C++ programavimo kalba

Operatorių perkrovimas

(7 paskaita)

Operatorių perdengimas

Programavimo kalbose naudojami operatoriai pasižymi polimorfizmu (daugiavariantiškumu). Kaip pavyzdys gali būti operatorius **+**, kuris taikomas tiek *int*, tiek *float* tipo kintamiesiems, nors sudėties veiksmai procesoriaus registruose atliekami nevienodai.

Klasėse galima naujai apibrėžti operatorius, t.y. pakeisti jų veikimą. Toks naujas operatorių perorientavimas yra vadinamas **perdengimu (perkrovimu).** Operatorių aprašai nuo funkcijų skiriasi vartojamu žodžiu operator.

Operatoriaus perdengimo sintaksė :

[tipas] ***operator*** *<Operatoriaus_simbolis> (parametrai);*

Perdengimas draudžiamas operatoriams: . :: ?:

Operatorių perdengimas gali galioti lokaliai t.y. klasės ribose arba globaliai t.y. visoje programoje.

Operatoriai – tai funkcijos, todėl operatorių perkrovimas siejamas su funkcijų perkrovimu.

Pavyzdys

```
#include <iostream>
using namespace std:
class Counter
{ private:   unsigned int count;
  public:    Counter(): count(0)
             {  }
             unsigned int Rodo()
              { return count;  }
             void operator ++ ()              // objektas dešinėje ++ pusėje
             {  ++ count; }
};
void main()
{Counter c1, c2;
 cout<< "c1="<<c1.Rodo()<<endl;          // Ekrane ->  c1=0
 cout<< "c2="<<c2.Rodo()<<endl;          // Ekrane ->  c2=0
 ++c1;
 ++c2;
 cout<< "c1="<<c1.Rodo()<<endl;          // Ekrane ->  c1=1
 cout<< "c2="<<c2.Rodo()<<endl;  }       // Ekrane ->  c2=1
```

Operatorių perdengimas

```
include <iostream> using
namespace std;

class Vector
{ public:
      int x, y;
      Vector ()
      { };
      Vector (int, int);
      Vector operator +
      (Vector);
};
```

```
Vector::Vector (int a, int b)
{ x = a; y = b; }

Vector Vector::operator+ (Vector param)
{ Vector temp;
   temp.x = x + param.x;
   temp.y = y + param.y;
   return (temp); }

int main ()
{ Vector a (3,1); Vector b (1,2);
  Vector c;
  c = a + b;
  cout << c.x << "," << c.y;
  return 0; }
```

Perdengiami operatoriai

Operatorių perdengimas – tai funkcijos operatoriai, kurių pavadinimai siejami su raktiniu žodžiu *operator* ir po juo einančiu operatoriumi.

Taisyklė:

Funkcijos-operatoriai turi būti nestatiniai klasės metodai arba turėti klasės tipo argumentą arba nuorodą į klasę.

Sintaksė:

Binariniai operatoriai (*turi du operandus: +, -, * , / , %, ….*)

[tipas] ***operator*** *<Operatoriaus_simbolis> (par1);* *(lokalus)*

[tipas] ***operator*** *<Operatoriaus_simbolis> (par1, par2);* *(globalus)*

Unariniai operatoriai (*turi tik vieną operandą ++, --, +=, -=, ….*)

[tipas] ***operator*** *<Operatoriaus_simbolis> ();* *(lokalus)*

[tipas] ***operator*** *<Operatoriaus_simbolis> (par1);* *(globalus)*

Perdengiami operatoriai

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

Pavyzdys (lokalus perdengimas)

```
class Point {
public:
        Point (int a, int b)
            {Point::x = a; Point::y = b;}
        Point operator + (Point p)
            {return Point(x + p.x, y + p.y);}
        Point operator - (Point p)
            {return Point(x - p.x, y - p.y);}
private:
        int x, y;
};

void main () {
Point p1(10,20), p2(10,20);
Point p3 = p1 + p2;
Point p4 = p1 - p2;
}
```

Pavyzdys (globalus perdengimas)

```
class Point {
public:
        Point (int a, int b)  {Point::x = a; Point::y = b;}

        friend Point operator + (Point p, Point q)  //globalus operatrorius
                {return Point(p.x + q.x, p.y + q.y);}

        friend Point operator - (Point p, Point q)
                {return Point(p.x - q.x, p.y - q.y);}
private:
        int x, y;
};
```

Inkremento operatorių perdengimas

```
#include <iostream>
using namespace std;
class Counter
{ private:   unsigned int count;
  public:    Counter(): count(0)
             {  }
            Counter(int c): count(c)
             { }
            unsigned int Rodo()
            { return count;  }
            Counter operator ++ ()                  // prefiksas
            {  return Counter (++ count); }         // didina ir priskiria

            Counter operator ++ (int)               // postfiksas
            {  return Counter( count++); }          // priskiria ir didina
};
```

Inkremento operatorių perdengimas

```
void main()
{Counter c1, c2;
 cout<< "c1="<<c1.Rodo()<<endl;       // c1=0
 cout<< "c2="<<c2.Rodo()<<endl;       // c2=0
 ++c1;
 c2=++c1;

 cout<< "c1="<<c1.Rodo()<<endl;       // c1=2
 cout<< "c2="<<c2.Rodo()<<endl;       // c2=2

c2=c1++;
cout<< "c1="<<c1.Rodo()<<endl;        // c1=3
cout<< "c2="<<c2.Rodo()<<endl;        // c2=2

}
```

Palyginimo operatorių perdengimas

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

class Distance
{ private:   int metrai, centimetrai;
  public:    Distance(): metrai(0), centimetrai(0)    {  }
             Distance(int m, int cm): metrai(m), centimetrai(cm)     {   }
             void ShowDist ( )
             { cout<< metrai<<“   “<<centimetrai<<endl;
             bool operator < (Distance) const;
 };

bool Distance::operator < (Distance d2) const;
{ int sum1 = metrai*100 + centimetrai;
  int sum2 = d2.metrai*100 + d2.centimetrai;
  return (sum1 < sum2) ? true : false;
}
```

Palyginimo operatorių perdengimas

```
// Pagrindinė programa

void main()
{  Distance dist1;
   Distance dist2(12, 44);
      cout<< “dist1 “; dist1.ShowDist();
      cout<< “dist2 “; dist2.ShowDist();
  if (dist1< dist2)
        cout<< “dist1 mažiau nei dist2 \n”;
 else
     cout<< “dist1 daugiau nei dist2 \n”;
}
```

Operatoriaus << ir >> perdengimas

Operatoriaus << perdengimas leidžia efektyviai naudoti išvedimo procedūrą ir objektą išvesti analogiškai kaip bet kurį kitą kintamąjį.

```
#include <iostream>
using namespace std;
class Distance
{ private:   int metrai, centimetrai;
  public:    Distance(): metrai(0), centimetrai(0)    {  }
             Distance(int m, int cm): metrai(m), centimetrai(cm)    {   }
             friend istream& operator >> (istream& s, Distance& d );
             friend ostream& operator << (ostream& s, Distance& d );
 };

istream& operator >> (istream& s, Distance& d )
{cout<<“Ivesk metrus:”;   s>>d.metrai;
 cout<<“Ivesk centimetrus:”;  s>>d.centimetrai;
 return s;
}
```

Operatoriaus << ir >>perdengimas

```
ostream& operator << (ostream& s, Distance& d )
{ s<<d.metrai;<<“  “<<d.centimetrai;
  return s;
}

void main()
{  Distance dist1, dist2;
   Distance dist3(11, 6);
      cout<< “Ivesk dist reiksmes”;
      cin>>dist1>>dist2;
   cout<<“dist1=“<<dist1<<endl;
   cout<<“dist2=“<<dist2<<endl;
   cout<<“dist3=“<<dist3<<endl;
}
```

Pirmas parametras – tai nuoroda į istream, antras – nuoroda, kuri bus modifikuojama.

Operatoriaus << ir >>perdengimas

Panaudojame tą pačią klasę **Distance** su perkrautais operatoriais, tačiau darbui su failais main() funkcijoje.

```
void main()
{ ofstream ofile;
  ofile.open("duomenys.dat" );
  Distance dist1;
  for (int i=0; i<5; i++)
    { cout<< "Ivesk dist reiksmes";
       cin>>dist1;
       ofile<<dist1; }
ofile.close();
ifstream ifile;
ifile.open("duomenys.dat");
```

```
while(true)
{  ifile>>dist1;
   if (ifile.eof() )
        break;
cout<<"Distance"<<dist1<<endl;
}
}
```

[ ] perkrovimas (pavyzdys 1)

```
#include <iostream.h>
#include <process.h>            // dėl funkcijos exit()
const int LIMIT = 100;
class masyvas {
    private:  int arr[LIMIT];
    public:   int& operator [ ] (int n)
                {   if (n < 0 || n  >= LIMIT )
                        { cout << “Indekso numeris neteisingas\n”; exit(1); }
                    return arr[n];
                }
};
       void main()
       { masyvas sa1;
           for (int j=0; j <LIMIT; j++)
           {   sa1[ j ] = j+10;
               cout << “Elementas “ << j << “lygus “ << sa1[ j ]<<endl;
           } }
```

[ ] perkrovimas (pavyzdys 2)

```
// Pavyzdys su asociatyviniu masyvu
#include <iostream.h>
#include <string.h>
class AssocVec {
public:
        AssocVec          (const int dim);
        ~AssocVec         (void);
        int& operator [ ] (const char *idx);
private:
        struct VecElem {
                char *index;
                int  value;
        }  *elems;                  // asociatyvinio masyvo elementai
        int dim;                    // galimas elementų skaičius
        int used;                   // esamas elementų skaičius
};
```

[ ] perkrovimas (pavyzdys 2)

```
AssocVec::AssocVec (const int dim)
{
        AssocVec::dim = dim;
        used = 0;
        elems = new VecElem[dim];
}
AssocVec::~AssocVec (void)
{
        for (register i = 0; i < used; ++i)
        delete elems[i].index;
        delete [ ] elems;
}
```

[ ] perkrovimas (pavyzdys 2)

```
int& AssocVec::operator [ ] (const char *idx)
{
        for (register i = 0; i < used; ++i)                  // ieškomas elementas
                if (strcmp(idx, elems[i].index) == 0)
                        return elems[i].value;
        if (used < dim &&                                    // sukuriamas naujas elementas
           (elems[used].index = new char[strlen(idx)+1]) != 0)
        {       strcpy(elems[used].index, idx);
                elems[used].value = used + 1;
                return elems[used++].value;
        }
        static int dummy = 0;
        return dummy;}
```

Naudojame perkrautą [ ] taip:

```
AssocVec count(5);
count["apple"] = 5;
count["orange"] = 10;
count["fruit"] = count["apple"] + count["orange"];   // Atsakymas: count["fruit"] = 15.
```

= perkrovimas

Operatoriaus = perdengimas leidžia priskirti objektui naujas reikšmes.

```
#include <iostream.h>
class Distance
{ private:   int metrai;
  public:    Distance(): metrai(0) {  }
             Distance(int m): metrai(m)  {   }
             void display ()
              { cout<< metrai<<endl;}
             operator = ( int d )
              { metrai = d;
              cout<<“Priskirimas sekmingas”<<endl;}
 };

 void main()
{  Distance dist1, dist2(11);
     dist1=17;                              // perkrovimas
     Distance dist3 = dist2;                // neperkrovimas;
  dist1.display(); dist2.display(); dist3.display(); }
```

Operatorius *new* ir *new[ ]*

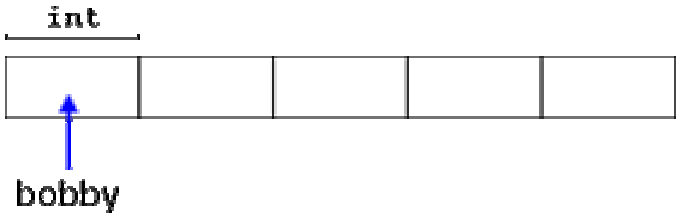
Norint dinamiškai išskirti atmintį, naudojamas operatorius **new**. Jis išskiria atmintį nurodytam elementų kiekiui ir gražina rodyklę į išskirto atminties bloko pradžią.

Sintaksės variantai:

pointer **= new** *tipas;* *pointer* **= new** *tipas* **[***elementai***];**

Pavyzdžiui:

```
int * bobby;
bobby = new int [5];      // išskirtas atminties blokas 5 int tipo elementams
```




```
int * bobby;
bobby = new int [5];
if (bobby == NULL) {
  {cout<< “Atminties klaida\n”; exit(1); }     // klaida išskiriant atmintį.
  };
```

Operatorius *delete ir delete[ ]*

Jei dinamiškai išskirta atmintis daugiau nenaudojama, ją galima atlaisvinti. Tam tikslui reikia naudoti operatorių **delete**.

Sintaksė:

delete *pointer*; **delete [ ]** *pointer*;

```
#include <iostream.h>
#include <stdlib.h>
int main ()
{  char input [100];
   int i,n;
   long * L;
   cout << “Kiek bus skaičių? ";
   cin.getline (input,100);
   i=atoi (input);
   L= new long[i];
   if (L == NULL) exit (1);
   for (n=0; n<i; n++)
    {   cout << "Enter number: ";
        cin.getline (input,100);
    L[n]=atol (input);
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
       cout << L[n] << ", ";
    delete[ ]  L;
    return 0;
}
```

new ir delete perdengimas

Dinaminio atminties paskirstymo operatoriai **new** ir **delete** gali būti perkrauti klasėse. Toks perkrovimas susijęs su racionalesniu atminties panaudojimu.

```
class C
{
public:
void* operator new[ ] (size_t);
void operator delete[ ] (void*);

};

void* C::operator new[ ] (size_t allocSize)
{
    return ::operator new[ ] (allocSize);
}
```

new ir delete perdengimas

```
void C::operator delete[ ] (void *p)
{
::operator delete[ ] (p); // global operator
}

int main()
{
C *p;
p = new C[10];
delete[ ] p;
}
```