Přetížení operátorů (operator overloading)

Přetížení operátorů je definování jejich chování pro další datové typy. Je specifickým typem polymorfismu. Přetížení operátorů je v C++ u některých operátorů definováno přímo v jazyce (operátory **<<** a **>>** ve streamech). Uživatel si může také definovat vlastní přetížení operátorů.

U přetížených operátorů zůstávají zachovány jejich charakteristické vlastnosti

- arita
- precedence
- asociativita

V definici přetížení musí být aspoň jeden z operandů třída.

Lze přetížit operátory:

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

Operátory **+**, **-**, *****, **&** lze přetížit binární i unární.

Nelze přetížit operátory:

`::`	`.*`	`.`	`?:`	`sizeof`

Přetížení lze definovat:

- Členskou funkcí třídy, je-li první (levý) operand operace třída.
- Globální funkcí.

Tvar členské funkce pro unární operátor

```
typ operator operátor () { }
```

Tvar členské funkce pro binární operátor

```
typ operator operátor (pravý_operand) { }
```

Tvar globální funkce pro unární operátor

```
typ operator operátor (operand) { }
```

Tvar globální funkce pro binární operátor

```
typ operator operátor (levý_operand, pravý_operand) { }
```

Tvar funkce pro konverzi datového typu dané třídy na stanovený datový typ:

```
operator typ () { }
```

Je-li operand uveden jako parametr funkce přetížení:

- pro datové typy s malou velikostí ho lze zadat hodnotou
- pro datové typy s větší velikostí se zadává referencí

Úloha: Je dána třída pro uložení data

```
class Datum { char den,mesic; short rok; };
```

Potřebujeme setřídit pole dat. Použijeme k tomu algoritmus třídění vkládáním, který srovnává tříděné prvky operací **<**. Pro třídění definujeme tuto operaci ve třídě *Datum*.

```
class Datum { char den,mesic; short rok;
  public: Datum(char d,char m,short r): den(d),mesic(m),rok(r)
          { }
          bool operator < (Datum d) const
          {
            if (rok!=d.rok)      return rok<d.rok;
            if (mesic!=d.mesic) return mesic<d.mesic;
                                 return den<d.den;
          }
};

void tridit(Datum A[], int n)
{
  for (int i=1; i<n; ++i)
  { Datum x=A[i];
    int j;
    for (j=i-1; j>=0 && x<A[j]; --j) A[j+1]=A[j];
    A[j+1]=x;
  }
};
```

Poznámka: V našem případě objekty třídy `Datum` mají malou velikost `sizeof(Datum)=4`, stačí je zadat hodnotou.

Úloha: Sestavíme třídu pro práci s řetězci. Budeme v ní mít přetížené operátory:

```
Retez << znak     // přidá znak k řetězci v objektu, vrací referenci na objekt
Retez += řetěz    // přidá řetězec k řetězci v objektu, vrací referenci na objekt
Retez += Retez    // přidá řetězec obsažený v jiném objektu, vrací referenci na objekt
```

```
Retez = řetěz     // uloží do objektu nový řetězec, vrací referenci na objekt
Retez = Retez     // do objektu uloží kopii řetězce obsaženého v jiném objektu
(const char *) Retez    // vrátí řetězec uložený v objektu
```

```
class Retez { char *p;
  public: Retez(int n) { p=new char[n+1]; *p=0; }
          Retez & operator << (char);
          Retez & operator += (const char *);
          Retez & operator += (const Retez &);
          Retez & operator = (const char *);
          Retez & operator = (const Retez &);
          operator const char * () const { return p; }
          ~Retez() { delete [] p; }
};
```

V přetížení operátorů pro jednoduchost nejsou ošetřeny situace, kdy délka výsledného řetězce překročí rozsah pole, ve kterém je řetězec uložen.

```
Retez &Retez::operator << (char z)
{
  auto l=strlen(p);
  p[l]=z;
  p[l+1]=0;
  return *this;
}

Retez &Retez::operator += (const char *s)
{
  strcat(p,s);
  return *this;
}

Retez &Retez::operator += (const Retez &r)
{
  strcat(p,r.p);
  return *this;
}

Retez &Retez::operator = (const char *s)
{
  strcpy(p,s);
  return *this;
}
```

```
Retez &Retez::operator = (const Retez &r)
{
  strcpy(p,r.p);
  return *this;
}
Retez r(100);
r << 'C' << '+' << '+';
r += " je ";
Retez s(100);
s = "objektove orientovany jazyk";
r += s;
cout << r << endl;  // C++ je objektove orientovany jazyk
```

Přetížení operátorů ++ a --

Přetížení prefixových operátorů **++** a **--** má tvar:

```
  typ operator operátor () { }
```

Funkce pro přetížení postfixových operátorů **++** a **--** má tvar:

```
  typ operator operátor (int) { }
```

Sestavíme třídu pro uložení souřadnice. Přetíženým operátorem ++ se souřadnice zvětší o krok, jehož hodnota je zadaná v konstruktoru třídy.

```
class Souradnice { float x, krok;
  public:
    Souradnice(float x,float k):x(x),krok(k) { }
    float operator ++ () { return x+=krok; }
    float operator ++ (int) { auto v=x; x+=krok; return v; }
    operator float () const { return x; }
};
Souradnice s(0,0.5);
cout << ++s << endl;   // 0.5
cout << s++ << endl;   // 0.5
cout << s << endl;     // 1
```

Přetížení operátoru indexu

Přetížení operátoru má tvar zápisu přetížení unárního operátoru:

```
  typ operator [] (index) { }
```

Sestavíme třídu pro uložení pole. Přetížení operátor indexu vrací referenci na prvek pole.

```
class Pole { int *p; const unsigned n;

  public: Pole(unsigned n):n(n) { p=new int [n]; }

          int & operator [] (unsigned i) const
          {
            if (i<n) return p[i];
            cerr << "Chybny index: " << i << endl;
            abort();
          }

          ~Pole() { delete [] p; }
};
```

Přetížení operátoru funkce

```
  typ operator () (...parametry...) { }
```

Sestavíme třídu pro lineární rovnici. Přetížíme operátor funkce pro zadání koeficientů rovnice a přetížíme operátor funkce pro zjištění kořene rovnice.

```
class Rovnice { float a,b;

  public: void operator () (float aa,float bb) { a=aa; b=bb; }

          float operator () () const { return -b/a; }
};

Rovnice r;

r(4,-3);

cout << r();    // 0.75
```
