KIV/CPP – Programování v jazyce C++

01. Úvod do moderního C++

Martin Úbl

KIV ZČU

2023/2024

Jazyk C++

- 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, C++23

Jazyk C++

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

Jazyk C++

- 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`, `consteval`, `constinit`)
- 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
  - destruktor
  - virtuální metody (polymorfismus)
  - oddělení definice od implementace
- zjednodušený příklad na dalších slajdech
  - zatím bez některých do budoucna podstatných věcí
  - např. `const` a reference, virtuální destruktor, 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 destruktor

```
class Player {                                  // Player.h
  public:
    Player() : mName("Unknown␣player") {
      //
    }
    virtual ~Player() {
      // zajisti spravne volani destruktoru
      // 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 destruktor Player (`adam`),
  NEvolá destruktor 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 destruktor Player (`josef`)
- náchylné na chyby
  - náchylné na chyby
  - pokud možno, alokovat staticky
  - nebo chytré ukazatele (další přednášky)

- explicitní scope
- ohraničena složenými závorkami
- ve spojení s nějakou řídicí strukturou nebo deklarací (cyklus, funkce, ...)

```
void someFunction()
{
  //
}
```

- nebo samostatná

```
// nejaky vnejsi kod

{
  //
}
```

- obě fungují stejně (platnost identifikátoru i životnost objektu)

- 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 virtualní, 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
- vyšší abstrakce, než pole (např. v C)
- standard definuje např. výpočetní a paměťové složitosti
- konkrétní implementaci zpravidla nepředepisuje
  - např. požadavkem je sekvenční přístup ke konstantnímu počtu prvků v konstantním čase
  - programátor standardní knihovny pak implementuje pomocí statického pole

- 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 v další přednášce
- `std::list` – seznam
- `std::map` – asociativní úložiště
- `std::set` – množina
- `std::queue, deque` – fronta, oboustranná fronta
- a mnoho dalších...

- hlavičkové soubory
  - z STL bez přípony
  - jinak často přípona .hpp nebo .h
  - preprocesoru to je jedno
- C++ varianta C hlavičkových souborů standardní knihovny
  - aby se zamezilo konfliktům, jsou definovány wrappery
  - z názvu souboru se odebere přípona a připojí se znak „c“ na začátek
  - např. `math.h` se změní na `cmath`
  - funkce je pak vhodné prefixovat `std::`
    - např. `std::max`, `std::pow`, ...

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

Coding standards

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

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