Pokročilé programování
v jazyce C pro chemiky
(C3220)

Reference, konstantní a statické metody, přetížené funkce a operátory

Reference

- Reference jsou speciálním typem proměnných, které samy neuchovávají žádnou hodnotu, jen odkazují na jiné proměnné
- Reference jsou v podstatě ukazatele (jako v C), které se ale automaticky dereferencují (není třeba používat * a ->)
- Reference se definují přidáním znaku **&** mezi typ a jméno proměnné
- Reference je v okamžiku vytvoření pevně svázána s odkazovanou proměnnou, nelze ji „odpojit“ či „přesměrovat“

```
int a = 1, b = 2;
int &r = a;

a = 3;
cout << r;          // Vypise se cislo 3

// Aktualni hodnota promenne b bude zkopirovana do promenne,
// na kterou odkazuje reference r (tedy do a)
r = b;

b = 4;
cout << r;          // Vypise se cislo 2
```

Předávání parametrů hodnotou

- Způsob, kterým se obvykle v C/C++ 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 (argumentů)

```
  // 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 odkazem

- Při předávání parametrů odkazem (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 **&**

```
  // V pameti se nevytvari nova promenna, pouze nova
  // reference n je pouzita pro praci s puvodni promenou a
void myFunction(int &n)
{
  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
- Některé objektové proměnné kopírovat vůbec nelze (například tehdy, když reprezentují nějaký externí objekt dle principu RAII)

```
// 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 ke zbytečnému kopírování
- 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 referenci 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)

```
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 objektový parametr a zavolali bychom jeho metodu, mohla by tato metoda modifikovat data daného objektu, což by porušilo požadavek na konstantnost
- 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`** za seznamem parametrů, 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 metody dané třídy
- Tento přístup garantuje, že proměnná, která bude předána do funkce jako konstantní parametr (referencí) nebude nijak modifikována; ani přímo, ani prostřednictvím 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 !!!
}
```

Konstantní proměnné

- Konstantní proměnné specifikujeme s klíčovým slovem **`const`**, tyto konstantní proměnné musí mít hodnotu inicializovanou při definici, jejich hodnotu nelze v průběhu programu měnit
- Konstantní proměnné mohou být lokální, globální nebo členy třídy
- V C++ používáme globální konstantní proměnné namísto symbolických konstant definovaných direktivou #define používaných v jazyce C
- Globální konstantní proměnné lze využívat např. pro specifikaci velikosti polí
- Názvy globálních konstant často píšeme velkými písmeny

```
const int MAX_ITEMS = 1000;

int main()
{
  int itemsArray[MAX_ITEMS];

  return 0;
}
```

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 (*overloaded*)
- Je-li funkce volána, překladač analyzuje předávané argumenty a podle nich 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 ax, int ay)
{ x = ax; y = ay; }

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

void FilledCicle::setValues(int ax, int ay, int c, int fc)
{ setValues(ax, ay, 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

- Velmi často se přetěžování užívá pro konstruktory

```
class Circle
{
public:
  Circle();
  Circle(int ax, int ay);
  Circle(int ax, int ay, int c);
private:
  int x = 0, y = 0, color = 1;
};

Circle::Circle() {} // vse dostane vychozi hodnoty

Circle::Circle(int ax, int ay) : x(ax), y(ay) {} // vychozi jen barva

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 téměř všechny dostupné operátory, nejčastěji se přetěžují operátory: `=`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `[]`, `<<`, `>>`, `+`, `-`, `*`, `/`.
- V praxi vytváříme vlastní přetížené operátory pouze v odůvodněných případech, jejich nadužívání 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 pracující nad objekty implementujeme přednostně jako metody
- Volá se vždy metoda objektu na levé straně operátoru a jako argument se mu předá objekt na pravé straně (kromě unárních operátorů jako `!`, `->`, `++` a `--`, ty žádný druhý objekt nemají)

Přetížené operátory jako metody

```
class Vector2D
{
public:
  double operator*(const Vector2D &v) const;
  bool operator!(void) const;
};

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

bool Vector2D::operator!(void) const
{
  return (x == 0 && y == 0);
}

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

  if (!v1) {           // Tady se zavola operator! objektu v1
    cout << "Vektor v1 je nulovy!" << endl;
  }

  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; // Proud musime vzdy vratit, aby slo operace retezit
}

// 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;
}
```

Statické členy třídy

- Statické členy třídy deklarujeme s klíčovým slovem **`static`**
- Statické proměnné se chovají podobně jako globální proměnné, program pro ně alokuje paměť ihned po spuštění programu, při vytváření objektových proměnných se již pro ně žádná další paměť nealokuje.
- Statické členské proměnné nejsou svázány s žádnou konkrétní instancí třídy (všechny instance sdílí tutéž jednu proměnnou)
- Statické členské proměnné tedy můžeme použít, aniž bychom vytvořili příslušný objekt (tj. definovali objektovou proměnnou).
- Ke statickým proměnným přistupujeme zvenčí přes jméno třídy a **`::`** (`JmenoTridy::jmenoPromenne`). Pokud však k nim přistupujeme z metod téže třídy, stačí použít přímo jméno proměnné.
- Statické proměnné inicializujeme v samostatné definici mimo třídu (narozdíl od nestatických proměnných, které musíme inicializovat v těle třídy či v konstruktoru).
- Statické proměnné často definujeme v sekci **`public`**, aby byly přístupné i z funkcí a metod jiných tříd (pokud však toto nepotřebujeme, mohou být **`protected`** nebo **`private`**).

Statické členy třídy - příklad

```
class Vector3D
{
  public:
      // Nasledujici staticka promenna bude obsahovat celkovy pocet
      // aktualne existujicich promennych typu Vector3D
    static int totalVectCount;
    Vector3D();  // Konstruktor
    ~Vector3D(); // Destruktor
};

int Vector3D::totalVectCount = 0;  // Inicializace staticke promenne

Vector3D::Vector3D()  { totalVectCount++; }

Vector3D::~Vector3D() { totalVectCount--; }

int main()
{
  Vector3D v1, v2, v3; // Vola se trikrat konstruktor Vector3D()

    // Pokud volame staticke promenne mimo metody tridy, musime
    // pred jmeno promenne uvest jmeno tridy oddelene ctyrteckou
  cout << "Celkovy pocet vektoru" << Vector3D::totalVectCount << endl;

  return 0;
}
```

Konstantní statické členy třídy

- Statické proměnné můžeme zároveň definovat jako konstantní tam, kde je to vhodné. Pokud jsou typu int, můžeme je inicializovat přímo ve třídě.

```
class Shape
{
  public:
      // Nasledujici konstantni staticka promenna bude pouzita
      // pro cislo modre barvy
    static const int COLOR_BLUE = 3;
    void draw(int dev);
};

void Shape::draw(int dev)
{
  g2_pen(dev, COLOR_BLUE);
    // Dalsi kod metody
}

int main()
{
  Shape s1;

  cout << "Modra barva ma cislo: " << Shape::COLOR_BLUE << endl;

  return 0;
}
```

Statické metody

- Statické metody deklarujeme s klíčovým slovem **`static`**
- Klíčové slovo **`static`** uvádíme pouze v deklaraci ve třídě, v definici mimo třídu již ne
- Statické metody se chovají se podobně jako nečlenské funkce, tj. můžeme je volat, aniž bychom vytvořili objekt (tj. definovali objektovou proměnnou)
- Ke statickým metodám přistupujeme přes jméno třídy a **`::`** (`JmenoTridy::jmenoMetody()`), pokud je však voláme z metod třídy, stačí použít přímo jméno metody
- Statické metody mohou operovat pouze nad statickými daty dané třídy
- Výhodou statických metod oproti nečlenským funkcím je vyšší přehlednost programu a zejména zabránění kolizím v názvech (několik tříd může mít statickou metodu se stejným názvem)

Statické metody - příklad

```
class Shape
{
  public:
    static int getColorNumberFromString(const string &colorName);
      // Definice dalsich clenu tridy
};

  // Tady jiz slovo static neuvadime
int Shape::getColorNumberFromString(const string &colorName)
{
  if (colorName == "blue")
    return 3;
  return 0;  // Pokud je nazev barvy neznamy, vrati 0
}


int main()
{
    // Metodu Shape::getColorNumberFromString() muzeme volat aniz
    // bychom definovali promennou tridy Shape
  cout << "Cislo modre barvy: "
       << Shape::getColorNumberFromString("blue") << endl;

  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 Shape implementujte statickou metodu `getColorNumberFromString()`, která bude přijímat název barvy jako parametr (black, blue, green, red, yellow) a vracet číslo barvy.
   - Ve třídě Shape vytvořte statické konstantní proměnné pro čísla barev (pro barvy: 0 white, 1 black, 3 blue, 7 green, 19 red, 25 yellow)
   - Ve třídách `Circle` a `FilledCircle` přidejte další variantu metody `setValues()`, která bude místo čísla barvy (a barvy výplně) přijímat text s názvem barvy (původní variantu metody však také ponechte).
   - V každé ze tříd Shape, `Circle`, `FilledCircle` a `Rectangle` implementujte dva konstruktory, jeden bez parametrů (ponechávající výchozí hodnoty určené v těle třídy) a jeden s parametry, které bude ukládat do členů dané třídy
   - 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 `shapes2.dat` (v adresáři `/home/tootea/C3220/data/`), který se od `shapes1.dat` liší tím, že jsou v něm barvy specifikovány formou textu místo čísel. **2 body**

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()` (výchozí nulové souřadnice) 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 navíc přijímat parametr typu `string`, který se uživateli vypíše před tím než se načtou/vypíší souřadnice.
   - Operátor ***** pro skalární součin a operátor **+** pro součet vektorů.
   - Samostatnou funkci `swapVectors()` (ne metodu), která zamění dva vektory (hodnoty jejich souřadnic)
   - 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 pokud možno jako metody.

   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 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/tootea/C3220/data/`) souřadnice dvou vektorů a do výstupního souboru zapíše souřadnice součtu těchto dvou vektorů. Jména vstupního a výstupního souboru budou specifikována na příkazovém řádku. Pro načítání a zápis do souboru využijte implementované operátory. **nepovinná, 2 body**