FUNKCIJŲ PERKROVIMAS

Kaip jau minėta perkrovimas – tai vienas iš polimorfizmo atvejų. C++ kalboje naudojamas funkcijų ir operacijų (operatorių) perkrovimas. Funkcijų perkrovimas – kai eilė funkcijų turi tą patį vardą bet skirtingą skaičių argumentų arba jų tipą. Tokios funkcijos vadinamos perkrautomis (angl. *overloaded*).

Perkrovimas naudingas tuomet, kai siekiama supaprastinti programą t.y. funkcijoms, atliekančioms panašius veiksmus su skirtingais argumentų sąrašais, suteikti tokius pat pavadinimus. Kuri iš perkrautų funkcijų bus iškviesta, sprendžia kompiliatorius.

Pavyzdys 3.9. Funcijos *abs* perkrovimas.

```
# include <iostream.h>
int abs (int a);
float abs (float b);
double abs (double c);
void main ()
{  int x = 255; float y = -1.2f; double  z = 155.0;
   cout << abs (x)<< endl;
   cout << abs (y)<< endl;
   cout << abs (z)<< endl;
}
int abs (int x)
{
        return x < 0 ? -x:x;
}
float abs (float y)
{
        return y < 0 ? -y:y;
}
double abs (double z)
{
        return z < 0 ? -z:z;
}
```

Funkciją *abs* galima būtų apskaičiuoti ir pasinaudojus bibliotekinėmis funkcijomis, tačiau, tokiu atveju, reikės skirtingų funkcijų, pvz., *abs(), fabs(), labs()* ir pan.

Kitas pavyzdys − funkcijos perkrovimas naudojant ne tik skirtingą argumento tipą, bet ir skirtingą argumentų skaičių.

Pavyzdys 3.10. Simbolių sekos išvedimo funkcijos perkrovimas.

```
# include <iostream.h>
void repchr();
void repchr(char);
void repchr(char, int);
void main ()
{ repchr();
  repchr('*');
  repchr('+', 25);
}

void repchr()
{ for (int i=0; i<50; i++)
        cout<< '–';
        cout<<endl;}
void repchr(char ch)
{ for (int i=0; i<50; i++)
        cout<< ch;
        cout<<endl;}
void repchr(char ch, int n)
{ for (int i=0; i<n; i++)
        cout<< ch;
        cout<<endl;}
```

Programos vykdymo rezultatai:

```
--------------------------------------------------
**************************************************
++++++++++++++++++++++++++
```

Perkraunamos funkcijos pasižymi tokiomis ypatybėmis:

- bet kurios dvi perkraunamos funkcijos turi turėti skirtingus argumentų sąrašus;
- perkraunamos funkcijos, turinčios tuos pačius argumentų sąrašus, negali skirtis tik grąžinamos reikšmės tipu;

- klasės metodai nelaikomi perkrautais, jei skirtumas yra tik tas, kad vienas iš jų apibrėžtas kaip **static;**
- **typedef** operatorius neįtakoja perkrovimą, nes įveda tik tipų sinonimus bet ne naujus tipus, pvz., kompiliatorius traktuoja žemiau esančias funkcijas kaip vienodas ir jei jos bus paskelbtos klasėje kartu – bus klaida:

  ```
  typedef char* ptr;

  void SetVal (char* s);
  void SetVal (ptr s);
  ```

- visi **enum** tipai traktuojami kaip skirtingi;
- vienmačių masyvų atveju tipas *masyvas* ir tipas *rodykle* traktuojami kaip vienodi, tačiau daugiamačių masyvų atveju antras ir tolimesni indeksai gali būti naudojami skelbiant perkraunamas funkcijas, pvz.:

  ```
  void SetVal (char sm[]);

  void SetVal (char sm[][5]);
  ```

Perkraunamų funkcijų vardų dekodavimas.

Kompiliuojant programą, perkraunamos funkcijos gauna skirtingus vardus. Tokie vardai generuojami kompiliatoriaus atsižvelgiant į programuotojo apibrėžtą funkcijos vardą, argumentų skaičių ir jų tipą. Pvz., duota klasės apibrėžtis:

```
class AnyClass
{
    public
        void SetVal ();
        void SetVal (int);
        void SetVal (int, int);
        void SetVal (int, int, int);
        void SetVal (int&, int, int);
}
```

Tokiu atveju, pvz., *Borland C++ Builder* taip dekoduos perkraunamų funkcijų vardus:

```
@AnyClass@SetVal@qv
@AnyClass@SetVal@qi
@AnyClass@SetVal@qii
```

@AnyClass@SetVal@qiii
@AnyClass@SetVal@qriii

Kadangi funkcijų dekodavimas neapima funkcijos gražinamos reikšmės, būtent todėl negalima perkrauti funkcijų, turinčių tuos pačius argumentų sąrašus, bet skirtingus grąžinamos reikšmės tipus.
Pastaba. Skirtingi kompiliatoriai nauduoja skirtingas dekodavimo sistemas.

Perkraunamų funkcijų vardų matomumas

Kompiliatorius nustato ir apdoroja perkraunamų funkcijų skelbimus toje pačioje matomumo srityje. Todėl, jei, pvz., išvestinėje klasėje būtų paskelbta funkcija tokiu pat vardu kaip bazinėje klasėje, išvestinės klasės funkcija tik perdengs (nuslėps) bazinės klasės funkciją, bet ne perkraus jos. Taip atsitinka dėl to, kad šių funkcijų matomumo sritys yra skirtingos. Analogiškai, nebus perkrautos vienodus vardus turinčios funkcijos, kurios paskelbtos failo matomumo srityje (t.y. globali funkcija) ir bloko matomumo srityje (lokali funkcija).

Pavyzdys 3.10a. Funkcijų vardų matomumas.

```
# include <iostream.h>

void funkcija(int i)
{
        cout<< “Globalios funkcijos iškvietimas ”
        cout<<i<<endl;
}
void funkcija(char* str)
{
        cout<< “Lokalios funkcijos iškvietimas ”
        cout<<str<<endl;
}
main()
{
  extern void funkcija(char*);
  funkcija(10);            //klaida, nes iškviečiama nebematoma funkcija
  funkcija(“Eilute”);    //klaidos nėra, lokali funkcija perdengs globalią
  return 0;
  }
```

Klasių metodų perkrovimas

Perkraunami klasės metodai gali būti apibrėžti skirtingose klasės prieigos srityse (pvz. vienas metodas *private*, o kitas – *public* srityje). Vieta, kur metodas apibrėžtas, nekeičia metodų perkrovimui galiojančių taisyklių.

Pavyzdys 3.11. Klasių metodų perkrovimas

```
class AnyClass
{  private :
    int Func (int a) ;
      { …. }
    public:
    AnyClass();
    double Func (double b, char* c) ;
     { ….}
};
main()
{ AnyClass * ptr = new AnyClass;
  ptr -> Func(104);              //klaida, nes iškviečiama uždara funkcija
  ptr -> Func(12.2, “Eilute”); //klaidos nėra, nes funkcija atvira
  delete ptr;
  return 0;
};
```

Konstruktorių perkrovimas

Perkrovimas dažniausiai naudojamas kuriant būtent perkrautus konstruktorius. Taip programuotojui suteikiama galimybė kurti įvairius klasės atstovus – objektus, naudojant skirtingus argumentų sąrašus.
Pastaba. Destruktorių perkrauti negalima.

Pavyzdys 3.12. Konstruktorių perkrovimas

```
Class Rect
{ private:
        int x,y,w,h;
  public:
        Rect()  {x=y=w=h=0;}
        Rect(int a, int b)  {x=a; y=b; w=h=10;}
        Rect(int a, int b, int c, int d)  {x=a; y=b; w=c; h=d;}
        Rect(const Rect&);
```

```
        Rect(const Rect&, int =0, int  =0);
};
Rect::Rect(const Rect& rc)
{       x=rc.x;
        y=rc.y;
        w=rc.w;
        h=rc.h }
Rect::Rect(const Rect& rc, int _x, int _y)
{       x=_x;
        y=_y;
        w=rc.w;
        h=rc.h; }
void main ()
{
        Rect ab;
        Rect bc(10, 20);
        Rect rc(3,4,5,6);
        Rect newrc(rc, 14, 34);
}
```

Klasė *Rect* turi kelis konstruktorius, kas suteikia galimybę kurti įvairius geometrinius objektus (stačiakampius). Pabrėžtina, kad perkrautas kopijavimo konstruktorius

```
        Rect(const Rect&, int =0, int  =0);
```

tik dalį parametrų kopijuoja, o kitus inicializuoja kitaip. Tokiu būdu naudojant kopijavimo konstruktorius, papildomais parametrais galima gana lanksčiai inicializuoti objektus.

Kopijavimo konstruktoriaus ypatybės

Kopijos (kopijavimo) konstruktorius naudojamas kuriant naują objektą kaip esamo objekto kopiją. Jei kopijavimo konstruktorius nėra apibrėžtas programuotojo, kompiliatorius sukuria kopijavimo konstruktorių pagal nutylėjimą. Toks konstruktorius sukuria naują objektą ir atlieka tiesioginį pobitinį egzistuojančio objekto kopijavimą. Tai vyksta ir tada, kai funkcijos argumentu yra objektas, t.y. vyksta funkcijos argumento (objekto) perdavimas pagal reikšmę. Tokiu būdu pobitinė (tiksli) egzistuojančio objekto kopija ir perduodama funkcijos parametrui.

Tačiau, jei objektas turi savyje nuorodas arba rodyklės, tai objekto kopija turės rodykles į tą pačią atminties sritį ir pradinio objekto atminties sritis gali būti pakeista. Be to, kai funkcija baigs savo darbą, destruktorius panaikins objekto kopiją, kas taip pat gali pakeisti pradinio objekto atminties sritį.

Jei funkcijos gražinama reikšmė yra objektas, prieš išeinant iš jos kompiliatorius sukurs laikiną objektą funkcijos gražinamos reikšmės saugojimui. Kai funkcija baigs darbą, šis objektas išeis iš matomumo srities ir kompiliatorius iškvies jam destruktorių. Jei šis destruktorius išvalys objektui skirtą dinaminę atmintį, įvyks klaida.

Taigi, kompiliatoriaus sukuriamas kopijavimo konstruktorius pagal nutylėjimą sukelia problemų. Todėl, jei objektas turi savyje nuorodas arba rodyklės, tam, kad būtų išvengta kreipinių į tą pačią atminties sritį, programuotojas turi aprašyti savo kopijavimo konstruktorių. Toks konstruktorius taip pat būtinas, kai klasėje perkraunamas priskyrimo operatorius. Kai programuotojas pateikia savo kopijavimo konstruktorių, būtent jis bus naudojimas inicializuojant vieną objektą kitu, tame tarpe ir perduodant objektą funkcijai.

C++ pateikia programuotojui įvairias kopijavimo konstruktorių formas. Kadangi viena iš tokių formų naudoja priskyrimo operatoriaus ženklą, reikėtų skirti kopijavimo konstruktorių nuo priskyrimo operatoriaus:

```
AnyClass obj1;
AnyClass func();        //funkcijos skelbimas

//Čia kopijavimo konstruktorius inicializuoja obj2 naudojant obj1
AnyClass obj2(obj1);

//Taip pat kopijavimo konstruktorius,  kita objekto kopijavimo forma
AnyClass obj2 = obj1;

//Objekto gražinimas iš funkcijos naudojant kopijavimo konstruktorių
obj1 = func();


//Tai priskyrimo operacija
obj2 = obj1;  //
```

Tipų suvedimo (transformavimo) konstruktoriai

Kaip jau minėta, konstruktorių perkrovimas suteikia programuotojui galimybę kurti įvairius klasės atstovus – objektus, naudojant skirtingus argumentų sąrašus. Kita konstruktorių perkrovimo suteikiama galimybė – aprūpinti programuotoją tipų transformavimo priemonėmis.

Tarkime, yra apibrėžta funkcija, kurios argumentas – nuoroda į klasės AnyClass objektą

```
void funkcija(AnyClass&  rOb);
```

Jei kreipinyje į šią funkciją bus nurodytas kito tipo argumentas, tai kompiliatorius nežinos kaip transformuoti tipą ir įvyks klaida. Jei yra poreikis nurodyti kompiliatoriui kaip transformuoti kokį nors tipą T į tipą AnyClass, reikia apibrėžti klasės konstruktorių su vienu parametru T tipo. Pvz., jei klasėje apibrėžtas konstruktorius

```
AnyClass(T  t);
```

kompiliatorius interpretuoja

```
AnyClass ob(t)   kaip   AnyClass ob = AnyClass(t);
```

Tai reiškia, kad kreipinys į funkciją

```
funkcija(t)
```

bus traktuojamas kaip

```
funkcija(AnyClass(t));
```

Tai neišreikštinis tipo transformavimas, kuriam įgyvendinti kompiliatorius sukuria laikiną klasės AnyClass objektą, perduodamas jam funkcijos parametrą.

Pavyzdys 3.12a. Tipų transformavimo konstruktoriai

```
class AnyClass
{
   int x;
   public:
   AnyClass();          //Konstruktorius pagal nutylėjimą
   AnyClass(int);       //Tipo int transformavimo konstruktorius
   AnyClass(long);      //Tipo long transformavimo konstruktorius
   AnyClass(double);  //Tipo double transformavimo konstruktorius
};
   AnyClass :: AnyClass();  {x = 0;}
   AnyClass :: AnyClass(int  _x)  {x = _x;};
```

```
   AnyClass :: AnyClass(long _x) {x = _x;};
   AnyClass :: AnyClass(double _x) {x = _x;};

void funkcija(AnyClass& rOb}
{
//funkcijos blokas
}

main()
{
  funkcija(10);              //int transformuojamas į AnyClass
  funkcija(10L);             //long transformuojamas į AnyClass
  funkcija(10.0);            //double transformuojamas į AnyClass
  AnyClass ob;
  funkcija(ob);              // transformavimas nevykdomas
  return 0;
};
```

Analogiškai galima sudaryti konstruktorių, kuris transformuotų vieną klasę į kitą.

Baziniai (rezervuoti) žodžiai *overload* ir *explicit*

Bazinis žodis ***overload*** buvo naudojamas perkraunamų funkcijų apibrėžimui senesniuose C++ versijose: overload <funkcijos vardas>. Naujuose kompiliatoriuose žodis nenaudojamas.

Skelbiant konstruktorių klasėje gali būti naudojamas bazinis žodis ***explicit***:

```
class T
{
   public:
   explicit  T(int);
   explicit  T(double);
  {
       //konstruktoriaus blokas
   }
 };

 T::T(int)
{
```

```
        // ....
}
```

Konstruktorius, kuris skelbiamas su baziniu žodžiu ***explicit*** nedalyvauja neišreikštiniame tipų transformavime ir gali būti naudojamas tik tada, kai argumentų tipų transformavimas nereikalingas. Pvz.:

```
void f(T) {};       //funkcijos apibrėžimas
void g(int i)
{
 f(i);  //klaida, nes uždraustas neišreikštinis int transformavimas į tipą T
}
void h()
{
 T  ob(10);         //teisingai
}
```

Pastaba. Konstruktoriai su keliais parametrais bet kuriuo atveju nedalyvauja neišreikštiniame tipų transformavime, todėl su jais žodis ***explicit*** nenaudojamas.

Perkrautos funkcijos adresas

Perkrautos funkcijos adresą galima rasti nurodžius funkcijos vardą be argumentų sąrašo ir be skliaustų dešinėje pusėje priskyrimo operatoriaus. Kairėje priskyrimo pusėje turi būti rodyklė į atitinkamo tipo funkciją.

Pavyzdys 3.13. Perkrautos funkcijos adreso radimas.

```
int Func(int i, int j);
int Func(long l);
int (*ptr)(int, int) = Func;        //randamas pirmos funkcijos adresas
ptr(10,20);                         //iškviečiama funkcija
```

Kompiliatorius nustato kokią funkciją reikia iškviesti pagal bendras perkrovimo taisykles, t.y. ieškoma funkcija, kuri tiksliai atitinka rodyklės užduotą funkcijos tipą. Jei tikslaus atitikmens nepatikta (t.y. adresas nustatomas nevienareikšmiai), generuojama klaida.

OPERATORIŲ PERKROVIMAS

C++ ir kitose programavimo kalbose naudojami operatoriai pasižymi polimorfizmu. Pvz., operatorius + gali būti taikomas tiek **int**, tiek **float** tipo kintamiesiems, nors sudėties veiksmai procesoriuje iš tikrųjų bus atliekami skirtingai.

Programuotojui taip pat suteikiama galimybė perkrauti operatorius darbui su naujai apibrėžtais duomenų tipais (klasėmis). Operatorius C++ kalboje galima perkrauti tiek globaliai, tiek klasės ribose. Perkraunami operatoriai aprašomi naudojant bazinį žodį **operator**. Pvz., norint perkrauti sudėties operatorių nurodoma **operator+**, o kartu su priskyrimu: **operator+=**

Operatorių veikimo keitimas yra vadinamas perkrovimu arba perdengimu. Bendra operatoriaus perkrovimo (perdengimo) sintaksė :

<tipas> **operator**<operatoriaus_simbolis> (<param_tipas> parametras);

Iš tikrųjų, operatoriai – tai funkcijos, todėl operatorių perkrovimas siejamas su funkcijų perkrovimu. Perkrauti galima beveik visus C++ operatorius, išskyrus: **. .* :: ?: # ##**

Operatorinė funkcija dažnai apibrėžiama klasės viduje kaip klasės metodas.

Pavyzdys 3.14. Operatorių perkrovimas klasėje.

```
#include <iostream.h>
class Taskas
{
        int x, y;
   public:
        Taskas() { x=0, y=0; }
        Taskas(int _x, int _y) { x=_x, y=_y; }
        void GautiKoord(int& _x, int& _y) {  _x=x, _y=y;}
        Taskas operator+(Taskas& ob);
        Taskas operator-(Taskas& ob);
};

Taskas
Taskas :: operator+ ( Taskas&  ob)
{
      Taskas  tempOb;
       tempOb.x = x + ob.x;
```

```
        tempOb.y = y + ob.y;
        return tempOb;
};

Taskas
Taskas :: operator- ( Taskas&  ob)
{
       Taskas  tempOb;
        tempOb.x = x – ob.x;
        tempOb.y = y – ob.y;
        return tempOb;
};

main ()
{
    int x, y;
    Taskas  PtA(10,20), PtB(3,8), PtC;
    PtC = PtA + PtB;                //Perkrauto sudėties operatoriaus iškvietimas
    PtC.GautiKoord(x,y);
    cout << "PtC.x =" << x <<  "  PtC.y =" << y << endl;
    PtC = PtA – PtB;                //Perkrauto atimties operatoriaus iškvietimas
    PtC.GautiKoord(x,y);
    cout << "PtC.x =" << x <<  "  PtC.y =" << y << endl;
    return 0;
 }
```

Šios programos vykdymo rezultatas:

```
PtC.x =13    PtC.y =28
PtC.x =7     PtC.y =12
```

Čia naudojamas parametrų perdavimas pagal adresą (nuoroda), o gražina operatorinės funkcijos klasės *Taskas* objektą. Tai leidžia sudaryti, pvz., tokias sudėtines išraiškas:

```
PtC = PtA + PtB – PtC;
(PtA + PtB).GautiKoord(x,y);
```

Sudėties operatoriaus gražinamas laikinas objektas naudojamas metodo *GautiKoord(x,y)* iškvietimui.