućeg operatora =
    - premeštajućeg operatora =
    - destruktora
  - alternativa: *copy-and-swap* idiom
    - objedinjeni kopirajući i premeštajući operator dodele

Primer preklapanja operatora = (1)

- Dopuna ranijeg primera klase `Tekst`

```
#include <cstring>
class Tekst {
public:
  Tekst(const char* niz);
  Tekst(const Tekst& t){kopiraj(t);}  // kopirajuci konstr.
  Tekst(Tekst&& t){premesti(t);}   // premestajuci konstr.
  Tekst& operator=(const Tekst&);  // kopirajuci operator=
  Tekst& operator=(Tekst&&);       // premestajuci operator=
  ~Tekst(){brisi();}               // destruktor
private:
  char *niska;
  void kopiraj(const Tekst& t){
    niska=new char [strlen(t.niska)+1];// zauzimanje prostora
    strcpy(niska,t.niska);             // kopiranje niske
  }
  void premesti(Tekst& t){niska=t.niska; t.niska=nullptr;}
  void brisi() {delete [] niska; niska=nullptr;}
};
```

Primer preklapanja operatora = (2)

```
Tekst& Tekst::operator=(const Tekst& t) {
  if (&t!=this) {brisi(); kopiraj(t);}
  return *this;
}
Tekst& Tekst::operator=(Tekst&& t) {
  if (&t!=this) {brisi(); premesti(t);}
  return *this;
}
void main () {
  Tekst a("Pozdrav!"), // Tekst(const char*) [###]
        b=a,           // Tekst(const Tekst&)
        c=Tekst("123");// ###; Tekst(Tekst&&)
  a=b;                 // Tekst::operator=(const Tekst&);
  b=Tekst("ABC");      // ###; Tekst::operator=(Tekst&&);
}
```

Preklapanje operatora ++ i --

- Problem: postoje prefiksne i postfiksne varijante operatora `++` i `--`
- Za preklapanje prefiksnih oblika operatora `++` i `--`:
  - koriste se uobičajene operatorske funkcije:
    - u obliku metoda klase `T`: `T& operator@@()`
    - u obliku globalne (prijateljske) funkcije: `T& operator@@(T&)`
- Za preklapanje postfiksnih oblika operatora `++` i `--`:
  - operatorska funkcija sadrži i jedan dodatni arument tipa `int` i to:
    - u obliku metoda klase `T`: `T operator@@(int)`
    - u obliku globalne (prijateljske) funkcije: `T operator@@(T&, int)`
- Ako se za poziv funkcije koristi postfiksni operator `++` ili `--`
  ➔ parametar tipa `int` ima vrednost `0`
- Ako se za poziv funkcije koristi notacija
  `t.operator@@(k)` ili `operator@@(t,k)` gde je `@@` ili `++` ili `--`
  ➔ može biti `k!=0`

Preklapanje operatora [] (1)

- Operator `[]` je binarni operator kojem odgovara funkcija `operator[]()`
- Operatorska funkcija `operator[]()` mora da bude nestatički metod
  - funkcija `operator[]()` ne može da bude globalna (prijateljska) funkcija
- Parametar:
  - kod standardnog indeksiranja indeksni izraz mora biti celobrojnog tipa
  - kod preklopljenog operatora `[]` indeksni izraz može biti proizvoljnog tipa
- Pozivanje:
  - `zbirka[ind]` je ekvivalent izrazu `zbirka.operator[](ind)`
- Preklapanje operatora `[]` u nekoj klasi omogućava izraze sa `o[i]`, gde je o objekat date klase
- Moguće primene:
  - klasa čiji objekti sadrže nizove sa zadatim granicama indeksa: funkcija indeksiranja može da proverava granice
  - asocijativni pristup elementima zbirke

Preklapanje operatora [] (2)

- Ako izraz sa operatorom `[]` može da bude prvi operand operacije =
  - na primer: `o[i]=izraz;`
  - tip funkcije `operator[]()` treba da bude referenca na objekat
- Dva potpisa funkcije:
  - `Tip& operator[](indeks); // za promenljive zbirke`
  - `const Tip& operator[](indeks) const; // za nepromenljive z.`
- Izraz sa preklopljenim operatorom `[]` nije indeksiranje
  već samo koristi notaciju indeksiranja
  - operatorska funkcija `operator[]()` dejstvuje na objekat svoje klase,
    a ne na niz objekata, kao standardni operator `[]`
  - izraz `obj[ind]` ne može da se zameni sa `*(obj+ind)`, kao kod nizova
- Prirodno je da struktura objekta bude zbirka (kolekcija) elemenata,
  a da se operatorom `[]` odabira neki element (komponenta)

Preklapanje operatora ( )

- Operator `()` je binarni operator kojem odgovara funkcija `operator()()`
- Operatorska funkcija `operator()()` mora da bude nestatički metod
  - funkcija `operator()()` ne može da bude globalna (prijateljska) funkcija
- Parametri:
  - proizvoljan broj proizvoljnog tipa
- Pozivanje:
  - `f(a1,...,aN)` je ekvivalent izrazu `f.operator()(a1,...,aN)`
- Preklapanje operatora`()` u nekoj klasi omogućava izraze sa `o(a1,...,aN)`, gde je o objekat date klase
- Primer:
  `P p; float x;`
  - ako je `P` klasa polinoma, može se pisati `p(x)` za vrednost polinoma u `x`, ako se preklopi funkcija `operator()(float)`
- Klasa sa preklopljenim `operator()` – *funkcijska klasa*
  - objekat funkcijske klase – *funkcijski objekat*

Preklapanje operatora ->

- Operator `->` je binarni operator
  - međutim, preklapa se kao unarni funkcijom `operator->()`
- Operatorska funkcija `operator->()` mora da bude nestatički metod
  - funkcija `operator->()` ne može da bude globalna (prijateljska) funkcija
- Parametri:
  - funkcija `operator->()` mora biti bez parametara (unarni operator)
- Pozivanje:
  - `o->clan` je ekvivalent izrazu `(o.operator->())->clan`
- Rezultat:
  - treba da bude tipa pokazivača na objekat klase koja sadrži `clan` ili
  - objekat (ili referenca) klase za koju je takođe definisan `operator->`
- Primena:
  - pristup članovima objekata preko "pametnih pokazivača"

Primer preklapanja operatora ->

- Odbrojavaju se indirektni pristupi objektu (strukture):

```
struct X{int m};
class Xptr {
  X *p;  int bp;
public:
  Xptr(X *px):p(px),bp(0){} // konstruktor
  X& operator*() {bp++; return *p;}
  X* operator->(){bp++; return p;}
};
void main(){
  X x;  Xptr pp=&x; // poziva se konstruktor
  (*pp).m=1;        // poziva se (pp.operator*()).m;
  int i=pp->m;      // poziva se (pp.operator->())->m;
}
```

Preklapanje operatora `(tip)` (1)

- Preklapanje *cast* operatora je drugi način za konverziju klasnih tipova
  - prvi način konverzije korisničkih tipova je pomoću konstruktora konverzije
- Operator `(T)` je unarni operator kojem odgovara funkcija `operator T()`
  - funkcija treba da izvrši konverziju objekta klase čiji je član u tip `T`
  - `T` može da bude standardni, izvedeni (npr. pokazivač) ili klasni tip
- Operatorska funkcija `operator T()` mora da bude nestatički metod
  - funkcija `operator T()` ne može da bude globalna (prijateljska) funkcija
- Parametri:
  - funkcija nema parametre (unarni operator, član klase)
- Rezultat:
  - tip rezultata funkcije ne sme da bude naveden u deklaraciji/definiciji
  - podrazumeva se na osnovu imena funkcije

Preklapanje operatora `(tip)` (2)

- Pozivanje [za funkciju `X::operator T()`i objekat `X x;`]:
  - `(T)x` ili `T(x)` je ekvivalent izrazu `x.operator T()`
  - drugi oblik je notacijski isti kao da je u pitanju konstruktor `T`
  - moguć oblik je i statički kast: `static_cast<T>(x)`
- Za razliku od konstruktora `T(x)` preklopljeni operator `(T)x`
  - može da se koristi za `T` koje je standardni tip
  - operand `x` mora biti objekat klase (`X`), odnosno `X` ne može biti primitivan tip
  - primer: `int(x)` - konvertuje `x` tipa `X` u tip `int`
- Oblik notacije `T(x)` ne može da se koristi za tipove sa većim brojem reči
  - primer: `(unsigned long) x` nije isto što i `unsigned long(x)`
- Konverzija se primenjuje implicitno (automatski)
- Ako se želi sprečiti implicitna konverzija:
  - modifikator `explicit`, kao kod konstruktora konverzije

Preklapanje operatora `(tip)` (3)

- Primer:

```
class X{ public:
  operator int() {return 1;}
  explicit operator double(){ return 2;}
}
int a=X(); int b=(int)X(); double c=(double)X(); double d=X();
// a==1, b==1, c==2.0, d==1.0;
```

- Konverzija se primenjuje automatski, ako je jednoznačan izbor konverzije
  - ako je definisan konstr. konverzije `T(X)` i operatorska f-ja `X::operator T()`, `t=x` je dvoznačno
  - ako su definisane obe konverzije `T(x)` i `X(t)` i operatori `+` za oba tipa `X` i `T` `x+t` je dvoznačno
- Problem sa konverzijom tipa pri prenosu parametara po referenci
  - rezultat konverzije stvarnog argumenta je privremeni objekat, pa se u funkciju prenosi njegova adresa
  - izmene u funkciji se odnose na taj privremeni objekat

Preklapanje `new` i `delete`

- Može se preuzeti kontrola nad alokacijom memorije od ugrađenog alokatora
  - na primer, kada su objekti klase mali, može se precizno vršiti njihova alokacija, tako da se smanji režija alokacije
- Za ovakve potrebe mogu se preklopiti `new` i `delete` za neku klasu
- Operatorske funkcije `new` i `delete` su statički (`static`) metodi
  - čak i ako nisu tako deklarisane, one su statičke
  - razlog: one se pozivaju pre nego što je objekat stvarno kreiran, odnosno pošto je uništen
- Unutar tela ovih operatorskih funkcija:
  - ne treba eksplicitno pozivati konstruktor, odnosno destruktor
  - konstruktor se implicitno poziva posle operatorske funkcije `new`
  - destruktor se implicitno poziva pre operatorske funkcije `delete`
- Ove operatorske funkcije služe samo da:
  - obezbede prostor za smeštanje dinamičkog objekta (`new`)
  - oslobode prostor koji je bio alociran za dinamički objekat (`delete`)

Preklapanje new

- Operatorska funkcija `new` se deklariše se na sledeći način:
  - `void* operator new (size_t velicina,T2 par2,…,TN parN);`
  - `void* operator new[] (size_t velicina,T2 par2,…,TN parN);`
- Poziva se na način:
  - `new (arg2,…,argN) T(izraz,…,izraz)`
  - `new (arg2,…,argN) T[duzina]`
- Tip `size_t` je celobrojni tip definisan u `<stddef.h>`, odnosno `<cstddef>`
  - služi za izražavanje veličina objekata u bajtovima
- Parametar `velicina` daje veličinu prostora koji treba alocirati za objekat
- Argument za `velicina` je `sizeof(T)`, gde je `T` klasa u kojoj je `new`
  - formira se na osnovu tipa `T` u operaciji `new T`
- Parametri `parX`, odnosno argumenti `argX` – opcione dodatne informacije
- Opciona lista *`izraza`* je inicijalizator – određuju izbor konstruktora
- Broj elemenata niza je *`duzina`*, zauzima se `sizeof(T)*`*`duzina`* bajtova
- Operator `new` treba da vrati pokazivač na alocirani prostor
- Klasa može imati više preklopljenih operatora `new`

Preklapanje `delete`

- Operatorska funkcija `delete` se deklariše na sledeći način:
  - `void operator delete (void* pokazivac);`
  - `void operator delete (void* pokazivac, size_t velicina);`
  - `void operator delete[] (void* pokazivac);`
  - `void operator delete[] (void* pokazivac,size_t velicina);`
- Parametar `pokazivac` je pokazivač na prostor koji treba osloboditi
- Parametar `velicina` određuje prostor u bajtima koji treba osloboditi
- Ako nedostaje odgovarajući stvarni argument za `velicina`
  - funkcija mora sama da odredi veličinu, na osnovu ranije alokacije
- Funkcija `delete` ne vraća rezultat
- Klasa može imati samo po jednu (za podatak i niz) `delete` funkciju
  - bez obzira što postoje po dve preklopljene operatorske funkcije `delete`

Dohvatanje ugrađenih new i delete

- Ako su u klasi T preklopljeni operatori new i delete, ugrađeni operatori new i delete mogu da se unutar dosega T pozivaju:
  - eksplicitno, preko unarnog operatora ::
    - za pojedinačne podatke tipa T - ::new T , odnosno :: delete pt
    - za nizove elemenata tipa T - ::new T[*duz*] , odnosno :: delete[] pt
  - implicitno, kada se dinamički kreiraju/uništavaju objekti koji nisu tipa T
- Primer:

```
#include <cstddef>
using namespace std;
class XX {public:
  void* operator new (size_t sz){ return new char[sz]; }
              // koristi se ugrađeni new
  void operator delete (void *p) { delete [] p; }
              // koristi se ugrađeni delete
};
```

- Metodi new i delete ne mogu biti virtuelni, ali se nasleđuju

Standardni U/I tokovi

- Kao ni jezik C, ni C++ ne sadrži ugrađene U/I naredbe
  - U/I operacije se realizuju standardnim bibliotekama
- Za C++ postoji standardna U/I biblioteka realizovana u duhu OOP-a
- Na raspolaganju je i stara C biblioteka sa funkcijama `scanf` i `printf`
  - njeno korišćenje nije u duhu C++ i treba izbegavati
- Deklaracije C++ biblioteke za U/I nalaze se u zaglavlju `<iostream>`
- Biblioteka `iostream` sadrži dve osnovne klase za U/I:
  - `istream` (apstrakcija ulaznog toka)
  - `ostream` (apstrakcija izlaznog toka)
- Iz navedenih klasa su izvedene klase `ifstream` i `ofstream`
  - objektu klase `ifstream`/`ofstream` može da se pridruži jedna datoteka za U/I
- Datotekama se pristupa isključivo preko objekata ovih klasa
  - odnosno metoda ili prijatelja ovih klasa

Standardni objekti i operacije za U/I

- U biblioteci `iostream` definisana su i dva globalna objekta:
  - objekat `cin` klase `istream`
    - koji je pridružen glavnom ulaznom toku sa uređaja (obično tastatura)
  - objekat `cout` klase `ostream`
    - koji je pridružen glavnom izlaznom toku na uređaj (obično ekran)
- Za klasu `istream` i sve ugrađene tipove, preklopljen je `operator>>` koji služi za ulaz podataka
  - `istream& operator>>(istream &is, T &t);`
  - gde je `T` neki ugrađeni tip objekta koji se čita
- Za klasu `ostream` i sve ugrađene tipove, preklopljen je `operator<<` koji služi za izlaz podataka:
  - `ostream& operator<<(ostream &os, const T &t);`
  - gde je `T` neki ugrađeni tip objekta koji se ispisuje

Korišćenje operatora >> i <<

- Operatorske funkcije `operator>>` i `operator<<`
  - vraćaju referencu na levi operand
    - posledica: može se vršiti višestruki U/I u istoj naredbi
  - operatori su asocijativni sleva-udesno
    - posledica: podaci se ispisuju/učitavaju u prirodnom redosledu
- Operatore >> i << treba koristiti za jednostavne U/I operacije
- Primer:

```
#include <iostream>
using namespace std;
void main () {
  int i;
  cin>>i;
  cout<<"i="<<i<<endl;
}
```

Preklapanje operatora >> i <<

- Korisnik može da definiše značenja operatora >> i << za svoje tipove
  - to se postiže definisanjem odgovarajućih globalnih funkcija prijatelja date klase
- Razlog zbog kojeg preklopljen operator ne može biti metod:
  - prvi operand je tipa `istream&` odnosno `ostream&`
- Primer za klasu `Kompleksni`:

```
#include <iostream>
using namespace std;
class Kompleksni{
  double real, imag;
  friend ostream& operator<<(ostream&, const Kompleksni&);
public: //... kao i ranije
};
ostream& operator<<(ostream &os, const Kompleksni &c) {
  return os<<"("<<c.real<<","<<c.imag<<")";
}
void main () {
  Kompleksni c(0.5,0.1);
  cout<<"c="<<c<<endl;  // ispisuje se: c=(0.5,0.1)
}
```

Operatori za nabrajanja

- Nabrajanja su celobrojni tip, ali podrazumevano su dozvoljene samo:
  - operacija dodele vrednosti istom tipu nabrajanja
  - operacija konverzije (kast) u celobrojnu vrednost – može i implicitno
    - osim za nabrajanja sa ograničenim dosegom
  - operacija konverzije iz celobrojne vrednosti – samo eksplicitno
- Dozvoljeno je preklapanje operatora koji se ne preklapaju kao metodi
  - nije dozvoljeno za: `= [] () -> (`*tip*`) new delete`
  - ugrađeni = i (*tip*) zadovoljavaju realne potrebe, ostali i nemaju mnogo smisla
- Primer:

```
enum Dani {PO,UT,SR,CE,PE,SU,NE};
inline Dani operator+(Dani d, int k)
  {k=(int(d)+k)%7; if(k<0) k+=7; return Dani(k);}
inline Dani& operator+=(Dani& d, int k){return d=d+k;}
inline Dani& operator++(Dani& d)
  {return d=Dani(d<NE?int(d)+1:PO);}
inline Dani operator++(Dani& d, int){Dani e(d);++d;return e;}
```