Pokročilé programování v jazyce C pro chemiky

(C3220)

Statické proměnné a metody, šablony v C++

Globální konstantní proměnné

- Konstantní proměnné specifikujeme s klíčovým slovem **`const`**, tyto konstantní proměné 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 konstantních proměnných často píšeme velkými písmeny

```
const int MAX_ITEMS = 1000;

int main()
{
  int itemsArray[MAX_ITEMS];

  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 proměnné tedy můžeme použít aniž bychom vytvořili příslušný objekt (tj. definovali objektovou proměnnou).
- Ke statickým proměnným přistupujeme přes jméno třídy a :: (Jmeno_tridy::jmeno_promenne). 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 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 typy 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 jemno tridy oddelene dvema dvojteckama
  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 Graphic
{
  public:
      // Nasledujici konstantni staticka promenna bude pouzita
      // pro cislo modre barvy
    static const int COLOR_BLUE = 3;
    void draw(int dev);
};

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

int main()
{
  Graphic g1;

  cout << "Modra barva ma cislo: " << Graphic::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 :: (Jmeno_tridy::jmeno_metody()), 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 Graphic
{
  public:
    static int getColorNumberFromString(const string &colorName);
      // Definice dalsich clenu tridy
};

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


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

  return 0;
}
```

Šablony funkcí

- Šablony funkcí používáme tam kde potřebujeme použít stejnou funkci pro odlišné typy argumentů a návratových hodnot
- Šablonu definujeme podobně jako funkci ale před hlavičkou funkce uvedeme `template <typename T>` nebo `template <class T>` kde T je volitelný symbol zastupující jméno typu
- Typů můžeme specifikovat i více: `template <typename T1, typename T2, typename T3>`
- V okamžiku volání jména šablony posoudí překladač typ předávaných argumentů a vygeneruje příslušnou funkci
- Šablony funkcí se v praxi používají jen v malé míře, ve standardní knihovně jsou však definovány některé užitečné šablonové funkce (např. `swap()`, `min()`, `max()`)
- Šablony lze definovat také pro metody (podobným způsobem jako pro funkce)

Šablony funkcí – příklad 1

```
  // Klasicke reseni bez pouziti sablon (pouzivame pretizeni funkci)
int getMax(int value1, int value2)
{ if (value1 > value2) return value1; else return value2; }

double getMax(double value1, double value2)
{ if (value1 > value2) return value1; else return value2; }

int main()
{
  int i1 = 3, i2 = 5, imax = 0;
  double d1 = 2.12, d2 = 6.78, dmax = 0.0;
  imax = getMax(i1, i2);  // Tady se vola prvni funkce
  dmax = getMax(d1, d2);  // Tady se vola druha funkce
  return 0;
}
```

```
  // Reseni s pouzitim sablony funkce
template <typename T> T getMax(T value1, T value2)
{ if (value1 > value2) return value1; else return value2;  }

int main()
{
  int i1 = 3, i2 = 5, imax = 0;
  double d1 = 2.12, d2 = 6.78, dmax = 0.0;
  imax = getMax(i1, i2); // Na zaklade sablony vygeneruje funkci pro int
  dmax = getMax(d1, d2); // Na zaklade sablony vygeneruje funkci pro double
  return 0;
}
```

Šablony funkcí – příklad 2

```
  // Sablona funkce pro zamenu dvou hodnot
template <typename T> void swapItems(T &item1, T &item2)
{
  T itemAux;
  itemAux = item1;
  item1 = item2;
  item2 = itemAux;
}

int main()
{
  int i1 = 3, i2 = 5;
  string s1 = "Prvni retezec", s2 = "Druhy retezec";
  Vector3D v1(3.1, 4.7, -9.0), v2(6.4, -1.2, 7.8);

  swapItems(i1, i2);  // Vygeneruje se funkce pro typ int
  swapItems(s1, s2);  // Vygeneruje se funkce pro typ string
  swapItems(v1, v2);  // Vygeneruje se funkce pro typ Vector3D

  return 0;
}
```

Šablony tříd

- Šablony tříd fungují podobným způsobem jako šablony funkcí, ale na jejich základě se generují celé třídy
- Šablony tříd se v praxi používají jen v malé míře, standardní knihovna však šablony tříd hojně využívá
- Příslušné objektové proměnné deklarujeme:
  `jmeno_sablony<jmeno_typu> jmeno_promenne`

```
template <typename T> class Vector3D // Sablona tridy pro 3D vektor
{
  public:
    Vector3D() { x = 0; y = 0; z = 0; };
    Vector3D(T ax, T ay, T az) { x = ax; y = ay; z = az; };
      // Zde budou definovany dalsi cleny sablony tridy
  private:
    T x, y, z;
};

int main()
{
  Vector3D<int> vectInt;
  Vector3D<double> vectDouble;
    // Zde je mozne pracovat s objektovymi promennymi vectInt a
    // vectDouble obvyklym zpusobem
}
```

Šablony tříd - příklad

```
  // Nasledujici priklad ukazuje jak definovat metodu mimo sablonu
template <typename T> class Vector3D
{
  public:
    Vector3D() { x = 0; y = 0; z = 0; }
    void printValues();  // Metoda bude definovana mimo sablonu tridy
      // Zde budou definovany dalsi cleny sablony tridy
  private:
    T x, y, z;
};

template<typename T> void Vector3D<T>::printValues()
{
  cout << "Souradnice vektoru:" << x << y << z << endl;
}

int main()
{
  Vector3D<int> vectInt;
  Vector3D<double> vectDouble;

  vectInt.printValues();
  vectDouble.printValues();

  return 0;
}
```

STL – standardní šablonová knihovna

- Součástí standardní knihovny jazyka C++ je standardní šablonová knihovna STL (standard template library)
- STL obsahuje například šablony funkcí implementující některé algoritmy nad souborem dat (třídění, vyhledávání, kopírování)
- Z šablonových tříd jsou nejpoužívanější kontejnery (zásobníky), které slouží k ukládání objektů libovolného typu
- Nejpoužívanějším kontejnerem je šablona `vector`, která je zobecněním polí, narozdíl od klasických polí není počet prvků v kontejnerech `vector` fixní, ale zvyšuje se podle počtu vložených prvků (velikost je limitována pouze dostupnou pamětí)
- Podrobnější dokumentace k STL na http://www.cppreference.com/wiki/

Kontejner `vector`

- Pro použití kontejneru `vector`, musíme vložit hlavičkový soubor `#include <vector>` a deklarovat `using namespace std;`
- Kontejner vector obsahuje následující metody:
  - `operator[]` - vrací příslušný prvek podobně jako klasické pole
  - `front()` - vrací první prvek
  - `back()` - vrací poslední prvek
  - `push_back()` - vloží prvek na konec (resp. jeho kopii)
  - `pop_back()` - vyjímá poslední prvek (nevrací nic)
  - `clear()` - vyjímá všechny prvky (kontejner je pak prázdný)
  - `insert()` – vkládá na prvek na specifikovanou pozici
  - `size()` - vrací aktuální počet prvků
  - `empty()` - vrací informaci o tom, zda je kontejner prázdný (totéž jako size() == 0)
- Operátor `[]` a metody `front()` a `back()` vracejí prvky formou referencí a nekontrolují meze (při překročení mezí pole je jejich chování nedefinované)
- Podrobnější výčet všech metod kontejneru `vector` lze najít na: http://www.cplusplus.com/reference/stl/vector/

Kontejner vector - příklad

```
#include <iostream>
#include <vector>
using namespace std;

class Circle
{
  // Tady by byly data a metody tridy Circle – viz drive
};

int main()
{
  Circle circle;
  vector<Circle> circlesContainer;  // Definice kontejneru pro
                                    // objekty Circle

  circle.readFile(sstream);  // Nactou se data do promenne circle
  circlesContainer.push_back(circle); // Kopie prvku se vlozi na konec

    // Cyklus pres prvky kontejneru vector
  for (int i=0; i < circlesContainer.size(); i++)
    circlesContainer[i].draw(dev);

    // Dalsi kod
}
```

Cvičení – část 1

1. Vytvořte program vycházející z programu z úlohy 1 z minulého cvičení s následujícími úpravami:
   - Program bude používat globální konstantní proměnné pro specifikaci meze polí (tj. např. `const int MAX_GRAPHIC = 100;`)
   - Ve třídě `Graphic` vytvořte statické konstantní proměnné pro čísla barev (pro barvy: 0 white, 1 black, 3 blue, 7 green, 19 red, 25 yellow)
   - Metodu `getColorNumberFromString()` ve třídě `Graphic` předělejte na statickou metodu
   - V programu dále vytvořte šablonu pro funkci která v cyklu vykreslí grafické objekty daného typu. Bude přijímat tři argumenty: pole grafických prvků, počet platných prvků v poli a číslo grafického zařízení. Tuto šablonu využijte pro vykreslení grafických objektů.

   Program otestujte se souborem *graphic2.dat* (v adresáři */home/martinp/C3220/data/)*. 1 bod

Cvičení – část 2

2. Modifikujte program následujícím způsobem:
   - Vytvořte šablonu třídy (s názvem např. `Container`) která bude obsahovat pole objektů a počet platných prvků v poli a dále následující metody
     - `Container()` - konstruktor bez parametrů
     - `int add(const T &item)` - přidá prvek (který přijme jako argument) na konec pole vrátí pořadí prvku v poli
     - `T get(int pos) const` - vrátí prvek na dané pozici (pozici prvku přijme jako argument)
     - `T &operator[](int pos)` - totéž jako `get()` ale prvek vrátí referencí
     - `int count() const` - vrátí počet prvků pole
   - Program upravte tak aby pro uložení grafických objektů používal výše uvedenou šablonu `Container` namísto obyčejných polí.

   Pozn.: Pro vykreslování objektů nyní nepoužívejte šablonovou funkci z předchozí úlohy ale vykreslujte podobně jako jako v podobně jako v původní verzi programu. Pro správné fungování programu je potřeba, aby metoda ve které se provádí vykreslování (`Drawing::draw()`) nebyla konstantní, protože se z ní volá nekonstantní operátor `[]` šablony `Container`. 3 body

3. Program modifikujte tak, že místo výše uvedené šablony `Container` bude používat kontejner `vector` ze standardní knihovny. 1 bod