

X36PJC

# 10. přednáška

Přetěžování operátorů

# Minulá přednáška

- Třídy
- Konstruktory
  - Implicitní
  - Kopírující
  - Uživatelské konverze

# Přetěžování operátorů

- Operátor (+, -, \*, /, ...) - jiný zápis volání funkce.
- Standardní operátory jsou přetížené:
  - 3 + 3**
  - 3 + 3.5**
  - 3.5 + 3.5**
- V C++ lze na rozdíl od většiny ostatních jazyků operátory dále přetěžovat pro jiné typy parametrů:
  - nelze zavést nové operátory,
  - nelze předefinovat stávající operátory,
  - nelze měnit aritu operátorů,
  - nelze měnit prioritu operátorů.
- Existují v Javě přetížené operátory?

# Přetěžování operátorů

- Přetížení operátorů:
  - klíčové slovo **operator**,
  - přetížení pomocí funkce,
  - přetížení pomocí metody.
- Metodou lze přetížit všechny operátory kromě:  
**:: . .\* ?:**
- Funkcí dále nelze přetížit operátory:  
**-> ->\* = () []**
- Přetížit funkcí či třídní metodou lze i operátory:  
**new new[] delete delete[]**
- Metodou lze přetížit i operátory přetypování.

# Přetěžování operátorů

- Přetížením operátoru nelze změnit jeho prioritu.
- Alespoň jeden parametr přetíženého operátoru musí být UDT (user defined type).
- Nelze tedy například přetížit operátor + s parametry int, double.
- Přetížený operátor nesmí mít implicitní hodnoty parametrů (výjimku tvoří operator ( ) )

# Přehled přetížitelných operátorů

<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code>^</code>
<code>&amp;</code>	<code> </code>	<code>~</code>	<code>!</code>	<code>,</code>	<code>=</code>
<code>&lt;</code>	<code>&gt;</code>	<code>&lt;=</code>	<code>&gt;=</code>	<code>++</code>	<code>--</code>
<code>&lt;&lt;</code>	<code>&gt;&gt;</code>	<code>==</code>	<code>!=</code>	<code>&amp;&amp;</code>	<code>  </code>
<code>+=</code>	<code>-=</code>	<code>/=</code>	<code>%=</code>	<code>^=</code>	<code>&amp;=</code>
<code> =</code>	<code>*=</code>	<code>&lt;&lt;=</code>	<code>&gt;&gt;=</code>	<code>[]</code>	<code>()</code>
<code>-&gt;</code>	<code>-&gt;*</code>	<code>new</code>	<code>new []</code>	<code>delete</code>	<code>delete []</code>

# Přetěžování operátorů funkcí

```
struct TCplx
{
    double re, im;
};
TCplx operator + ( TCplx a, TCplx b )
{
    TCplx c;
    c . re = a . re + b . re;
    c . im = a . im + b . im;
    return c;
}
TCplx x = { 1, 0 }, y = { 2, 1 }, z;
z = x + y;
z = operator + ( x, y ); // jiný zápis volání op.
```

# Přetěžování operátorů funkcí

- Operátor + přetížený v příkladu nebude použit pro:  
    **TCplx a, b;**  
    **a = b + 3;**  
    **a = 4 + b;**
- Řešení 1:
  - přetížit operátor + i pro parametry typu **double**,
  - pro každý operátor musíme přetěžovat 3 varianty.

```
TCplx operator+ ( TCplx a, double b )
```

```
{ ... }
```

```
TCplx operator+ ( double a, TCplx b )
```

```
{ ... }
```



# Přetěžování operátorů funkcí

- Řešení 2:
  - využít konstruktor uživatelské konverze z typu **double** na **typ TCplx**,
  - ponechat pouze jeden přetížený operátor **+**.

```
TCplx::TCplx ( double a )  
{  
    re = a;  
    im = 0;  
}
```

```
TCplx a, b;  
a = 4 + b; // a = TCplx ( 4 ) + b;
```

# Přetěžování operátorů funkcí

- Parametry funkce přetíženého operátoru:

```
TCplx operator+ ( TCplx a, TCplx b )  
{ ... } // ok, ale pomale pro velke instance
```

```
TCplx operator+ ( TCplx & a, TCplx & b )  
{ ... } // !!! mj. nebude asociativni.Proc?
```

```
TCplx operator+ ( const TCplx & a,  
                  const TCplx & b )  
{ ... } // ok
```

# Přetěžování operátorů funkcí

- Návratový typ funkce přetíženého operátoru:

```
TCplx & operator+ ( TCplx a, TCplx b )  
{  
    TCplx res;  
    res . re = a . re + b . re;  
    res . im = a . im + b . im;  
    return res; //!! chyba - reference na lok. prom.  
}
```

```
TCplx & operator+ ( TCplx a, TCplx b )  
{  
    static TCplx res;  
    res . re = a . re + b . re;  
    res . im = a . im + b . im;  
    return res; //!! chyba – asociativita, thready  
}
```

# Přetěžování operátorů funkcí

```
TCplx & operator+ ( TCplx a, TCplx b )
```

```
{  
    TCplx * res = new TCplx;  
    res . re = a . re + b . re;  
    res . im = a . im + b . im;  
    return * res; //!! chyba – nelze smazat  
}
```

```
TCplx operator+ ( TCplx a, TCplx b )
```

```
{  
    TCplx res;  
    res . re = a . re + b . re;  
    res . im = a . im + b . im;  
    return res; // ok  
}
```

# Časté chyby

- Modifikace parametrů:
  - standardní operátory s výjimkou těch, které mají vedlejší efekt, nemění své parametry.
  - první parametr mění operátory přiřazení (**+=, -=, ...**) a **operátory inkrementu/dekrementu (++, --)**,
  - tomu by mělo odpovídat použití **const v parametrech**,
  - pokud Vámi přetížený operátor bude měnit operandy, bude kód pro ostatní programátory nečitelný a nepochopitelný (problémy s údržbou, předáním kódu, ... ).

# Časté chyby - logické

- Unární operátory (např. mínus):

```
TCplx & operator- ( TCplx & a )
```

```
{
```

```
  a . re = - a . re;    // !! mení parametr
```

```
  a . im = - a . im;
```

```
  return ( a );
```

```
}
```

Analogie:

```
int a = 10, b;
```

```
b = -a;    // a = 10 beze zmeny, b = -10
```

```
TCplx c = { 4, 5 }, d;
```

```
d = -c;    // c = -4 - 5i, d = -4 - 5i
```

# Unární mínus

```
TCplx operator- ( const TCplx & a )  
{  
    TCplx res;  
  
    res . re = - a . re;  
    res . im = - a . im;  
    return ( res );  
} // OK
```

# Přetěžování operátorů metodou

- Přetížení operátoru funkcí:
  - nelze pro všechny operátory,
  - problémy s přístupem ke členským proměnným objektů.
- Řešení – přetížení metodou:
  - přístup ke členským proměnným,
  - použitelné pro všechny operátory, které lze přetížit.
- Realizace:
  - jméno metody – **operator ...**,
  - parametry – o jeden méně, než je arita operátoru,
  - levý operand – instance nad kterou je metoda spuštěna.



# Přetěžování operátorů metodou

```
class CCplx
{
    double re, im;
public:
    CCplx ( double r, double i=0 ) : re(r), im(i) {}
    CCplx operator - ( void ) const;
    CCplx operator + ( const CCplx & x ) const;
};

CCplx CCplx::operator - ( void ) const
{ // unarni minus – 0 parametru
    return ( CCplx ( -re, -im ) );
}

CCplx CCplx::operator + ( const CCplx & x ) const
{ // binarni plus – 1 parametr
    return ( CCplx ( re + x.re, im + x.im ) );
}
```

# Přetěžování operátorů metodou

```
CCplx a (3, 4), b (2);
```

```
CCplx c = a + b;
```

```
// c = a . operator + ( b );
```

```
CCplx d = a + 4;
```

```
// ok, d = a . operator + ( CCplx ( 4 ) );
```

```
CCplx e = 5 + a;
```

```
// chyba, neexistuje operator + ( int, CCplx & )
```

# Přetěžování operátorů – friend

- Přetížit operátor+ pro (**double,const CCplx&**):
  - nelze metodou (**int není třída**),
  - musí se přetížit funkcí.
- Problém s funkcí – přístup ke členským proměnným třídy **CCplx**:
  - zviditelnit je **public (nevhodné)**,
  - přístup pomocí čtecích metod (getter) – zdržení,
  - delegovat na tuto funkci právo přístupu ke členským proměnným (friend funkce).

# Přetěžování operátorů – friend

```
class CCplx
{
    double re, im;
public:
    CCplx ( double r, double i=0 ) : re(r), im(i) {}
    CCplx operator - ( void ) const;
    CCplx operator + ( const CCplx & x ) const;
    friend CCplx operator + ( double a,
                             const CCplx & b );
};
CCplx operator + ( double a, const CCplx & x )
{ // binarni plus funkci – 2 parametry
    return ( CCplx ( a + x.re, x.im ) );
}
```

# Přetěžování operátorů – **friend**

- **friend dává práva přístupu:**
  - funkci,
  - jiné třídě.
- Označená funkce/třída má stejná práva přístupu jako metody.
- Neexistuje **friend metoda**, existuje pouze **friend funkce**.
- Neplývejte **friend**.

Přetěžování operátorů << a >>

# Přetěžování operátorů << a >>

- Operace nad vstupními a výstupními proudy:
  - formátovaný výstup – operátor << ,
  - formátovaný vstup – operátor >> .
- Standardní knihovna přetěžuje operátory formátovaného vstupu a výstupu pro:
  - primitivní datové typy (**int, char, double,...**),
  - některé třídy standardní knihovny (**string**).
- Pro vlastní třídu **T si operátor musíme přetížit sami:**
  - **ostream & operator << ( ostream &, const T & );**
  - **istream & operator >> ( istream &, T & );**
- Přetížení metodou nelze – (nemáme přístup ke třídám **istream/ostream**),
- Musíme přetížit pomocí funkcí nebo **friend funkcí**.

# Přetěžování operátorů << a >>

- Parametry **ostream/istream**:
  - generické výstupní/vstupní proudy,
  - abstraktní třídy (předchůdci tříd **ofstream/ifstream** a **ostrstream/istrstream**) – předané referencí.
- Operátor <<:
  - zapisovaný objekt předán referencí (šetří kopii),
  - konstantní reference (není třeba ji zapisovat).
- Operátor >>:
  - čtený objekt předán referencí,
  - nesmí být konstantní (výstupní parametr).



# Přetěžování operátorů << a >>

```
// reseni s friend funkci
class CStr
{
    char * str;
    int len;
public:
    CStr ( const char * str );
    friend ostream & operator << ( ostream & os,
                                   const CStr & x );
};
ostream & operator << ( ostream & os,
                       const CStr & x )
{
    os << x . str;
    return ( os );
}
```

# Přetěžování operátorů << a >>

```
// alternativni reseni bez friend
class CStr
{
    char * str;
    int len;
public:
    CStr ( const char * str );
    ...
    void print( ostream & os ) const { os << str; }
};
ostream & operator << ( ostream & os,
                       const CStr & x );
{
    x . print ( os );
    return ( os );
}
```

# Přetěžování operátoru =

- Podobný kopírujícímu konstrukturu.
- Je možné jej přetížit, pokud není přetížen, systém jej vygeneruje automaticky:
  - staticky alokované členské proměnné – objekty jsou zkopírované jejich operátory =,
  - staticky alokované členské proměnné skalárních typů jsou zkopírované binárně.
- Operátor = **se použije**:
  - pokud dochází ke kopírování a
  - nevzniká nová instance.
- Kopírující konstruktor se použije:
  - vzniká nová instance a
  - vzniká zkopírováním existující instance.

# Operátor = vs. kopírující konstruktor

- Kopírující konstruktor:
  - inicializuje instanci (alokuje její prostředky),
  - původní obsah – nedefinovaný (binární smetí).
- Operátor =:
  - kopíruje do existující instance,
  - typicky uvolní prostředky původní instance,
  - poté alokuje prostředky pro novou instanci.
- Destruktor:
  - uvolňuje alokované prostředky.
- Často platí:
  - `operator=`  $\Leftrightarrow$  `destruktor` + `kopírující konstruktor`

# Přetěžování operátoru =

```
class CStr
{
    char * str;
    int len;
public:
    CStr ( const char * str );
    CStr ( const CStr & src );
    ~CStr ( void );
    CStr & operator= ( const CStr & src );
    ...
};
CStr::CStr ( const char * str )
{
    len = strlen ( str );
    this -> str = new char [len + 1];
    strncpy ( this -> str, str, len + 1 );
}
```

# Přetěžování operátoru =

```
CStr::~CStr ( void )
```

```
{ delete [] str; }
```

```
CStr::CStr ( const CStr & src )
```

```
{
```

```
    len  = src . len;
```

```
    str  = new char [len + 1];
```

```
    strncpy ( str, src . str, len + 1 ); }
```

```
CStr & CStr::operator= ( const CStr & src )
```

```
{
```

```
    delete [] str;
```

```
    len  = src . len;
```

```
    str  = new char [len + 1];
```

```
    strncpy ( str, src . str, len + 1 );
```

```
    return *this; }
```

# Přetěžování operátoru =

```
CStr a ( "Ahoj" );  
CStr b = a;    // kopirující konstruktor  
CStr c ( a ); // kopirující konstruktor  
c = b;        // operator =  
a = a;        // operator =, chyba !
```

```
CStr & CStr::operator= ( const CStr & src )  
{  
    if ( this != & src ) // oprava chyby a = a  
    {  
        delete [] str;  
        len  = src . len;  
        str  = new char [len + 1];  
        strncpy ( str, src . str, len + 1 );  
    }  
    return ( *this ); }
```

# Přetěžování operátoru []

- Operace indexace.
- Pouze pro datový typ třída, přetížit metodou.
- Lze indexovat i jiným typem než celým číslem (např. řetězcem).
- Lze indexovat pouze jedním indexem:
  - lze ale vrátit datový typ, který bude znovu indexovatelný,
  - nebo přetížit operátor – volání funkce ()



# Přetěžování operátoru []

```
class CStr
{
    char * str;
    int len;
public:
    char & operator [] ( int idx );
    ...
};
char & CStr::operator [] ( int idx )
{
    if ( idx < 0 || idx >= len ) throw "mimo meze";
    return str[idx];
}
```

```
CStr a ( "test" );
a[2] = a[1];    // obsah bude 'teet'
```

# Přetěžování operátoru ()

- Pouze pro datový typ třída, přetížit metodou.
- Arita operátoru – libovolná (jediný takový operátor).
- Typ operandů – libovolný (velká variabilita).
- Využití – hlavně jako funktor v STL.
- Nazývá se také někdy funkční objekt.

```
class CStr
{
    char * str;
    int len;
    CStr ( const char * str, int len );
public:
    CStr operator () ( int from, int to ) const;
    // priklad - podretezec.
    ...
};
```

# Přetěžování operátoru ()

```
CStr CStr::operator () ( int from, int to ) const
{
    if ( from > to || from < 0 || to >= len ) throw ...;
    return ( CStr ( str + from, to - from ) ); }
```

```
CStr::CStr ( const char * str, int len )
{
    this -> len = len;
    this -> str = new char [len+1];
    strncpy ( this -> str, str, len + 1 );
    this -> str [len] = 0;
}
```

```
CStr a ("Test dlouheho retezce" );
CStr b = a(3, 10); // "t dlouh"
```

# Přetěžování operátorů ++ a --

- Operátor má dvě varianty – prefixovou a postfixovou.
- Rozlišení – dummy parametr **int pro postfixovou variantu.**
- Rozdíl – postfixová varianta vrací ještě nemodifikovanou instanci (je dražší).

```
class CStr
{ char * str;
  int len;
public:
  // priklad – operator -- zkrati retezec o 1 znak
  CStr operator -- ( void );
  // prefixova varianta
  CStr operator -- ( int );
  // postfixova varianta.
};
```

# Přetěžování operátorů ++ a --

```
CStr CStr::operator -- ( void )  
{  
    if ( len ) str[--len] = 0;  
    return ( *this );  
}
```

```
CStr CStr::operator -- ( int )  
{  
    CStr res = *this; // kopirující konstruktor  
    --(*this);      // this -> operator -- ();  
    return res;     // původní nezměněná hodnota  
}
```

```
CStr a ( "test retezce" );  
cout << a --;  
cout << -- a;
```

# Přetěžování operátorů přetypování

- Přetížit lze operátory přetypování.
- Musí se postupovat velmi obezřetně – zejména u přetypování na skalární typy se standardními konverzemi.

```
class CStr
```

```
{
```

```
    char * str;
```

```
    int len;
```

```
    public:
```

```
        // priklad – pretypovani na const char *
```

```
        operator const char * ( void ) const;
```

```
};
```

# Přetěžování operátorů přetypování

```
CStr::operator const char * ( void ) const  
{  
    return str;  
}
```

```
CStr a;
```

```
cout << strlen ( (const char *) a );
```

```
cout << strlen ( a ); // přetypování se aut. vloží
```

```
cout << a;          // přetypování na const char *  
                  // nebo op << pro CStr?
```

# Přetěžování operátorů přetypování

```
class CCplx
{
public:
    CCplx ( double r=0, double i=0 );
    operator double ( void ) const;
    friend CCplx operator + ( const CCplx &a,
                             const CCplx & b );
protected:
    double m_Re, m_Im;
};
```

```
CCplx a ( 10, 20 ), b;
```

```
b = 4 + a; /* ?
```

```
error: ambiguous overload for 'operator+' in '4 + a'
candidates are: operator+(int, double) <built-in>
CCplx operator+(const CCplx&, const CCplx&) */
```



# Literatura

- C++ Primer (4th Edition) by Stanley B. Lippman, kapitola 7
- C++ Primer Plus (5th Edition) by Stephen Prata, kapitola 7
- Slajdy Ladislava Vagnera

Děkuji Vám za pozornost