**Pokročilé programování**
**v jazyce C pro chemiky**
(C3220)

# Reference, konstantní metody, přetížené funkce a operátory

# Předávání parametrů hodnotou

- Způsob kterým se obvykle předávají parametry funkcím (nebo metodám) se nazývá předávání hodnotou
- Při zavolání funkce se parametry funkce chovají jako lokální proměnné dané funkce, pro které se alokuje potřebná paměť a zkopírují se do nich předávané hodnoty
- Pokud měníme hodnoty parametrů uvnitř funkce, žádným způsobem tím neovlivníme hodnoty proměnných které byly předávány do funkce

```
  // V pameti se vytvori nova promenna n do ktere se
  // zkopiruje predavana hodnota (v tomto pripade 1)
void myFunction(int n)
{
  n = 5;
    // Do n se priradi 5, po opusteni funkce vsak promenna n
    // zanikne a jeji hodnota se navzdy ztrati
}

int main()
{
  int a = 1;
  myFunction(a);
  cout << a;          // Vypise se cislo 1
  return 0;
}
```

# Předávání parametrů referencí

- Při předávání parametrů referencí nedochází k alokaci paměti pro nové proměnné, ale pouze jména parametrů jsou použita pro přístup k proměnným které byly do funkce předány
- Parametry, které mají být předávány referencí musí mít mezi typem a jménem parametru uvedený znak **&**
- Reference jsou v podstatě ukazatele (viz. přednášky z C), které se automaticky dereferencují

```
  // V pameti se nevytvari nova promenna, pouze nove
  // jmeno n je pouzito pro praci s puvodni promenou a
void myFunction(int &n)     // Reference se specifikuje pomoci &
{
  n = 5;
    // Hodnota 5 se ve skutecnosti priradi puvodni promenne a
}

int main()
{
  int a = 1;
  myFunction(a);
  cout << a;          // Vypise se cislo 5
  return 0;
}
```

# Předávání parametrů referencí - příklad

```
// Funkce swapNumbers() zameni obsah dvou ciselnych promennych

void swapNumbers(int &n1, int &n2) // Parametry predavane referenci
{
  int n3 = 0;  // Pomocna promenna
  n3 = n2;
  n2 = n1;
  n1 = n3;
}

int main()
{
  int a = 3, b = 7;

  cout << a << ", " << b;   // Vypise se: 3, 7
  swapNumbers(a, b);
  cout << a << ", " << b;   // Vypise se: 7, 3

  // Pokud by funkce swapNumbers() neprijimala parametry
  // referenci, ale hodnotou, k zadne zamene hodnot v promennych
  // a, b by nedoslo a vypsala by se cisla 3, 7

  return 0;
}
```

# Předávání parametrů referencí

- Reference používáme také v případech kdy předávaná proměnná zabírá v paměti hodně místa a vytváření její kopie (při předávání hodnotou) by bylo spojeno s velkými paměťovými a výpočetními nároky
- Toto se týká zejména objektových proměnných které předáváme téměř vždy referencí

```
// Nekde na zacatku je definovana trida Circle
// Nasledujici metoda je ze cviceni 2, uloha 4
void Circle::setAverageCircle(Circle &circ1, Circle &circ2)
{
  x = (circ1.getCentreX() + circ2.getCentreX()) / 2;
  y = (circ1.getCentreY() + circ2.getCentreY()) / 2;
  radius = (circ1.getRadius() + circ2.getRadius()) / 2;
}

int main()
{
  Circle circ1(250, 250, 80, 7);
  Circle circ2(250, 250, 80, 3);
  Circle circ3(0, 0, 0, 19);

  circ3.setAverageCircle(circ1, circ2);
  return 0;
}
```

# Předávání parametrů konstantní referencí

- Předávání proměnných referencí používáme především v následujících dvou situacích:
  1) Pokud potřebujeme aby uvnitř volané funkce nebo metody mohla být modifikována hodnota předávané proměnné (např. jako v příkladu funkce `swapNumbers(int &a, int &b)`)
  2) Pokud chceme pouze zajistit vyšší efektivitu programu při předávání objektových proměnných, protože při použití referencí nebude docházet k alokaci paměti pro parametr a kopírování hodnoty do něj
- Případ 2) se používá v situacích, kdy nebude hodnota předávané proměné modifikována (narozdíl od případu 1)). Abychom zaručili, že skutečně nedojde ke změně hodnoty předávané proměnné, deklarujeme parametr jako konstantní použitím klíčové slova **`const`**

```
void Circle::setAverageCircle(const Circle &circ1, const Circle &circ2)
{
  // Zde bude kod metody
}
```

# Předávání parametrů konstantní referencí

- Hodnoty parametrů předaných konstantní referencí nelze měnit (překladač by ohlásil chybu)

```
// Nekde na zacatku je definovana trida Circle

class Circle
{
  public:
    void testNonConstant(Circle &circ);
    void testConstant(const Circle &circ);
    int x, y; // Verejne cleny tridy (jen pro potreby tohoto prikladu)
}

void Circle::testNonConstant(Circle &circ)
{
  circ.x = 12;  // Tohle bude fungovat, protoze circ je nekonstantni
                // parametr (a x je verejny clen tridy Circle)
}

void Circle::testConstant(const Circle &circ)
{
  circ.x = 12; // Tohle nebude fungovat, prekladac ohlasi chybu,
               // protoze circ je konstantni parametr
}
```

# Konstantní metody

- Pokud by funkce přijala konstantní parametr a zavolali bychom jeho metodu, mohla by tato metoda modifikovat data daného parametru což by porušilo požadavek na konstantnost (neměnnost hodnoty) tohoto parametru
- Pro konstantní parametry proto můžeme volat pouze tzv. konstantní metody, ostatní metody volat nelze (překladač by ohlásil chybu)
- Konstantní metody jsou specifikovány pomocí klíčového slova **`const`**, které musí být uvedeno v deklaraci ve třídě i v definici metody mimo třídu
- Uvnitř konstantní metody nelze měnit hodnotu žádného z datových členů třídy ani volat nekonstantní metodu dané třídy
- Tento přístup garantuje, že proměnná která bude předána do funkce jako konstantní parametr (referencí) nebude nijak modifikována, ani přímo ani prostřednitvím volání metody
- V praxi definujeme jako konstantní všechny metody, které neprovádí modifikaci dat dané třídy (výjimkou může být situace, kdy existuje vážný předpoklad, že v rámci budoucího vývoje programu by daná metoda mohla potřebovat tato data modifikovat)

# Konstantní metody - příklad

```
class Circle
{
  public:
    void nonConstantMethod();
    void constantMethod() const;
    void test(const Circle &circ);
    int x, y; // Verejne cleny tridy
}

void Circle::nonConstantMethod()
{
  x = 12;  // Tato metoda muze menit sva clenska data
}

void Circle::constantMethod() const
{
  // Tato konstantni metoda nemuze menit sva clenska data
  x = 12;  // Tady by prekladac nahlasil chybu !!!
}

void Circle::test(const Circle &circ)
{
  // Pro konstantni parametr muzeme volat konstantni metodu
  circ.constantMethod();
  // Nemuzeme vsak volat nekonstantni metodu (pro konstantni parametr)
  circ.nonConstantMethod(); // Prekladac by zde ohlasil chybu !!!
}
```

# Přetížení funkcí a metod

- V jazyce C++ lze definovat více funkcí/metod se stejným jménem ale různým počtem nebo typem parametrů
- Takovéto funkce se nazývají přetížené
- Je-li funkce volána, překladač analyzuje parametry které funkci předáváme a podle toho vybere odpovídající funkci/metodu

```
class Cicle
{
  public:
    void setColor(int c);
    void setColor(const string &colorName);
};

void Circle::setColor(int c)     { color = c; }

void Circle::setColor(const string &colorName)
{ if (colorName == "red") color = 19; else if ... }

int main()
{
  Circle circ;
  circ.setColor(3);          // Vola se prvni metoda
  circ.setColor("red");      // Vola se druha metoda
  return 0;
}
```

# Přetížení funkcí - příklad

```
class FilledCicle
{
  public:
    void setValues(int ax, int ay);
    void setValues(int ax, int ay, int c);
    void setValues(int ax, int ay, int c, int fc);
};


void FilledCicle::setValues(int x, int y)
{ x = ax; y = ay; }

void FilledCicle::setValues(int x, int y, int c)
{ x = ax; y = ay; color = c; }

void FilledCicle::setValues(int x, int y, int c, int fc)
{ x = ax; y = ay; color = c; fillColor = fc; }

int main()
{
  FilledCircle circ;
  circ.setValues(150, 230, 3);       // Vola se druha metoda
  circ.setValues(150, 230, 3, 17);   // Vola se treti metoda
  circ.setValues(150, 230);          // Vola se prvni metoda

  return 0;
}
```

# Přetížení konstruktorů

- Přetížit lze i konstruktory

```
class Circle
{
public:
  Circle();
  Circle(int ax, int ay);
  Circle(int ax, int ay, int c);
};

Circle::Circle()
{ x = 0; y = 0; color = 1; }

Circle::Circle(int ax, int ay)
{ x = ax; y = ay; color = 1; }

Circle::Circle(int ax, int ay, int c)
{ x = ax; y = ay; color = c; }

int main()
{
  Circle circ1;              // Tady se vola prvni konstruktor
  Circle circ2(150, 230);    // Tady se vola druhy konstruktor
  Circle circ3(150, 230, 3); // Tady se vola treti konstruktor
  return 0;
}
```

# Přetížené operátory

- Jazyk C a C++ obsahuje interní definice operátorů pro základní datové typy (`int`, `double`, `char` ...)
- V C++ je kromě toho možné definovat operátory pro uživatelské typy, tj. pro objektové proměnné
- Operátory se definují podobně jako funkce (a metody) pouze místo názvu funkce použijeme klíčové slovo **`operator`** a za ním uvedeme příslušný znak operátoru
- Přetížit lze prakticky všechny dostupné operátory, nejčastěji se přetěžují operátory: `=`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `[]`, `<<`, `>>`, `+`, `-`, `*`, `/`.
- V praxi používáme přetížené operátory pouze vzácně, jejich nadměrné používání může program znepřehlednit (zejména tam, kde nelze význam operátoru snadno odhadnout z kontextu).

# Přetížené operátory - příklad

```
class Vector2D
{
  public:
    Vector2D();
    double getX() const;
    double getY() const;
    void set(double ax, double ay);
  private:
    double x, y;
};

// Operator vrati skalarni soucin dvou vektoru
double operator*(const Vector2D &v1, const Vector2D &v2)
{
  return (v1.getX() * v2.getX() + v1.getY() * v2.getY());
}

int main()
{
  Vector2D v1, v2;
  double result = 0;

  result = v1 * v2;        // Tady se zavola operator*

  return 0;
}
```

# Přetížené operátory jako metody

- Operátory lze implementovat také jeko metody
- Volá se vždy metoda objektu na levé straně od operátoru a jako argument se mu předá objekt na pravé straně

```
class Vector2D
{
public:
  // Tady bude deklarace konstruktoru, promennych atd.
  double operator*(const Vector2D &v) const;
};

double Vector2D::operator*(const Vector2D &v) const
{
  return (x * v.getX() + y * v.getY());
}

int main()
{
  Vector2D v1, v2;
  double result = 0;

  result = v1 * v2;     // Tady se zavola operator* objektu v1
                        // a jako parametr se mu preda objekt v2
  return 0;
}
```

# Přetížené operátory proudů

```
// Na zacatku je definovana trida Vector2D

// Operator pro zapis promenne typu Vector2D do proudu
// Zapisovanou promennou predavame konstantni referenci,
// proud os vsak predavame nekonstantni referenci, protoze pri
// zapisu se meni stav vnitrnich promennych proudu
ostream& operator<< (ostream &os, const Vector2D &v)
{
  os << v.getX() << " " << v.getY() << endl;
  return os;
}

// Operator pro cteni promenne typu Vector2D z proudu
istream& operator>> (istream &os, Vector2D &v)
{
  double ax = 0.0, ay = 0.0;
  os >> ax >> ay;
  v.set(ax, ay);
  return os;
}

int main()
{
  Vector2D v1, v2, v3;
  cout << v1 << v2 << v3; // Vola se vyse definovany operator <<
  cin >> v1 >> v2 >> v3;  // Vola se vyse definovany operator >>
  return 0;
}
```

# Dodržujte následující pravidla

- Parametry objektových typů předávejte jako konstantní referenci, pokud neexistují důvody pro jiný přístup. Toto platí i pro řetězcové proměnné typu `string`.
- Parametry základních typů (`int`, `double` ...) předávejte obvyklým způsobem (tj. hodnotou a nekonstantní), pokud neexistují důvody pro jiný přístup.
- Všechny metody ve třídách, které nemění hodnoty datových členů třídy, definujte jako konstantní
- Operátory implementuje přednostně jako metody třídy. Pouze tam kde to není možné je implementujte jako funkce, např. pokud je operandem nalevo od operátoru neobjektová proměnná (`int`, `double`, ...) nebo pokud je jím objekt třídy, která je zabudovaná v knihovně a nelze do ní tedy přidávat metody/operátory (např. třídy proudů).

# Cvičení – 1. část

1. Upravte program z úlohy 2 z předchozího cvičení následujícím způsobem:
   - Ve třídě `Graphic` implementujte metodu `getColorNumberFromString()`, která bude přijímat název barvy jako parametr (black, blue, green, red, yellow) a vracet číslo barvy.
   - Ve třídách `Circle` a `FilledCircle` přidejte další variantu metody `setValues()`, která bude místo čísla barvy (a barvy výplně) přijímat text z názvem barvy (původní variantu metody však také ponechte).
   - V každé ze tříd `Graphic`, `Circle`, `FilledCircle` a `Rectangle` implementujte dva konstruktory, jeden bez parametrů a jeden s parametry, které budou nastavovat hodnoty členů tříd.
   - Všechny objektové parametry budou do metod předávány jako konstantní reference (týká se i parametrů typu `string`). Všechny metody, které nemění hodnotu datových členů třídy definujte jako konstantní metody.
   - Program upravte tak, aby byl schopen načítat soubor *graphic2.dat* (v adresáři */home/martinp/C3220/data/*), který se od *graphic1.dat* liší tím, že jsou v něm barvy specifikovány formou textu namísto čísel. **1 bod**

# Cvičení – 2. část

2. Vytvořte program pro práci s 3D vektory. V programu definujte třídu `Vector3D`, která bude obsahovat souřadnice vektoru `x`, `y`, `z` (typu `double`). Dále bude obsahovat následující metody:
   - Konstruktor bez parametrů `Vector3D()` a s parametry `Vector3D(double ax, double ay, double az)`
   - Metody `getX()`, `getY()`, `getZ()` pro získání hodnot souřadnic a metodu `set(double ax, double ay, double az)` pro jejich nastavení.
   - Metodu `readValues()`, která si od uživatele vyžádá zadání tří souřadnic a načte je a metodu `printValues()`, která vypíše souřadnice vektoru. Obě metody budou přijímat parametr typu `string`, který se uživateli vypíše před tím než se načtou/vypíší souřadnice.
   - Operátor * pro skalární součin a operátor + pro vektorový součet.
   - Funkci `swapVectors()`, která zamění dva vektory (hodnoty jejich souřadnic). Bude se jednat o samostatnou funkci, ne o metodu.
   - Objektové parametry předávejte do metod (vč. operátorů) jako konstantní reference (týká se i parametrů typu `string`). Všechny metody, které nemění hodnotu datových členů třídy definujte jako konstantní metody. Operátory implementujte ve třídách, pokud je to možné.

   Program si od uživatele vyžádá souřadnice dvou vektorů a na obrazovku vypíše souřadnice vektorového součtu a dále hodnotu skalárního součinu vektorů. Pak zavolá funkci `swapVectors()` pro ty zadané vektory a vypíše jejich souřadnice. **3 body**

# Cvičení – 3. část

3. Předchozí program modifikujte tak, že v něm implementujete operátory **>>** a **<<** pro načítání/zápis do proudu. Program načte ze souboru *vectors.dat* (v adresáři */home/martinp/C3220/data/*) souřadnice dvou vektorů a do výstupního souboru zapíše souřadnice vektorového součtu těchto dvou vektorů. Jména vstupního a výstupního souboru budou specifikovány na příkazovém řádku. Pro načítání a zápis do souboru využijte implementované operátory. **nepovinná, 2 body**