Konstantni objekti

1

- konstantni objekti so objekti, ki jim ne moremo spremeniti vrednosti podatkov
- konstantni objekt definiramo na naslednji način:

```
const CRazred ime;
const CRazred ime(parametri);
```

privzeti konstruktor

konstruktor s parametri

- v konstantnem objektu lahko kličemo samo konstantne metode

Konstantne metode

2

- konstantne metode so tiste metode, ki ne spremenijo podatkov objekta
- konstantno metodo deklariramo tako, da za glavo metode dodamo rezervirano besedo **const**
- vse metode, katerih namen ni spreminjanje objekta, deklariramo kot konstantne metode
  - če v konstantni metodi spreminjamo objektne spremenljivke, prevajalnik zazna napako

PRIMER: kompleksna števila

3

```
class CKompleksno {
  private:
    double re; // realni del števila
    double im; // imaginarni del števila

  public:
    CKompleksno();
    CKompleksno(double re, double im);

    void izpis() const;
    CKompleksno pristej(const CKompleksno &x) const;
};
```

funkciji naj ne bi spreminjali objekta, zato ju definiramo kot konstantni

PRIMER: kompleksna števila

4

```
CKompleksno::CKompleksno() {
  re = 1;
  im = 0;
}


CKompleksno::CKompleksno(double re, double im) {
  this->re = re;
  this->im = im;
}
```

PRIMER: kompleksna števila

5

```
void CKompleksno::izpis() const {
  // izpišimo v obliki a + b*i
  cout << re << " + " << im << "i";
}

CKompleksno CKompleksno::
           pristej(const CKompleksno &x) const {
  // izračunamo vsoto dveh kompleksnih števil
  CKompleksno vsota;
  vsota.re = re + x.re;
  vsota.im = im + x.im;
  return vsota;
}
```

PRIMER: kompleksna števila

6

```
#include "Kompleksno.h"
...
int main() {
  CKompleksno   a,         // a = 1
                b(1,2),    // b = 1 + 2i
                c;
  c = a.pristej(b);  // c = a+b;
  c.izpis();         // cout << c;

  return 0;
}
```

ne moremo, ni definiran operator + za razred CKompleksno

ne moremo, ni definiran operator << za razred CKompleksno

Prekrivanje operatorjev

7

- spoznali smo že: C++ omogoča prekrivanje funkcij
- določene operatorje želimo prekriti
  - definirati želimo operatorje za svoje razrede
  - operator lahko prekrijemo, ko je vsaj eden od operandov uporabniško definiran razredni tip
  - ne moremo spreminjati prioritete, asociativnosti in števila operandov
  - ne moremo vpeljevati novih operatorjev
- prekrijemo lahko večino operatorjev
  - ne moremo prekriti naslednjih operatorjev:
    . :: ?: sizeof

Prekrivanje operatorjev

8

- operator lahko prekrijemo na enega izmed dveh načinov:
  - kor razredno funkcijo

```
tip_rezultata CRazred::operator@(parametri) {
}
```

    - `tip_rezultata` – tip rezultata operatorja
    - `CRazred` – za kateri razred definiramo operator
    - `operator` – rezervirana beseda
    - `@` - simbol operatorja, ki ga prekrivamo
    - `parametri` – parametri, ki jih operator potrebuje
  - kot globalno prijateljsko funkcijo (več v nadaljevanju)

Prekrivanje unarnih operatorjev

9

- unarni operatorji delujejo nad enim operandom
- primeri unarnih operatorjev: - (kot predznak), ++, --

PRIMER:

```
// operator- (kot predznak) za kompleksna števila
// v deklaracijo razreda dodamo
CKompleksno operator-() const;

// funkcijo definiramo
CKompleksno CKompleksno::operator-() const {
  return CKompleksno(-re, -im);
}
```

Prekrivanje binarnih operatorjev

10

- binarni operatorji delujejo nad dvema operandoma
- uporabljamo jih v obliki a @ b, @ - simbol operatorja
- primeri binarnih operatorjev:
  - +, - (kot odštevanje), *, /, % - aritmetični operatorji
  - ==, != - relacijski operatorji
  - =, <<, >>
- binarnega operatorja ne moremo definirati kot razredno funkcijo, kadar operand na levi strani operatorja, ni razrednega tipa, za katerega definiramo operator
  - v tem primeru ga definiramo kot prijateljsko funkcijo

Prekrivanje operatorjev za aritmetične operacije

11

- primer za operator + za kompleksna števila
- namesto funkcije

```
CKompleksno pristej(const CKompleksno &x) const;
```

- prekrijemo operator +

```
CKompleksno operator+(const CKompleksno &x) const;
```

Prekrivanje operatorjev za aritmetične operacije

12

- definicija funkcije je enaka:

```
CKompleksno CKompleksno::
     operator+(const CKompleksno &x) const {
  // izračunamo vsoto dveh kompleksnih števil
  CKompleksno vsota;
  vsota.re = re + x.re;
  vsota.im = im + x.im;
  return vsota;
}
```

Prekrivanje operatorjev za relacijske operacije

13

- primer za operator == za kompleksna števila
  - kompleksni števili sta enaki, kadar imata enak realni in imaginarni del
- prekrijemo operator ==

```
bool operator==(const CKompleksno &x) const;
```

- definicija te funkcije:

```
bool CKompleksno::operator==(const CKompleksno &aK)
  const {
  if (re == aK.re && im == aK.im)
      return true;
  else
      return false;
}
```

Prijateljske funkcije

14

- prijateljske funkcije so funkcije, ki lahko dostopajo do zasebnih spremenljivk in metod razreda
- prijateljske funkcije niso razredne funkcije
- moramo jih napovedati v razredu, do katerega imajo dostop
  - pred deklaracijo funkcije napišemo rezervirano besedo **friend**

Prijateljske funkcije - primer

15

- za izpis kompleksnega števila želimo uporabiti operator <<
- npr. cout << x; // kjer je x tipa CKompleksno
- operator << je binarni operator
  - levi operand je cout (ni tipa CKompleksno) zato ga ne moremo definirati kot razredno funkcijo
  - definiramo ga kot prijateljsko funkcijo
- v deklaraciji razreda napovemo prijateljsko funkcijo
  - `friend ostream& operator<<(ostream &out, CKompleksno &x);`

Prijateljske funkcije - primer

16

```
friend ostream& operator<<(ostream &out,
  CKompleksno &x);
```

- prijateljska funkcija
- vrne referenco na objekt za izpisovanje, to omogoča zaporedno uporabo operatorja
- ime operatorja
- levi operand, izhodni podatkovni tok, npr. cout
- desni operand – kaj izpisujemo – kompleksno število (razredni tip, za katerega prekrivamo operator)

Prijateljske funkcije - primer

17

- prijateljsko funkcijo definiramo izven razreda:

```
ostream& operator<<(ostream &out,
                    CKompleksno &x){
  out << x.re << " + " << x.im << "i";
  return out;
}
```

čeprav ni razredna funkcija, lahko dostopamo do zasebnih komponent razreda, saj je to prijateljska funkcija razreda CKompleksno

Celoten primer za kompleksna števila (kompleksno.h)

18

```
#include <iostream>
using namespace std;

class CKompleksno {
  private:
    double re; // realni del števila
    double im; // imaginarni del števila
  public:
    CKompleksno();
    CKompleksno(double re, double im);
    ~CKompleksno();
    CKompleksno operator-() const;
    CKompleksno operator+(const CKompleksno &x) const;
    bool operator==(const CKompleksno &x) const;
    friend ostream& operator<<(ostream &out, CKompleksno &x);
};
```

Celoten primer za kompleksna števila (kompleksno.cpp)

19

```
#include "kompleksno.h"

CKompleksno::CKompleksno() {
  re = 1;
  im = 0;
}

CKompleksno::CKompleksno(double re, double im) {
  this->re = re;
  this->im = im;
}

CKompleksno::~CKompleksno(){
}
```

Celoten primer za kompleksna števila (kompleksno.cpp)

20

```
CKompleksno CKompleksno::operator-() const {
  return CKompleksno(-re, -im);
}

CKompleksno CKompleksno::operator+(const CKompleksno &x) const {
  CKompleksno vsota;
  vsota.re = re + x.re;
  vsota.im = im + x.im;
  return vsota;
}

bool CKompleksno::operator==(const CKompleksno &aK) const {
   if (re == aK.re && im == aK.im)
         return true;
   else
         return false;
}
```

Celoten primer za kompleksna števila (kompleksno.cpp)

21

```
// definiramo prijateljsko funkcijo
// NI razredna funkcija!!!
ostream& operator<<(ostream &out, CKompleksno &x){
   out << x.re << " + " << x.im << "i";
   return out;
}
```

Celoten primer za kompleksna števila (main.cpp)

22

```
#include <iostream>
#include "kompleksno.h"
using namespace std;

int main()
{
    CKompleksno a, b(1,2), c;
    c = a+b;
    a = -c;
    cout << "c = " << c << endl;
    cout << "a = " << a << endl;
    if (a==c)
      cout << "a in c sta enaki.";
    else
      cout << "a in c nista enaki.";
    return 0;
}
```