

**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 deklaruujeme 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 jmenné 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 definována promenna, funkce a trida jako na predchozi
    // strane
}

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
- Použití `using namespace` se obecně vyhýbáme, je vhodné jen ve velmi omezených případech. Zejména ve spojení s jmennými prostory knihoven vede k nebezpečným kolizím jmen.

```
namespace mujprostor
```

```
{  
    // Zde bude definována 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;` (*raději nepoužívejte*)
- Některé knihovny nemají jména uzavřena ve jm. prostoru (např. Qt)

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

using std::cout; // Zprístupnime 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; // Zprístupnime 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. opuštěním 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 rozsah)
- 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 v něm v případě chyby vyvoláme výjimku pomocí **throw nejaky\_objekt\_popisujici\_chybu**
- Vzniklé výjimky zachytáváme v jednom nebo více blocích **catch**, které následují ihned za blokem **try**
- Nezachycené výjimky „probublávají“ do volajících funkcí. Nejsou-li zachyceny ani v **main()**, je program ukončen s chybou.
- Více na: <http://www.cplusplus.com/doc/tutorial/exceptions/>

# Výjimky v C++ - příklad

```
#include <stdexcept>

void funkce()
{
    ifstream ifile;

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

        // Nejaký další kód
        if (necoSeNepovedlo())
            throw std::runtime_error("Neco se nepovedlo!");
    }
    catch (int e) // Zachyti jakoukoli vyjimku typu int
    {
        cout << "Chyba cislo: " << e << endl;
    }
    // Vyjimka typu std::runtime_error bude predana volajici funkci
}
```



# 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; // Implicitní konverze promenne typu int na double

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

# Explicitní přetypování v C++

- Klasické přetypování ve stylu jazyka C (pomocí názvu typu v závorkách) má mnohá úskalí, zejména se těžko hledá v kódu
- Jazyk C++ nabízí lepší alternativu pomocí klíčového slova **static\_cast**, z němž se uvede cílový typ ve špičatých závorkách, jako by se jednalo o šablonu
- `static_cast` zakazuje provádět nebezpečné konverze (například s ním nelze odstranit modifikátor `const`)

```
void funkce(const string &s)
{
    int a;
    double d = 1.5;

    a = static_cast<int>(d);

    // Toto se ale nezkompiluje!
    string &x = static_cast<string &>(s);
    x = "Ha!";
}
```

# Konverze u objektových typů

- U objektových typů používáme přetypování hlavně v souvislosti s ukazateli a referencemi
- Implicitně lze přetypovat ukazatel/referenci třídy na ukazatel/referenci jeho základní třídy
- Při přetypování opačným směrem (ze základní třídy na odvozenou) musíme použít explicitní přetypování. Toto však smíme udělat pouze v případě, kdy ukazatel ukazuje na instanci třídy, na kterou přetypováváme (popř. na některého z jejích potomků).

```
class Circle : public Shape ...

int main()
{
    Circle circle1;

    // Implicitni konverze na zakladni tridu
    Shape &shape1 = circle1;

    // Explicitni konverze ze zakladni tridy na odvozenou
    // Vime, ze shape1 ukazuje ve skutecnosti na objekt typu Circle
    Circle &circle2 = static_cast<Circle &>(shape1);
}
```

# Dynamická konverze typu

- Dynamická konverze typu se používá při přetypování z typu základní třídy na typ odvozené třídy, přitom však dochází ke kontrole přípustnosti této konverze, tj. zdali zdrojový ukazatel skutečně ukazuje na objekt té třídy, na niž chceme konvertovat. Pokud to není splněno, `dynamic_cast` vrátí `nullptr`.

```
class Circle : public Shape ...

int main()
{
    Circle *circle1 = nullptr;
    Circle *circle2 = nullptr;
    Shape *shape1 = new Circle();
    Shape *shape2 = new Shape();

    circle1 = dynamic_cast<Circle *>(shape1); // Uspesna konverze
    if (circle1 != nullptr)
        cout << "Konverze byla uspesna" << endl;

    circle2 = dynamic_cast<Circle*>(shape2); // Neuspesna konverze
    if (circle2 == nullptr)
        cout << "Konverze nebyla uspesna" << endl;
}
```

# Výchozí a smazané konstruktory

- Nechceme-li třídě definovat vlastní konstruktor, můžeme ovlivňovat generování konstruktorů kompilátorem pomocí klíčových slov `default` a `delete`
- Nastavení konstruktoru na default přinutí kompilátor vygenerovat takový konstruktor, jako by třída neměla žádné uživatelsky definované konstruktory (více méně jako konstruktor s prázdným tělem)
- Použití `delete` zakáže automatické generování konstruktoru

```
class A // Tuto tridu nepujde vytvorit bez argumentu
{
public:
    A() = delete;
    A(int x) : val(x) {}
private:
    int val;
}
```

```
class B // Tuto tridu lze vytvorit bez argumentu i s nim
{
    B() = default;
    B(int x) : val(x) {}
private:
    int val = 0;
}
```

# Návrhové vzory

- Návrhové vzory (angl. *design patterns*) jsou soubory řešení zaměřených na některé často se vyskytující situace při vývoji programů
- Návrhové vzory využívají principů objektového programování a bývají obecně použitelné v rámci různých objektově orientovaných programovacích jazyků (tedy nikoliv pouze C++)
- Základní návrhové vzory vycházejí z knihy Design Patterns, kterou sepsali E. Gamma, R. Helm, R. Johnson a J. Vlissides (často označovaní jako „Gang of Four“ nebo GoF)
- Další informace:  
[https://en.wikipedia.org/wiki/Software\\_design\\_pattern](https://en.wikipedia.org/wiki/Software_design_pattern)

# Návrhový vzor Singleton

- Vzor Singleton (Jedináček) se používá při návrhu tříd, které mají vytvářet pouze jednu globální instanci (např. logovací třída či generátor náhodných čísel)

```
class Singleton
{
public:
    static Singleton &getInstance()
    {
        if (instance == nullptr) {
            instance = std::unique_ptr<Singleton>(new Singleton());
        }
        return *instance;
    }

private:
    Singleton() = default;
    static std::unique_ptr<Singleton> instance;
};

void funkce1()
{
    Singleton &singl = Singleton::getInstance();
}
```