9. Preopterećenje operatora

Preopterećeni operator se definira kao funkcija i može se koristiti kao funkcija čije ime se sastoji od ključne riječi `operator` i oznake operatora čije se djelovanje definira tom funkcijom.

```
class Counter
{
private:
        int m_count;
public:
   //konstruktori i destruktor
   //...
   // pristupnici i mutatori
   int get_count() {return m_count;};

   // definiranje operatora ++, =, ==, < , <<

   Counter& operator = (const Counter &);

   Counter& operator++();            //prefiks inkrement
   Counter& operator++(int unused); //postfiks inkrement

   friend bool operator == (const Counter &c1, const Counter& c2);
   friend bool operator < (const Counter &c1, const Counter& c2);
   friend ostream& operator << (ostream &s, const Counter &c1);
};
```

Ovakova specifikacije klase `Counter` omogućuje izraze oblika:

```
Counter a;

++a;

Counter b = a++;

if(a==b) cout << "vrijednost oba brojaca je" << a;
```

Postupno ćemo upoznati pravila za definiranje operatorskog sučelja klase.

Preopterećenje operatora je važno ako se želi formirati klasu koja se može primijeniti u algoritmima iz standardne biblioteke STL.

9.1. Oblici operatorskih funkcija

Većina operatorskih funkcija može biti deklarirana kao:

- **nestatička članska funkcija ili kao**
- **nečlanska globalna funkcija.**

Primjerice, operator == može biti deklariran na više načina:

1. deklaracije operatora == kao članske nestatičke funkcije

```
class Counter
{
private:
   int m_count;
public:
//...
  int get_count() {return m_count;};
  Counter & operator =(const Counter &);
  bool operator ==(const Counter & d) const;
};
```

2. deklaracije operatora == kao globalne funkcije

```
bool operator == (const Counter& c1, const Counter& c2);
```

3. deklaracije operatora == kao prijateljske funkcije

```
class Counter
{
private:
   int m_count;
public:
//...
  int get_count() {return m_count;};
  Counter & operator =(const Counter &);
  friend bool operator == (const Counter& c1, const Counter& c2);
};
```

Izuzetak od ovog pravila su operatori

```
[], (), =, i ->
```

koji mogu biti deklarirani samo kao nestatičke članske funkcije (to osigurava da je njihov prvi operand Lvrijednost).

Isto vrijedi i za složene operatora tipa

```
+=, *=    itd.
```

Preopručeni oblici operatorskih funkcija.

Operator je simbolički označen s @, a objekti i klase s a,b i c:
a je objekt klase A, b je objekt klase B, a c je objekt klase C.

Izraz	Operator (@)	Članska funkcija	Globalna funkcija
`@a`	`+ - * & ! ~ ++ --`	`A::operator@()`	`operator@(A)`
`a@`	`++ --`	`A::operator@(int)`	`operator@(A, int)`
`a@b`	`+ - * / % ^ & | < > == != <= >= << >> && || ,`	`A::operator@(B)`	`operator@(A, B)`
`a@b`	`= += -= *= /= %= ^= &= |= <<= >>= [ ]`	`A::operator@(B)`	-
`a(b,c..)`	`()`	`A::operator()(B, C..)`	-
`a->b`	`->`	`A::operator->()`	-

Za aritmetičke i relacijske operatore prepručuje se deklaracija u obliku friend funkcija. To osigurava kompatibilnost s STL algoritmima koji su implementirani u standardnoj biblioteci.

9.2. Sučelje operatora

Sučelje operatora se opisuje tipom argumenata i tipom rezultata operatorske funkcije. Preporučljivo je koristiti sučelje opisano u tablici 2.

Tip operatorske funkcije	Vraća	Prijenos argumenata	const funkcija
Aritmetički (+, -, *, /)	vrijednost	const referenca	da
Pridjela vrjednosti (=, +=, -=)	referencu (*this)	const referenca	ne
Relacijski (<, ==)	vrijednost (bool)	const referenca	da
Unarni prefiks (++, --)	referencu (*this)	-	ne
Unarni postfiks (++, --)	vrijednost	vrijednost (int dummy)	ne
Unarni - i +	vrijednost	-	da
Indeksni []	referencu	vrijednost (int)	ne
Funkcijski ()	po volji	po volji	ne

Implementacija je u nadležnosti programera.
Pri izradi sučelja i implementacije treba voditi računa:

- **da se poštuju pravila asocijativnosti i prioriteta djelovanja operatora kakovi vrijede za proste skalarne tipove,**
  t2+t1/t2 se uvijek interpretira kao t2+(t1/t2)
- **da primjena operatora zadovoljava kontekst izraza u kojim oni koriste (glupo bi bilo dati značaj operatoru koji se razlikuje od značaja kojeg ima u C jeziku, iako je to dopušteno).**

Primjer: operator ==.

To je binarni i simetrični operator. Njime se uspoređuje dva istovrsna operanda, i dobija rezultat tipa bool (ako operandi nisu istog tipa , primjerice int i char tada kompajler vrši integralnu promociju tipa char i int). Simetričnost je u tome da operandi mogu zamijeniti mjesto. Ovo svojstvo implicira da se treba deklarirati friend operatorsku funkcija, jer ako se operator deklarira pomoću članske funkcije tada se dobija asimetričnost u izvršenju izraza

```
class Counter
{
private:
   int m_count;
public:
//...
// 1 - asimetrična deklaracija ne zadovoljava primjenu
// bool operator ==(const Counter &);

// 2 - simetrična deklaracija zadovoljava primjenu operatora
  friend bool operator == (const Counter& c1, const Counter& c2) const;
};

//// implementacija
bool operator == (const Counter &c1, const Counter& c2)
{
  return c1.m_count == c2.m_count;
}
```

Zašto nas ne zadovoljava deklaracija članskom funkcijom? Kompajler skriveno definira tu člansku funkciju tako da joj se pri pozivu šalje vrijednost `this` pokazivača. To otprilike odgovara C deklaraciji:

```
bool  Counter_operator_== (Counter *This,  const Counter& c2);
```

Očito je da imamo dva različita tipa za koje ne postoji standardna konverzija tipova, stoga nas ova implementacija ne zadovoljava.
Primjena deklaracije i implementacije kao `friend` funkcije osigurava ne samo simetričnost, već je njena primjena nužna ako se želi koristiti algoritme iz standardne STL biblioteke.

Primjer: operator pridjele vrijednosti =.

Njime se mijenja lijevi operand a desni ostaje nedirnut. Sučelje mora izrazitu tu činjenicu. Povoljna je deklaracija članskom funkcijom jer imamo asimetrično djelovanje operatora. Argument funkcije može biti const referenca. Funkcija mora vratiti referencu kako bi se mogli koristiti izrazi oblika:

```
a = b = c;
```

To je očito ako napišeno operatorski oblik:

```
a.operator=(b.operator=(c));
```

vidimo da b.operator=(c) mora vratiti referencu jer je to argument funkcije a.operator=(...).
Implementacija se realizira tako da funkcija `operator=` vraća dereferencirani `this` pokazivač, tj.

```
Counter& Counter::operator=(const Counter& c)
{
    m_count = c. m_count;
    m_mod = c. m_count;
    return *this;
}
```

Da zaključimo,

sučelje operatora se mora realizirati tako se osigura sličnost s djelovanjem tog operatora na standardne tipove. Implementcija je u nadležnosti programera.

9.3 Restrikcije

Mogu se preopteretiti slijedeći operatori:

```
new   delete     new[]      delete[]
+     -     *     /     %     ^     &     |     ~     !     =     <     >     +=    -=
*=    /=    %=     ^=    &=    |=    <<    >>    >>=   <<=   ==    !=     <=    >=
&&    ||    ++    --    ,     ->*   ->    ()    []
```

s tim da se operatori: +, -, * i & mogu koristiti u unarnom i binarnom obliku.

Ne smije se mijenjati značaj slijedećih operatora:

- **Točka operator .**
- **operator indirekcija pokazivača na člana klase .***
- **operator dosega ::**
- **operator sizeof**

zbog toga jer oni djeluju na ime a ne na objekt.

Ne smije se mijenjati značaj ternarnog uvjetnog operatora `?:`.

Ne smije se mijenjati značaj novih type cast operatora: `static_cast<>`, `dynamic_cast<>`, `reinterpret_cast<>`, i `const_cast<>` .

Ne smije se mijenjati značaj # i ## predprocesorkih simbola.

Za operatorske funkcije vrijede pravila naslijeđivanja kao i za ostale članske funkcije. Izuzetak je jedino operator pridjele vrijednosti, koji je uvijek definiran za svaku klasu (ako definiranje operatora pridjele vrijednosti ne izvrši programmer, tada to implicitno obavlja sam kompajler).

Pri definiranuju operatora uvijek mora bar jedan argument biti korisnički definiranog tipa. Izuzetak je jedino kod definiranja operatora new i delete. To osigurava da korisnik ne može mijenjati značaj izraza koji sadrže standardne tipove.

Ne smije se izmišljati nove operatore. Primjerice, nedozvoljeno je deklarirati znak @ kao operator:

```
void operator @ (int);  // nije dozvoljeno
```

Broj argumenata mora odgovarati broju argumenata koji se koriste u standardnim izrazima.

Za razliku od običnih funkcija, u operatorskim deklaracijama se ne smiju koristiti predodređeni parametri. Izuzetak je jedino operator ().

9.4. Postfiks i Prefiks Operatori

Operatori -- i ++ mogu se koristiti kao prefiks i postfiks operatori. Da bi se mogla razlikovati definicija postfiks i prefiks djelovanja ovih operatora, pravilo je da se postfiks operatori deklariraju s jednim argumentom tipa int, iako se taj argument ne koristi, a prefiks operatori se deklariraju bez argumenata. Primjerice, za klasu Counter se inkrement operator definira na slijedeći način:

```
class Counter
{    int m_count;
public:
    Counter & operator++();          //prefiks
    Counter operator++(int unused); //postfiks
    ...
};
// primjena
Counter c1, c2;
//prefiks: prvo inkrementira c2, i pridjeljujec1
c1 = ++c2;
//postfiks; prvo pridjeli c2 u c1 pa incrementiraj c2
c1 = c2++;

//implementacija
const Counter& Counter::operator++() // prefiks
{  ++m_count;;                       // inkrementiraj brojač
   return *this;                     // vrati njegovu referencu
}
const Counter Counter::operator++(int unused) // postfiks
{Counter temp(*this);                         // zapamti stanje u temp
    ++ m_count;;                              // inkrementiraj brojač
    return temp;                              // vrati temp objekt
}
```

Uočite da se u postfiks verziji vraća referenca, a u prefiks verziji se vraća vrijednost. Mogli smo obje verzije napisati tako da se vraća `void`. U tom slučaju inkrementiranje brojača bi se moglo izvršiti samo u prostim naredbama tipa:

```
++c;  ili c++;
```

Kada koristiti povrat referenca, a kada povrat vrijednosti objekta? U principu treba koristiti povrat reference (`*this`) uvijek kada je to moguće. U postfiks verziji operatora ++ to nije moguće jer `*this` predstavlja referencu inkrementirane vrijednosti, a funkcija mora vratiti prethodnu vrijednost objekta.

9.5. Korištenje operatora kao normalnih funkcija

```
Counter c1, c2;
bool equal;
c1.operator++(0);              // ekvivalentno: c1++;
c1.operator++();               // ekvivalentno: ++c1;
equal = operator==(c1, c2);    // ekvivalentno: equal = c1==c2;
```

9.6. Višestruko preopterećenje operatora

Normalne funkcije se mogu višestruko preopteretiti na način da se definira više verzija s različitim parametrima. Isto vrijedi i za operatore; preopterećenje može biti višestruko.

Tri verzije operatora == za klasu Counter

```
bool operator == (const Counter &c1, const Counter& c2);
bool operator == (const Counter &c, int i);
bool operator == (int i, const Counter& c);
```

U tom slučaju moguće je uspoređivati vrijednost brojača s cjelobrojnom varijablom.

Na samom početku ovog poglavlja definirana je `friend` funkcija

```
ostream & operator << (ostream & s, const Counter &c1)
{
 return s << c1.get_count(); // !ostream objekt vraća ref.
}
```

koja omogućuje da se operatorom << usmjerava vrijednost brojača na izlazni tok. Primjerice,

```
Counter c;
cout << c;
```

ispisuje vrijednost brojača na standardnom izlazu. Funkcija vraća referencu na ostream objekt. To je nužno za korištenje izraze oblika

```
Counter a, b, c;
cout << a << b << c;
```

Operatori << i >> su vjerojatno najopterećeniji operatori, jer se definiraju u većini klasa koje komuniciraju s ulazno/izlaznim tokovima.

9.7. Preopterećenje operatora pobrojanih tipova

Pobrojani tip obuhvaća skup konstanti. Cilj nam je da se pomoću operatora ++ i -- može iterativno mijenjati vrijednosti iz definiranog skupa konstanti. Primjer:

```
enum Days{
  Monday, Tuesday,  Wednesday,   Thursday,
  Friday,   Saturday,   Sunday
};

Days& operator++(Days& d, int)  // postfix ++
{
  if (d == Sunday)
     return d = Monday;
  int temp = d;         //pretvori u int
  return d = (Days) (++temp);
}

int main()
{
 Days day = Monday;
 for (;;) //prikaz  days kao cjelobrojnioh vrijednosti
 {
   cout<< day <<endl;
   day++;
   if (day == Sunday)
     break;
 }
 return 0;
}
```

Ako želimo da se simboličke konstante ispisuju svojim imenom, a ne numeričkom vrijednošću, tada se može preopteretiti << operator;

```
ostream& operator<<(ostream& os, Days d)
{
  switch(d){
  case Monday:    return os<<"Monday";
  case Tuesday:   return os<<"Tuesday";
  case Wednesday: return os<<"Wednesday";
  case Thursday:  return os<<"Thursady";
  case Friday:    return os<<"Friday";
  case Saturday:  return os<<"Saturday";
  case Sunday:    return os<<"Sunday";
  default:        return os<<"Unknown";
  }
}
```

9.8 Klasa intArray i preopterećenje indeksnog operatora []

Kada klasa služi za apstrakciju niza elemenata poželjno je definirati indeksni operator `[]` pomoću kojeg se pristupa elementima niza. Poželjno je uvijek definirati dvije verzije operatora `[]`: const verziju i ne-const verziju. Primjerice, definirat ćemo klasu `iarray` koja nam može poslužiti za rad s cjelobrojnim dinamičkim nizovima.

```
const int def_iarray_size = 5;

class iarray  // niz cjelobrovnih elemenata
{
int *m_arr;    // pokazivač na memoriju koja sadrži niz
int m_size;    // broj elemenata niza

void init(const int *array, int size); // pomoćna privatna funkcija

public:
     iarray(int def_size = def_iarray_size)
     {   init((const int *) 0, def_size);
     }

     iarray(const iarray& rhs)    // kopirni konstruktor
     {  init(rhs.m_arr, rhs.m_size);
     }

     iarray(const int *array, int size) // konstruktor pomoću postojećeg
     {     init(array, size);           // int * niza
     }

     ~iarray(void) { delete [] m_arr; } // destruktor
```

```
// pridjela vrijednosti
iarray& operator=(const iarray&);

int size() const {return m_size;}

// indeksni operator - const i non-const verzija

int& operator[](int index)  { return m_arr[index]; }

const int& operator[](int index) const    { return m_arr[index]; }

};

void iarray::init(const int *array, int size)
{
// alociraj memoriju veličine m_size cijelih brojeva
   m_size = size;
   m_arr = new int [size];

//inicijaliziraj niz

   if (array != 0)
   for (int index=0; index<size; index++)
   {
        m_arr[index] = array[index];
   }
}
```

```
iarray& iarray::operator=(const iarray& rhs)
{
  if (this != &rhs)
  {
    // ako se objekti razlikuju
    // dealociraj postojeću memoriju

    delete [] m_arr;

    // zatim alociraj i kopiraj rhs to lhs

    init(rhs.m_arr, rhs.m_size);

   }

   return *this;            // vrati referencu
}
```

Primjena:

```
iarray  a;
a[0] = 1;
for(int i = 1;  i < a.size(); i++)
    a[i] = 7;
```

Zašto je ovo bolji način rada s nizovima nego je to s prostim nizovima u C jeziku.

9.9 Operatori pretvorbe tipova

Opći oblik deklaracije operatora pretvorbe tipova je:

```
class Aclass {
    ....
public:
    operator tip () {...};  // operator pretvorbe u tip
}
```

Deklaracija operatora pretvorbe tipova se razlikuje od uobičajenih deklaracija u dvije stvari.

1. **ne deklarira se vrijednost funkcije, već deklaracija započima s riječju `operator`,**
2. **funkcija pretvorbe parametara se definira bez parametara.**

Primjerice, može nam biti korisno da se u nekim izrazima objekti tipa `Counter` tretiraju kao cjelobrojne varijable koje sadrže vrijednost brojača. To smo do sada realizirali na način da smo koristili člansku funkciju `get_count()`.

```
int square(int i) {return i*i;}

int main() { Counter c; c++;

int i = square(c.get_count());
....
```

Ako bi u klasi Counter definirali operator pretvorbe u tip `int`,

```
class Counter
{
   int m_count;
public;
    ...
    operator int () {return m_count;}
};
```

tada bi bilo moguće prethodni insert programa zapisati u obliku:

```
int square(int i) {return i*i;}

int main() {
Counter c;
c++;
int i = square(c); //c se tretira kao int
....
```

jer kompajler pri pozivu funkcije `square` raspolaže s operatorom pretvorbe objekta klase `Counter` u vrijednost tipa `int`.

Ova tehnika programiranja je dosta kritizirana u akademskim krugovima, jer se njome zaobilazi stroga kontrola tipova.

9.10. Funkcijski objekti

Funkcijski objekt se implementira kao klasa u kojoj je definiran operator poziva funkcije ().

```
class Fclass {
    ....
public:
    tip  operator () (parametri);
}
```

Objekti ove klase mogu se upotrebljavati kao funkcije:

```
Fclass  Fobj;
Fobj(argumenti ..)  // funkcijski objekt – vrši poziv funkcije
```

Regularne funkcije mogu imati proizvoljan broj argumenata, stoga je operator () iznimka među operatorskim funkcijama, jer se može definirati s proizvoljnim brojem argumenata uključujući i predodređene parametre

U slijedećem primjeru definirana klasa `increment`. Pomoću nje je definiran funkcijski objekt `incr`. Koristimo ga kao običnu funkciju (vidimo da `incr(n)` predstavlja poziv funkcije definirane u klasom `increment`).

```
class increment
{
  // generic increment function
  public :
     int operator() (int x) const { return ++x;}
};
```

```
void f(int n, const increment& incr)
{   cout << incr(n) << endl;
}

int  main()
{
  int i = 0;
  increment incr;
  f(i, incr);  // ispisuje 1
  return 0;
}
```

9.11. Zaključak

Koncept preopterećenja operatora je temeljni element implementacije apstraktnih tipova podataka.

- **Preopterećeni operatori uzrokuju iste efekte kao članske i globalne funkcije: mogu se naslijediti i mogu se višestruko preopteretiti.**
- **Preopterećeni operator može imati fiksni broj parametara, s tim da se ne smije deklarirati predodređene parametre (izuzetak su funkcijski objekti).**
- **Pravila asocijativnosti i prioriteta se preuzimaju iz pravila koji vrijede za izraze s standardnim tipovima.**
- **Preporučuje se da kontekst primjene preopterećenih operatora odgovara kontekstu primjene standardnih operatora.**
- **Specijalni tip preopterećenih operatora su operatori pretvorbe tipova. Za njih vrijede posebna pravila; ne vraćaju vrijednost i ne koriste argumente.**