přetížení operátorů (o)

- pro vlastní typy je možné přetížit i operátory (tj. definovat vlastní)
- pro definici slouží klíčové slovo operator následované typem/znakem operátoru
- deklarace pomocí „funkčního“ volání např. unární a binární + pro typ int by šlo psát:

```
int operator +(int a1) {}
int opreator +(int a1, int a2) {}
```

s „funkčním“ voláním

```
operator +(i);
operator+(i,j);
```

ve zkrácené formě

```
+i;
```

- `i + j;`
-

možnost volat i „funkčně“

```
operator=(i,operator+( j ))
```

nebo zkráceně

```
i=+j
```

- operátorem je i vstup a výstup do **stream**u, **new** a **delete**
- hlavní využití u vlastních tříd (objektů)

operátory **(o)**

- operátory lze v C++ přetížit
- správný operátor je vybrán podle seznamu parametrů (a dostupných konverzí), výběr rozliší překladač podle kontextu
- operátory unární mají jeden parametr – u funkcí proměnnou se kterou pracují, nebo ”this” u metod
- operátory binární mají dva parametry – dva parametry funkce, se kterými pracují nebo jeden parametr a ”this” u metod
- unární operátory:
  +, -, ~, !,++,--
- binární
  +,-,*,/,%,=,^,&,&&,|,||,>,<,>=,==, +=,*=,<<,>>,<<=, …
- ostatní operátory [ ], ( ), new, delete
- operátory matematické a logické
- nelze přetížit operátory:
  sizeof, ? :, ::, ., .*
- nelze změnit počet operandů a pravidla pro asociativitu a prioritu
- nelze použít implicitních parametrů

- slouží ke zpřehlednění programu
- snažíme se, aby se přetížené operátory chovaly podobně jako původní (např. nemění hodnoty operandů, + sčítá nebo spojuje…)
- klíčové slovo operator
- operátor má plné (funkční) a zkrácené volání

```
z = a + b
z.operator=(a.operator+(b))
```

- nejprve se volá operátor + a potom operátor =
- funkční zápis slouží i k definování operátoru

metoda patří ke třídě (T::) první parametr je this

```
T T::operator+(T & param) {}
```

(friend) funkce pro případ, že prvním operandem je „cizí“ typ

```
T operator+(double d, T&param) {}
```

Unární operátory

- mají jeden parametr (this)
- například + a – , ~, !, ++, --

```
complex          operator+(void) // zbytečně vrací objekt
complex        & operator+(void) // vrácený objekt lze změnit
complex          operator+(void) const
complex const & operator+(void) const
```

- operátor plus (+aaa) nemění prvek a výsledkem je hodnota tohoto (vně metody existujícího) prvku – proto lze vrátit referenci – což z úsporných důvodů děláme (výsledek by neměl být měněn=const),
- operátor mínus (-aaa) nemění prvek a výsledek je záporná (tj. odlišná) hodnota – proto musíme vytvořit nový prvek – vracíme hodnotou

- pokud nejsou operandy měněny (a většina standardních operátorů je nemění), potom by měly být označeny const (pro this i parametr). návrat referencí potom musí být také const.

- operátory ++ a `--` mají prefixovou a postfixovou notaci
- definice operátorů se odliší (fiktivním) parametrem typu int
- je-li definován pouze jeden, volá se pro obě varianty
- některé překladače obě varianty neumí
- při volání dáváme přednost ++a (netvoří tmp objekt)

`++( void) s voláním ++x`
`++( int ) s voláním x++. Argument int se však při volání nevyužívá`

```
T& operator ++(void)
T  operator ++( int)
```

operátor ++aa na rozdíl od aa++ netvoří dočasný prvek pro návratovou hodnotu a proto by měl v situacích, kdy lze tyto operátory zaměnit, dostávat přednost

Binární operátory

- mají dva parametry (u třídy je jedním z nich)
- například +, -, %, &&, &, <<, =, +=, <, …

```
complex complex::operator+ (const complex & c) const
complex complex::operator+ (         double c) const
complex          operator+ (double f, const complex & c)

a + b         a.operator+(b)
a + 3.14      a.operator+(3.14)
3.14 + a      operator+(3.14,a)
```

- výstupní hodnota různá od vstupní –> vrácení hodnotou
- mohou být přetížené – více stejných operátorů v jedné třídě
- lze přetížit i jako funkci (globální prostor, druhý parametr je třída) – často friend
- opět může nastat kolize při volání (implicitní konverze)
- parametry se (jako u standardních operátorů) nemění a tak by měly být označeny const, i metoda by měla být const

Operátor =

- pokud není napsán, vytváří se implicitně (mělká kopie – přesná kopie 1:1, memcopy)
- měl by (díky kompatibilitě) vracet hodnotu (musí fungovat zřetězení a = b = c = d;)
- obzvláště zde je nutné ošetřit případ a = a
- pro činnost s dynamickými daty nutno ošetřit mělké a hluboké kopie
- nadefinování zamezí vytvoření implicitního =
- je-li v sekci private, pak to znamená, že nelze použít (externě)
- od kopykonstruktoru se liší tím, že musí před kopírováním ošetřit proměnnou na levé straně

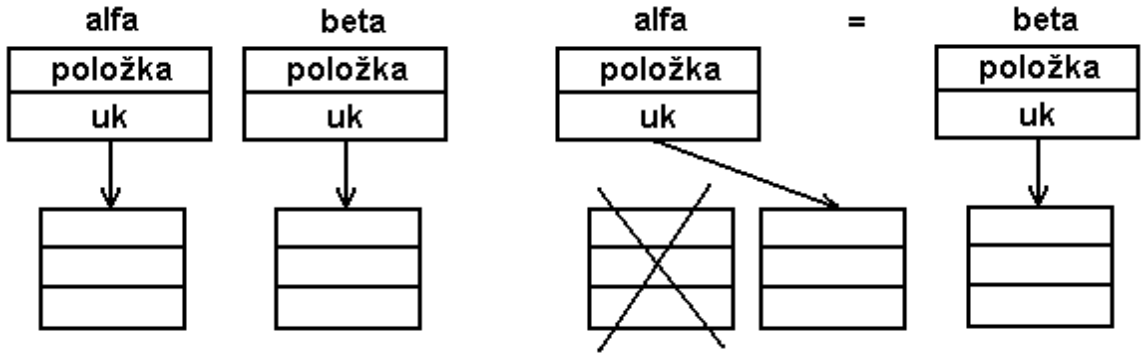
```
T& operator=(T const& r)
```

musí umožňovat

```
a = b = c = d = …;
a += b *= c /= d &= …;
```

Vysvětlete rozdíl mezi mělkým a hlubokým kopírováním.
Vysvětlete rozdíl mezi kopykonstruktorem a operátorem =.

Problém nastává v případě, že jsou v objektu ukazatele na paměť, kterou je nutné při zániku objektu odalokovat.
Na rozdíl od kopykonstruktoru je nutné si uvědomit, že v cílovém objektu jsou platná data. Před jejich naplněním novými hodnotami je nutné odalokovat původní paměť.
Při mělkém kopírování (i implicitně vytvářený operátor) dojde ke sdílení paměti (přiřazení ukazatele na paměť) aniž by o tom objekty věděly. Při odalokování dojde k problémům.
Při hlubokém kopírování (musí napsat tvůrce třídy) se vytvoří kopie dat paměti, na kterou ukazuje ukazatel, nebo se použije mechanizmus čítání odkazů na danou paměť.



Vysvětlete rozdíly mezi operátorem = pro:

a) ukazatele
b) objekty s členskými objekty bez ukazatelů
c) objekty obsahující ukazatele bez operátoru =
d) objekty obsahující ukazatele s operátorem =

Způsoby přiřazení:

```
string * a,* b;
a = new string;
b = new string;
a = b ;
delete a ;
delete b ;
```

- pouze přiřazení ukazatelů, oba ukazatele sdílí stejný objekt (stejná statická i dynamická data)
- chyba při druhém odalokování, protože odalokováváme stejný objekt podruhé

Objekty bez ukazatelů mohou využívat implicitní operátor = (memcopy), pokud po nich není požadován nějaký postranní efekt (nastavení čítače, kontrola dat…)

```
string {int delka; char *txt}
string a , b(”ahoj”);
a = b ;
”delete a” ; // volá překladač
”delete b” ;
```

- je vytvořeno a tedy použito implicitní =
- ukazatel txt ukazuje na stejná dynamická data (statické proměnné jsou zkopírovány, ale dále se používají nezávisle)
- pokud je nadefinován destruktor, který odalokuje txt (což by měl být), potom zde odalokováváme pamět, kterou odalokoval již destruktor pro prvek a

```
string {int delka; char *txt;operator =();}
string a , b(”ahoj”);
a = b ;
”delete a” ;
”delete b ”;
```

- použito nadefinované =
- v = se provede kopie dat pro ukazatel txt
- oba prvky mají svoji kopii dat statických i dynamických
- každý prvek si odalokovává svoji kopii

Konverzní operátory

- převod objektů na jiné typy
- využívá překladač při implicitních konverzích (zákaz pomocí klíčového slova explicit)
- opačný směr jako u konverzních konstruktorů (jiný typ -> můj typ/ můj typ -> jiný typ)
- například konverze na standardní typy – int, double…
- nemá návratovou hodnotu (je dána názvem)
- nemá parametr (jen this)
- v C++ lépe používat konverze pomocí "cast" (dynamic, static …)

```
operator typ(void)
T::operator int(void)
```

volán implicitně nebo

```
T aaa;
int i = int (aaa) ;
(int) aaa; // starý typ - nepoužívat
```

Přetížení funkčního volání ( )

- může mít libovolný počet parametrů
- takto vybaveným objektům se říká funkční objekty
- nedoporučuje se ho používat (plete se s funkcí)

```
operator ()(parametry )
double& T::operator()(int i,int j) { }
T aaa;

double d = aaa(4,5); // vypadá jako funkce
// ale je to funkční objekt
d  = aaa.operator()(5,5);
aaa(4,4) = d;
```

Přetížení indexování [ ]

- podobně jako operátor( ) ale má pouze jeden operand (+ this, je to tedy binární operátor)
- nejčastěji používán s návratovou hodnotou typu reference (l-hodnota)

```
double& T::operator[ ](int )

aaa[5] = 4;
d = aaa.operator[ ](3);
```

přetížení přístupu k prvkům třídy

- je možné přetížit ” -> ”
- musí vracet ukazatel na objekt třídy, pro kterou je operátor -> definován protože:

```
TT* T::operator->( param ) { }

x -> m;                    // je totéž co
(x.operator->( ) ) -> m;
```

operátory a STL

-je-li nutné používat relační operátory, stačí definovat operátory == a <

- ostatní operátory jsou z nich odvozeny pomocí template v knihovně <utility>

Operátory vstupu a výstupu

- knihovní funkce
- řešeno pomocí třídy
- použití (přetížení) operátorů bitových posunů << a >>
- díky přetížení operátoru není nutné kontrolovat typy (správný hledá překladač)
- pro vlastní typy nutno tyto operátory napsat
- pro práci s konzolou předdefinované objekty cin, cout, cerr v knihovně <iostream>
- objekty jsou v prostoru std::. Použití using: std::cout, std::endl.
- jelikož prvním operandem je "cizí" objekt, jedná se o (friend) funkce

```
cin >> i >> j >> k;
cout << i << ”text” << j << k << endl;

xstream & operator xx (xstream &, Typ& p) {  }

istream & operator >> (istream &, Typ& p) {  }
ostream & operator << (ostream &, Typ& p) {  }
```

statické metody (o)

- pouze jedna na třídu
- nemá this
- ve třídě označená static
- vlastní tělo ve zdrojové části
- nesmí být virtuální
- (jako friend funkce, může k private členům)
- externí volání se jménem třídy bez objektu **`Třída::fce( )`**