

KIV/TSI - Seminář C++

01. Úvod do moderního C++

Martin Úbl

KIV ZČU

2020/2021

- multiparadigmatický kompilovaný jazyk
 - procedurální
 - objektově orientovaný
 - generické programování
 - prvky funkcionálního programování
- Bjarne Stroustrup et al.
- standardy:
 - C++98, C++03
 - C++11, C++14, C++17
 - C++20?

- bezpečnost
 - práce s pamětí
 - typová bezpečnost
 - práce se zdroji
- čitelnost
- výkon
- efektivní vývoj
 - objektové paradigma
 - generické programování
- standardní knihovna (`libc` + STL)

- deterministická životnost objektu
- staticky typovaný
- šablony + statický polymorfismus

- alokace paměti
 - statická
 - dynamická - `new` a `delete` (vs. `malloc` a `free` v C)
- snaha upřednostnit statickou alokaci
 - resp. kontejnery které interně alokují dynamicky
 - správa paměti, zdrojů – RAII
- podpora compile-time vyhodnocení (`constexpr`)
- copy a move sémantika
- idiomy
 - RAII, Pimpl, CRTP, SFINAE, scope guard
 - a další... <https://en.cppreference.com/w/cpp/language>
- výjimky
- jmenné prostory

- Hello world!

```
#include <iostream>
```

```
int main(int argc, char** argv)
```

```
{
```

```
    std::cout << "Hello_world" << std::endl;
```

```
    return 0;
```

```
}
```

- Hello world!
- using namespace - může být špatná praktika (konflikty jmen)

```
#include <iostream>

using namespace std; // ugh!

int main(int argc, char** argv)
{
    cout << "Hello_world" << endl;
    return 0;
}
```

- datové typy stejné jako v C
 - zde spadají do kategorie POD (Plain Old Data)
- navíc např.:
 - třídy (`class`)
 - silně typované výčty (`enum class`)

- třída
 - klíčové slovo `class`
 - sekce viditelnosti (`public`, `private`, `protected`)
 - konstruktory
 - inicializační seznamy
 - destruktory
 - virtuální metody (polymorfismus)
 - oddělení definice od implementace
- zjednodušený příklad na dalších slajdech
 - zatím bez některých do budoucna podstatných věcí
 - např. `const` a reference, virtuální destruktory, viz dále

```
class Player { // Player.h
public:
    Player();
    Player(std::string name);
    ~Player();

    virtual double Get_X();
    virtual double Get_Y();
    std::string Get_Name();

protected:
    std::string mName;

private:
    double mX, mY;
};
```

```
#include "Player.h" // Player.cpp

Player::Player() : mName("Unknown_player") {
    // initialize
}

Player::Player(std::string name) : mName(name) {
    // initialize
}

Player::~Player() {
    // deinitialize
}

double Player::Get_X() {
    return mX;
}

...
```

- lze psát implementaci do deklarace třídy
- virtuální destruktork

```
class Player {                                     // Player.h
public:
    Player() : mName("Unknown_player") {
        //
    }
    virtual ~Player() {
        // zajisti spravne volani destruktorku
        // v polymorfnim schematu
    }
    ...
};
```

- reference
 - „bezpečný ukazatel“
 - vždy odkazuje na platný objekt/POD
 - vždy inicializována
 - nelze „převázat“ na jiný objekt/POD
 - symbol & (víceznačnost!)
 - nemůže být „null“
 - lze vázat na existující referenci

```
int a = 1;
int &b = a;
int &c = b;
int *d = &a; // mozne, ale neni bezpecne
```

```
b = 5;           // a == 5
c = 10;         // a == 10
*d = 15;        // a == 15
```

- statická alokace
- dynamická alokace
- zapouzdřená dynamická alokace

```
Player player("Adam");
```

```
Player* player = new Player("Adam");
```

```
std::unique_ptr<Player> player  
    = std::make_unique<Player>("Adam");
```

- scope
- vymezuje platnost jména (a objektu)

```
void someFunction()  
{  
    Player adam("Adam"); // (1)  
    Player* josef = new Player("Josef"); // (2)  
    ...  
} // (3)
```

- (1) volá konstruktor Player (adam)
- (2) volá konstruktor Player (josef)
- (3) volá destruktork Player (adam),
NEvolá destruktork Player (josef)

- scope
- vymezuje platnost jména (a objektu)

```
void someOtherFunction()  
{  
    Player* josef = new Player("Josef"); // (1)  
    ...  
    delete josef; // (2)  
}
```

- (1) volá konstruktor Player (josef)
- (2) volá destruktory Player (josef)
- náchylné na chyby
 - náchylné na chyby
 - pokud možno, alokovat staticky
 - nebo chytré ukazatele (další semináře)

- dědičnost
- „viditelnost“ dědičnosti

```
class Object {
public:
    Object(std::string name) {
        mName = name;
    }
    ...
};

class Player : public Object {
public:
    Player() : Object("Unknown_player") {
    }
    Player(std::string name) : Object(name) {
    }
};
```

- dědičnost a virtuální metody
- klíčové slovo override

```
class Base {  
    public:  
        virtual void Hello();  
        void Goodbye();  
};
```

```
class Derived : public Base {  
    public:  
        virtual void Hello() override;  
        void Goodbye(); // (1)  
};
```

- (1) override by způsobilo chybu při překladu – Base::Goodbye není virtuální, nelze ji tedy přepsat, pouze překrýt

- klíčové slovo auto
- automatická dedukce typu
- vždy musí být z čeho dedukovat
 - inicializace proměnné
 - návratová hodnota funkce

```
auto i = 15; // (1)
auto p = new Player(); // (2)
auto itr = playerMap.find(key); // (3)
auto ptr = std::make_unique<Player>(); // (4)
```

- (1) dedukce int
- (2) dedukce Player*
- (3) dedukce std::map<int, Player*>::iterator
- (4) dedukce std::unique_ptr<Player>

- STL kontejnery
- `std::array`
- generický typ (šablony), reprezentace pole pevné délky

```
#include <array>
```

```
std::array<int, 5> petCisel  
    = { 1, 2, 3, 4, 5 };
```

```
std::cout << petCisel[1]; // "2"
```

- analogie ve stylu C:

```
int petCisel[5] = { 1, 2, 3, 4, 5 };
```

- STL kontejnery
- `std::vector`
- generický typ (šablony), reprezentace pole proměnné délky

```
#include <vector>
```

```
std::vector<int> nekolikCisel  
    = { 1, 2, 3 };
```

```
 nekolikCisel.push_back(15);
```

```
std::cout << nekolikCisel[3]; // "15"
```

- další STL kontejnery viz další semináře
- `std::list` - spojový seznam
- `std::map` - asociativní úložiště
- `std::set` - množina
- `std::queue`, `deque` - fronta, oboustranná fronta
- a mnoho dalších...

- kompilátory
 - gcc/g++ (GNU)
 - clang/clang++ (LLVM)
 - icc (Intel)
 - MSVS cl (Microsoft)
 - a další...
- podpora standardů C++11 a vyšší
- vývojová prostředí
 - Visual Studio (Microsoft)
 - Dev-Cpp (Bloodshed)
 - Xcode (Apple)
 - CLion (JetBrains)
 - a další...

- „Zajeté“ standardy psaní kódu
- např. C++ Core Guidelines
 - <http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>
 - dobré procházet průběžně

- seznámení s prostředím
 - Microsoft Visual Studio
- jednoduchý „Hello world“ a ukázky práce s třídami a kontejnery