Pokročilé programování v jazyce C pro chemiky

(C3220)

Pokročilá témata jazyka C++

Prostory jmen

- U programů mohou někdy nastat kolize mezi jmény (tříd, funkcí, globálních proměnných atd.) pokud v různých částech programu použijeme neúmyslně stejná jména.
- Podobné kolize nejčastěji nastávají mezi jmény použitými v knihovnách a jmény v programu
- Pokud používáme více knihoven, mohou také nastat konflikty jmen mezi knihovnami
- Uvedeným kolizím lze předejít použitím prostorů jmen (angl. namespaces)
- Prostor jmen deklarujeme následovně:

  namespace jmeno_prostoru { deklarace proměnných, funkcí, tříd }
- Přistupujeme-li ke jménu (třídy, proměnné, funkce) uzavřenému v daném jmenném prostoru z vnitřku tohoto prostoru, používáme jména přímo (tj. tak jako bychom jmené prostory nepoužívali)
- Přistupujeme-li ke jménu z oblasti mimo jmenný prostor v němž je uzavřeno, musíme před každým takovým jménem uvést jméno jmenného prostoru oddělené dvojtečkou: jmeno_prostoru::jmeno

Prostory jmen - příklad

```
namespace mujprostor
{
  int mojePromenna = 0;   // Definice globalni promenne
    // K promenne mojePromenna muzeme v nasledujici funkci pristupovat
    // primo, protoze je definovana ve stejnem jmennem prostoru
  void mojeFunkce() { mojePromenna = 8; }
  class MojeTrida
  {
    public:
       // K promenne mojePromenna a funkci mojeFunkce() muzeme
       // pristupovat primo, protoze jsou definovany ve stejnem
       // jmennem prostoru jako tato trida
      void print() { mojePromenna = 8; mojeFunkce(); };
  }
}

int main()
{
  // Pro pristup ke jmenum ve jmennem prostoru musime uvest jmeno
  // tohoto prostoru (oddelene ::) pred kazdym jmenem z toho prostoru
  mujprostor::mojePromenna = 100;
  mujprostor::mojeFunkce();
  mujprostor::MojeTrida test;
  test.print();  // K promenne test uz pristupujeme primo, protoze
                 // neni definovana ve jmennem prostoru mujprostor
}
```

Deklarace `using`

- Pomocí klíčového slova `using` lze zpřístupnit některá jména z prostoru jmen tak abychom s nimi mohli pracovat přímo

```
namespace mujprostor
{
  // Zde bude definovana promenna, funkce a trida jako na predchozi
  // strance
}

int main()
{
    // Nasledujici deklarace nam umozni pristupovat k promenne
    // mojePromenna primo bez uvadeni jmena prostoru
  using mujprostor::mojePromenna;
  mojePromenna = 100;

    // K ostatnim jmenum vsak muzime pristupovat stejne jako
    // v predchozim pripade
  mujprostor::mojeFunkce();
  mujprostor::MojeTrida test;
  test.print();
}
```

Deklarace `using namespace`

- Deklarace `using namespace` umožňuje zpřístupnit všechna jména z daného prostoru jmen

```
namespace mujprostor
{
  // Zde bude definovana promenna, funkce a trida jako na predchozi
  // strance
}

int main()
{
    // Nasledujici deklarace nam umozni pristupovat ke vsem jmenum
    // z jmenneho prostoru mujprostor primo bez uvadeni jmena prostoru
  using namespace mujprostor;
  mojePromenna = 100;
  mojeFunkce();
  MojeTrida test;
  test.print();
}
```

Jmenný prostor standardní knihovny `std`

- Všechny jména používaná ve standardní knihovně C++ jsou uzavřena ve jmenném prostoru `std`
- Chceme-li používat jména standardní knihovny musíme použít jeden z následujících přístupů:
  - před každé jméno ze standardní knihovny uvést `std::`
  - pomocí deklarace `using` specifikovat jména která budeme používat
  - zpřístupnit všechna jména ve jmenném prostoru std použitím deklarace `using namespace std;`
- Některé knihovny nemají jména uzavřena ve jm. prostoru (např. Qt)

```
int value = 0;
std::cout << "Vystup na obrazovku";
std::cin >> value;

using std::cout;   // Zpristupnime jmeno cout z jmenneho prostoru std
cout << "Vystup na obrazovku";   // Zde jiz nemusime davat std::
std::cin >> value;               // Zde porad musime davat std::

using namespace std; // Zpristupnime vsechna jmena z jmen. prostoru std
cout << "Vystup na obrazovku";   // Zde nemusime davat std::
cin >> value;                    // Ani zde nemusime davat std::
```

Výjimky v C++

- V programu často nastanou chybové stavy, které vyžadují ukončení provádění kódu (např. opuštením funkce) a vhodnou reakci na vzniklou chybu (výpis chybové zprávy na obrazovku, uvolnění dynamicky alokované paměti a pod.)
- Chybové stavy často vznikají při práci se soubory (např. nelze zapisovat protože je disk plný), matematických operacích (pokus o dělení nulou) a ostatních situacích (např. předávané parametry obsahují hodnoty mimo přípustný rámec)
- Chybové stavy můžeme ošetřit pomocí běžných programátorských postupů, lepší řešení však nabízí použití tzv. výjimek jazyka C++
- Oblast kódu ve které očekáváme možnost vzniku chybového stavu uzavřeme do bloku `try {}` a uvnitř bloku v případě chyby vyvoláme vyjímku pomocí `throw parametr_chyby`
- Vzniklé výjimky zachytáváme v jednom nebo více blocích `catch`, které následují ihned za blokem `try`
- Výjimky patří mezi pokročilejší programovací techniky a jsou určeny pro zkušenější programátory
- Více na: http://www.cplusplus.com/doc/tutorial/exceptions/

Výjimky v C++ - příklad

```
int main()
{
  ifstream ifile;

  try
  {
    ifile.open("test.dat");
    if (ifile.fail())
      throw 20;

    // Nejaky dalsi kod
  }
  catch (int e)
  {
    cout << "Chyba cislo: " << e << endl;
  }
}
```

Implicitní a explicitní konverze typů

- V jazyce C++ (a také C) můžeme proměnné jednoho typu přiřadit proměnnou jiného typu, při tom dojde k tzv. konverzi typu.
- V případě, kdy konverze typu nemůže způsobit ztrátu dat, je konverze prováděná překladačem automaticky, jedná se o implicitní konverzi
- Pokud při konverzi dochází ke změně konvertované hodnoty (např. zaokrouhlení nebo ořezání) je třeba explicitně specifikovat, že chceme tuto konverzi provést, jedná se o explicitní konverzi

```
int main()
{
  int a;
  double d;

  d = a;  // Implicitnní konverze promenne typu int na double

  a = (int)d;  // Explicitni konverze double na int (pri ni dochazi
               //   k zaokrouhlovani smerem dolu)
}
```

Konverze u objektových typů

- U objektových typů používáme přetypování hlavně v souvislosti s ukazateli
- Implicitně lze přetypovat ukazatel třídy na ukazatel jeho základní třídy
- Při přetypování opačným směrem (ze základní třídy na odvozenou) musíme použít explicitní přetypování. Toto však možné udělat pouze v případě kdy ukazatel ukazuje na typ třídy na kterou přetypováváme (popř. některý z jeho předků).

```
class Circle : public Graphic  .....

int main()
{
  Circle *circle1 = new Circle();
  Circle *circle2 = 0;
  Graphic *graphic1 = 0;

  graphic1 = circle1;  // Implicitni konverze na zakladni tridu

  circle2 = (Circle*) graphic1; // Explicitni konverze ze zakladni
                                // tridy na odvozenou
}
```

Dynamická konverze typu

- Dynamická konverze typu se používá při přetypování z typu základní třídy na typ odvozené třídy, přitom však dochází k automatické kontrole přípustnosti této konverze, tj. zdali ukazatel skutečně ukazuje na objekt té třídy na niž chcem konvertovat. Pokud to není splněno, dojde při konverzi k nastavení ukazatele na 0.

```
class Circle : public Graphic  ....

int main()
{
  Circle *circle1 = 0;
  Circle *circle2 = 0;
  Graphic *graphic1 = new Circle();
  Graphic *graphic2 = new Graphic();

  circle1 = dynamic_cast<Circle*>(graphic1); // Uspesna konverze
  if (circle1 != 0)
    cout << “Konverze byla uspesna” << endl;

  circle2 = dynamic_cast<Circle*>(graphic2); //Neuspesna konverze
  if (circle2 == 0)
    cout << “Konverze nebyla uspesna” << endl;
}
```

Návrhové vzory

- Návrhové vzory (angl. design patterns) jsou soubory řešení zaměřených na některé často se vyskytující situace při vývoji programu
- Návrhové vzory využívají principů objektového programování a bývají obecně použitelné v rámci různých objektově orientovaných programovacích jazyků (tedy nikoliv pouze C++)
- V současnosti je používáno 24 základních návrhových vzorů, byly však navrženy i další
- Další informace:
  http://en.wikipedia.org/wiki/Software_design_pattern
  http://www.oodesign.com/

Návrhový vzor Singleton

- Vzor Singleton (Jedináček) se používá při návrhu tříd, které mají vytvářet pouze jednu instanci (typicky např. třída `Application`)

```
class Singleton
{
    public:
     static Singleton getInstance()
      {
          if (instance == 0)
              instance = new Singleton();

          return instance;
      }

    private:
      static Singleton *instance;
      Singleton()  { instance = 0; }
};

int main()
{
  Singleton *singl = Singleton::getInstance();
}
```