Pokročilé programování v jazyce C pro chemiky

(C3220)

Šablony, knihovna STL, algoritmy

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 samostatných funkcí se v praxi používají v menší 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 intenzivně využívají při tvorbě knihoven, hojně je využívá například standardní knihovna C++
- 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
  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)
- Knihovna STL nabízí mnoho šablonových tříd, z nichž jsou nejpoužívanější kontejnery (zásobníky), které slouží k ukládání objektů libovolného typu
- Kromě kontejneru `vector`, jsou k dispozici například i množiny (`set`) či mapy klíč-hodnota (`map`)
- Podrobnější dokumentace k STL na https://en.cppreference.com/w/cpp/standard_library

Kontejner `map`

- Pro použití kontejneru `unordered_map`, musíme vložit hlavičkový soubor `#include <map>` a deklarovat `using namespace std;`
- Tento kontejner obsahuje páry klíč-hodnota. Každému klíči je přiřazena právě jedna hodnota. Klíče jsou v mapě vždy vzestupně seřazeny (při iteraci je dostáváme od nejmenšího klíče po největší).
- Kontejner map obsahuje například následující metody:
  - `operator[]` - vrací prvek s příslušným klíčem. Pokud prvek s daným klíčem v mapě není, je vložen s výchozí hodnotou a vrácena reference na ni.
  - `erase()` - smaže prvek s daným klíčem, je-li přítomen
  - `count()` - vrací počet prvků s daným klíčem v mapě (0 nebo 1)
  - `clear()` - vyjímá všechny prvky (kontejner je pak prázdný)
  - `size()` - vrací aktuální počet prvků
  - `empty()` - vrací informaci o tom, zda je kontejner prázdný (totéž jako size() == 0)
- Nezáleží-li nám na pořadí prvků v mapě, používáme rychlejší variantu `unordered_map`
- Podrobnější výčet všech metod kontejneru `vector` lze najít na: https://en.cppreference.com/w/cpp/container/map

Kontejner map – příklad

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

int main()
{
  map<string, int> dnyVTydnu;  // Definice kontejneru s textovými klíči
                               // a celočíselnými hodnotami

  dnyVTydnu["pondeli"] = 1;
  dnyVTydnu["utery"] = 2;
  // atd.

  string den = "čtvrtek";

  cout << "Dnes je " << den << ", "
       << dnyVTydnu[den] << ". den v tomto tydnu." << endl;
  // Vypíše: Dnes je čtvrtek, 4. den v tomto týdnu.
}
```

Iterátory

- Mnoho operací s kontejnery v STL používá tzv. iterátory, což jsou speciální objekty podobné ukazatelům, ukazující do daného kontejneru na nějaký prvek
- Každý kontejner má následující metody, vracející příslušný iterátor
- `begin()` - vrací iterátor na první prvek kontejneru
- `end()` - vrací iterátor ukazující těsně za poslední prvek
- Iterátory jsou interně využívány range-based for cykly, ale můžeme je použít i přímo
- Hodnotu, na kterou iterátor ukazuje, získáme dereferencováním pomocí `*` či `->`
- Iterátory můžeme porovnávat pomocí `==` a `!=` a
- K posunu na další prvek použijeme `++`, některé iterátory umí i přejít na předchozí prvek pomocí `--`

Iterátory – příklad

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
  map<string, int> dnyVTydnu;  // Definice kontejneru s textovými klíči
                               // a celočíselnými hodnotami
  map<int, string> cisloNaDen; // Mapa pro překlad obráceným směrem

  dnyVTydnu["pondeli"] = 1;
  // atd.

  // iterujeme od begin(), dokud nenarazíme na end()
  for (map<string,int>::iterator it = dnyVTydnu.begin();
       it != dnyVTydnu.end(); ++it) {
    // u mapy je it->first klíč, it->second příslušná hodnota
    cisloNaDen[it->second] = it->first;
  }
}
```

Dedukce typů pomocí auto

- Často se setkáme se situací, kdy je typ nějaké proměnné “jasný” (např. daný návratovým typem nějaké funkce), ale je pracné jej ručně vypisovat
- Použijeme-li místo typu klíčové slovo `auto`, kompilátor si potřebný typ vydedukuje z kontextu (pokud to nejde jednoznačně, kompilátor ohlásí chybu)
- Místo `map<string, int>::iterator it = kontejner.begin()` tedy stačí jen `auto it = kontejner.begin()`
- Klíčové slovo `auto` můžeme dále dekorovat pomocí `const` či `&` a upřesnit tím, co má kompilátor udělat
- Pro obecnou iteraci tedy můžeme psát
  `for (const auto &x : kontejner) { … }`
- `auto` nelze použít pro typy parametrů funkce (nejsou při definici funkce známy), funguje jen pro proměnné a návratové typy funkcí
- Nadužívání auto ale může kód znepřehlednit, používáme jej jen tam, kde je to prospěšné, nepíšeme tedy `auto i = 5` místo `int i = 5`

Standardní knihovna algoritmů

- STL obsahuje také šablony funkcí implementující některé běžné algoritmy nad souborem dat (třídění, vyhledávání, kopírování)
- Tyto funkce jsou k dispozici v hlavičkovém souboru <algorithm>
- `find()` - najde první odpovídající prvek
- `count_if()` - vrací počet prvků odpovídajících nějakému kritériu
- `reverse()` - obrátí pořadí prvků
- `sort()` - seřadí prvky vzestupně
- `copy()` - zkopíruje daný rozsah do jiného kontejneru
- Většina algoritmů přijímá dva iterátory ukazující na začátek a konec rozsahu hodnot, nad nímž mají pracovat (typicky předáváme `begin()` a `end()` nějakého kontejneru).
- Mnoho dalších funkcí naleznete v detailní dokumentaci na https://en.cppreference.com/w/cpp/header/algorithm

Algoritmy – příklad

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

bool je_sude(int x) { return x % 2 == 0 }

int main()
{
  vector<int> cisla; // naplněný nějakými čísly

  sort(cisla.begin(), cisla.end()); // seřadí cisla vzestupně

  int pocetTrojek = count(cisla.begin(), cisla.end(), 3);

  int pocetSudych = count_if(cisla.begin(), cisla.end(), je_sude);
}
```

Lambda funkce

- STL obsahuje také šablony funkcí implementující některé běžné algoritmy nad souborem dat (třídění, vyhledávání, kopírování)
- Zejména při práci s algoritmy potřebujeme často funkce specifikující kritérium pro počítání, řazení apod.
- Abychom nemuseli pro každý případ vždy definovat globální funkci, existují v jazyku C++ tzv. Lambda funkce, což jsou dočasné (lokální) funkce
- Lambda funkce může mít přístup ke proměnným z bloku, ve kterém je definována
- Lambda funkci definujeme v místě použití takto:
  `[promenna1, promenna2] (int parametr1) { tělo Funkce; }`
- V hranatých závorkách uvedeme seznam vnějších proměnných, které chceme pro lambda funkci zpřístupnit (buďto kopií-hodnotou, nebo jako referenci přidáním & před proměnnou)
- Kulaté závorky specifikují typ parametrů jako u běžné funkce

Lambda funkce – příklad

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

int main()
{
  vector<int> cisla; // naplněný nějakými čísly
  int hranice = 5;

  // lambda s jedním parametrem, zpřístupníme jí proměnnou hranice
  int pocetVetsichNez = count_if(cisla.begin(), cisla.end(),
    [hranice] (int x) { return x > hranice; }
    );

  // lambda s jedním parametrem, nemá přístup k žádným proměnným z
main()
  int pocetSudych = count_if(cisla.begin(), cisla.end(),
    [] (int x) { return x % 2 == 0; }
    );
}
```

Cvičení – část 1

1. Vytvořte program vycházející z programu z úlohy 2 z minulého cvičení s následujícími úpravami:
   - Třída Vector3D bude šablonou, přičemž parametrem šablony bude datový typ T použitý k uložení souřadnic x, y, z
   - Upravte metody getX/Y/Z() a set(), tak aby přijímaly a vracely typ T (ne napevno float či double
   - `operator*` bude vracet vždy typ double
   - Můžete smazat operator+ a funkci swapVectors()
   - Funkce main() vytvoří dvě dvojice vektorů, jednu jako Vector3D<double> a druhou Vector3D<float>
   - Od uživatele vyžádejte a načtěte souřadnice obou double vektorů
   - Každý double vektor zkopírujte do odpovídajícího float vektoru pomocí set() a getX/Y/Z()
   - Nakonec program spočítá a vytiskne skalární součin double vektorů a skalární součin float vektorů.

   Program otestujte s vektory [1, 1, 0] a [1.00000001, -1, 0]. Skalární součin ve vyšší přesnosti by měl být nenulový, v nižší přesnosti nulový. **2 body**

Cvičení – část 2

2. Vytvořte program zpracovávající soubor `data1.dat` (z adresáře `/home/tootea/C3220/data/`). Soubor obsahuje na každém řádku textový identifikátor komplexu protein-ligand, následovaný dvěma hodnotami vazebných energií (referenční a predikovanou). Pro uložení jednotlivých záznamů definujte třídu `Complex`, která bude obsahovat:
   - Tři soukromé datové položky (string a dva double)
   - Metodu `void readLine(const string &line)` pro načtení hodnot do třídy
   - Metodu `void print(void) const` pro výpis hodnot na standardní výstup
   - Metody `getReference()` a `getPredicted()`, vracející hodnoty energií.

   Funkce `main()` vytvoří `vector` tříd `Complex` a do něj po řádcích načte obsah souboru, jehož jméno bude specifikováno jako první argument na příkazovém řádku.

   Vektor komplexů napřed seřaďte vzestupně dle predikovaných hodnot energie voláním `sort()` s vhodnou lambda funkcí (ta bude přijímat dva argumenty `const Complex &` a vracet true, pokud první argument má menší predikovanou energii než druhý). Položky seřazeného vektoru vypište pomocí `print()`.

   (pokračování na další straně)

Cvičení – část 3

2. (pokračování) Vytvořte histogram referenčních energií tak, že použijete mapu, jejímž klíčem bude referenční energie převedená (oříznutá) na `int` a hodnotou bude počet komplexů spadajících pod daný klíč. Výslednou mapu vypište (na každém řádku bude celočíselná energie následovaná počtem).

   Pomocí `count_if` a vhodné lambda funkce spočítejte, pro kolik komplexů je absolutní hodnota (`abs()`) rozdílu referenční a predikované energie menší, než číslo zadané jako druhý argument programu na příkazovém řádku. Výsledný počet vypište. **3 body**