NPRG 041 – cvičení 6
Programování v C++

Jiří Klepl

mail:
klepl@d3s.mff.cuni.cz

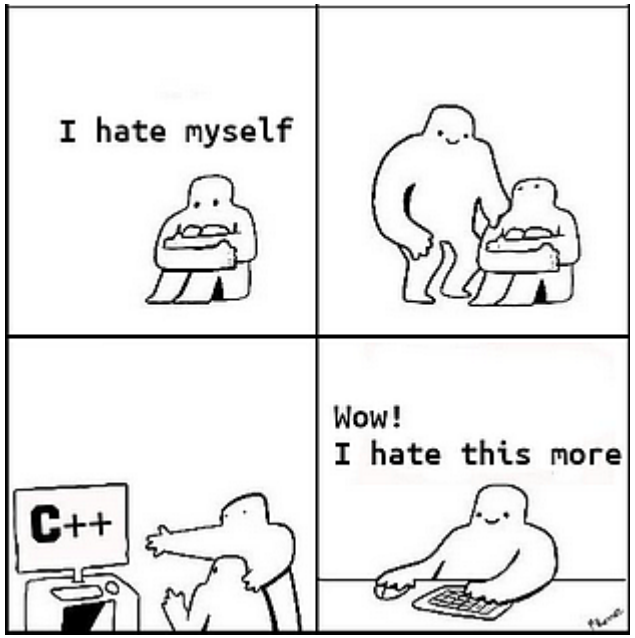
mattermost:
https://ulita.ms.mff.cuni.cz/mattermost/ar2324zs/channels/nprg041-cpp-klepl

Feedback k úloze Práce s algoritmy

TODO

Agenda

- **Práce s chybami v programu**
- **Pátá úloha: CMake projekt s externími závislostmi**
  - **Jednoduchá simulace zápočťáku**
  - **Ukázka práce s 2D grafikou**
  - **Ukázka práce s networkingem**
  - **Ukázka práce s multiprocessingem**
  - **Testování**



Předcházení chybám, řešení chyb

Asserty

```
double sq_root(double from) {
    assert(from >= 0);
    return sqrt(from);
}
```

V runtime kódu: **assert(condition)**

Pro konstanty: **static_assert(condition, "message")**

- **assert zabije program** a ohlásí chybu, **static_assert zabije přímo kompilaci**
- Pokud v kódu očekáváme, že něco platí – **invarianty**
  - Např. vztahy mezi konstantama, invarianty v loopě, preconditions funkcí
- Vždy, když funkce očekává nějaké preconditions, nebo zajišťuje nesamozřejmé postconditions, měli bychom to označit assertem
  - Zde např. isocpp doporučuje **Expects(cond)** a **Ensures(cond)**
- **Proč?**
  - Protože pokud invariant porušíme a nemáme assert, je těžké najít bug
  - Navíc dokumentují kód –pro programátora i compiler (a ten je zkontroluje)
- Jejich cena? 0 – **počítají se jen v debug verzi**

[[Atributy]]

- C++ nabízí několik standardních atributů (a compilery mnoho dalších)
  - Většina z nich nemění chování programu; typicky jen optimalizace a warningy
- My se teď zaměříme na ty, které zlepšují readabilitu a pomáhají předcházet chybám:
  - **[[nodiscard(“good reason”)]]** – označí, že výsledek funkce nemá být zahozen
    - To může významně pomoct při programování – špatné použití funkce (výsledek nikam nepřiřadíme) samo poradí, proč to je špatně
  - **[[noreturn]]** – funkce nevrací kontrolu zpět (nekonečná smyčka, throw, atd.)
    - **Ale pozor!** Tímhle atributem nesmíme lhát -> když jej použijeme, return je UB

```
[[nodiscard]] int computeSquare(int number) {
    return number * number;
}
```

```
[[noreturn]] void fatalError(const string& msg)
{
    cerr << "Fatal error: " << msg << endl;
    exit(EXIT_FAILURE); // exit the program
}
```

Výjimky

- **try { … throw sth_bad("murder"); … } catch(const sth_bad& err) { … }**
  - catch reaguje na každý throw stejného typu kdekoli ve scopu try
    - Samozřejmě i v zanořených voláních
  - Pokud catch nenastává, je to levnější než ekvivalentní kód s if statementy
  - Výjimky typicky tvoří class hierarchy, **catch(const std::exception&)** chytí jakoukoli správně definovanou výjimku, **catch(…)** chytí cokoliv (fakt …)
- V C++, prakticky každá funkce (mimo destruktory) je „potentially throwing“ – ne jako v Javě; ale existuje **noexcept** (= doesn‘t throw)
- Teoreticky jde házet cokoli – ale good design je házet jen:
  class specifically_named_error : public exception {
  const char *what() const; // v catch(err): std::cerr << err.what() << endl;
  };

Co se stane při throw

- **throw funguje skoro jako return** (kdyby try bylo volání funkce a catch bylo místo, kam se vrátit - nic po throw se neprovede)
- **RAII** – všechny objekty definované **mezi try** (který definuje aktivovaný catch) **a throw** jsou zahozeny (jako při řádném ukončení scopu)
  - **A a; try { B b; { C c; throw we_failed("horribly"); D d; } } catch(const we_failed& err) { std::cerr << err.what() << std::endl; } //prints: horribly**
  - Po **throw** a před samotnou aktivací catch: **úklid (v pořadí) c, b**
    - Objekt **d nikdy neexistoval;** pokud je něco stále špatně, tak **c, b** a **catch** to mají napravit
  - To samé platí samozřejmě pro **return** – ten ukončí všechny scopy ve funkci
- **C++ nemá finally { … }, ale díky RAII ho nepotřebuje**
  - **Destruktory totiž jsou finally!**

Exception safety při volání funkce

- Nejideálnější je **Nothrow exception guarantee** – pokud ve funkci **nikdy nikdy nikdy nenastane exception**, nebo se o její vyřízení stará, můžeme ji označit **noexcept**
  - **Ale pozor!** Pokud by měla hodit výjimku, tak nastane **std::terminate**
    - **std::terminate udělá SIGABRT – zabije program**
- Co chceme zachovávat: **Strong exception guarantee**
  - **Pokud vyhodíme výjimku**, zajistíme, že **program vypadá jako před zavoláním**
  - Většina funkcí tohle zachovává – pokud vector.push_back skončí výjimkou, tak se počet prvků ve vektoru nezmění (a nenastane realokace)
  - **Destruktory jsou velká výpomoc**, volají se i když nastane výjimka
    - Prostě podobně jako u finally bloků v jiných jazycích
- Pokud nedokážeme strong, tak alespoň **Basic exception guarantee**
  - Něco se pokazilo, ale logika programu stále drží, nic se nerozbilo, nejsou leaky
  - Třeba jsme jen ztratili nějaký prvek user-defined kontejneru (ale byl řádně uklizen)

Assertions vs if statement vs Exceptions

1. Je to naprosto normální chování programu -> **if statement**
   - Corner-casy algoritmu (e.g.: hranice polí, pokud saháme na sousední prvky)
   - Často nastávající a snadno vyřešitelné úkazy
2. Může být rozbitá logika programu a chceme to odhalit (v Debugu)
   - Chybně nastavené konstanty v kódu: **static_assert**
   - Chyba v runtimu: **assert**
     - Nedodržené invarianty v algoritmu, precoditions, postconditions
     - Pointer, který by neměl být nullptr, je nullptr
3. Chyba, který se neubráníme (jde odjinud) -> **exception**
   - Musí to být výjimečný stav (jinak se více vyplatí if); řešit lze až daleko od místa, kde se projevuje… try catch dokáže chytit výjimku vyhozenou v zanořených voláních

Build systémy: CMake, vcpkg

CMake – základy

1. Soubor **CMakeLists.txt**
   - První direktiva je obrana, aby uživatelův CMake podporoval vše potřebné
   - Druhý definuje projekt
   - Pak typicky nasetujeme standardy a optiony
   - Includy, add_executable, add_library, …
2. Program **cmake** nakonfiguruje a sestaví projekt

```
cmake_minimum_required(VERSION 3.20)

project("Simple Application"
    VERSION 0.1
    DESCRIPTION "Demonstration of CMake"
    LANGUAGES CXX
)

# Set the C++ standard to C++23 (globally)
set(CMAKE_CXX_STANDARD 23) # suggest using C++23
set(CMAKE_CXX_STANDARD_REQUIRED True) # Enforce C++23
set(CMAKE_CXX_EXTENSIONS OFF) # Disable extensions

# Add include directory (globally)
include_directories(include)

# Add executable
add_executable(application src/main.cpp
                           features/cool_feature.cpp)
```

CMake – základní použití

- **Začínáme ve složce se CMakeLists.txt** a vytvoříme **podsložku build**
- Vejdeme do složky build a spustíme konfiguraci:
  cmake -DCMAKE_BUILD_TYPE=<type> .. („..“ je cesta k projektu)
  - Build typ typicky: **Debug**, **Release**, **RelWithDebInfo** (tenhle např. na profiling)
- Potom konečně sestavení: **cmake --build .**
  - Useful optiony:
    - **-j [<jobs>]** – multithreaded kompilace
    - **--clean-first** – vyčištění cílové složky
    - **-t <target>** – sestavení pouze zadaného cíle
- Pokud chceme testovat (musíme definovat v cmake filu), tak: **ctest**

CMake – základní direktivy pro definici targetu

- **add_executable(<name> <sources>…)**
  - Vytvoří spustitelný soubor
- **add_library(<name> [STATIC | SHARED | MODULE] <sources>…)**
  - „Normální“ knihovna – staticky linkovaná, dynamicky, dynamicky na vyžádání
- **add_library(<name> INTERFACE <sources>…)**
  - „Interface“ knihovna – např. když je „header-only“, negeneruje žádné objekty
    - To se nám hodí např. na definici generických std-like knihoven – ty je potřeba generovat podle zadaných typů, nejde je předvyrobit
- **add_custom_target(<name> <command with args>)**
  - Typicky použijeme, pokud target vygeneruje nějaký script

CMake – základní konfigurace targetů

- Pro většinu těchto direktiv máme globální a target-specific verzi
  - Často se hodí každý target nakonfigurovat jinak; jindy chceme všechny nakonfigurovat stejně
  - **set** a **set_target_properties, include_directories** a **target_include_directories**
- Zde seznam těch hlavních (target-specific verze):
  - **target_include_directories** – kde má target hledat header fily
  - **target_link_libraries** – co chceme k targetu přilinkovat za knihovnu (přidá i potřebné includy)
  - **set_target_properties** – tím můžeme nastavit standard atd (viz předchozí slajd)
  - **target_compile_options** – zde můžeme nastavovat warningy atd. (*compiler-specific*)
  - **target_compile_definitions** – nastavení definů: MY_FLAG WITH_VALUE=1
    - Přeloží se na compilerem podporované optiony, pro gcc: -DMY_FLAG -DWITH_VALUE=1
- JE potřeba znát rozdíl mezi PRIVATE a PUBLIC: PRIVATE konfigurace se nepředává dál (třeba kdybychom dělali library a měli bychom header file s definicí něčeho, co nemá být v API: **target_include_directories(target PRIVATE header.hpp)**

CMake – podprojekty a základní proměnné

- Podprojekty: **add_subdirectory(<subdir> [<output>])**
  - Ve složce <subdir> máme další CMakeLists.txt s podprojektem,
    <output>: optional argument, který určuje cílovou podsložku, jinak <subdir>
- Základní proměnné (když je setujememe, tak bez ${…}):
  - **${CMAKE_SOURCE_DIR}** – cesta k projektu
  - **${CMAKE_BINARY_DIR}** – cesta k cílové složce
  - Spousta **${CMAKE_<LANG>_...}** proměnných – nastavujeme tím standard, optiony, atd.
    - Tyhle typicky nastavujeme: např. **set(CMAKE_CXX_STANDARD 23)**
    - **LANG**: nejčastěji (pro nás) **CXX**, může být i např. **C** nebo **CUDA** (programování pro GPU)

CMake – externí projekty

- **find_package(<package> [REQUIRED])**
  - + spousta optional argumentů: často doporučeny podle externího projektu
  - Přidá externí dependenci
    Na to většinou navážeme: **target_link_libraries(target PRIVATE package::…)**
  - Ten **package musí být nainstalován** (ukážeme si přenositelně s **vcpkg**, ale třeba na Ubuntu/Debian stačí nainstalovat přes apt)
- Pro malé/specifické projekty můžeme použít (místo find_package):

  Include(FetchContent)
  FetchContent_Declare(
  <package_name>
  GIT_REPOSITORY <url, např.: https://github.com/...>
  GIT_TAG <git branch, nebo tag name (release)>
)
  FetchContent_MakeAvailable(<package_name>)
  - To package stáhne do cílové složky jako dependenci
  - Typicky ho přímo na místě i sestaví (zatímco u find_package už je nainstalován)
  - Bývá to mnohem jednodušší na použití, pokud ten package nevyžaduje něco nainstalovat

vcpkg

- https://vcpkg.io/en/getting-started https://vcpkg.io/en/packages
- Dovoluje nám jednoduše instalovat package mimo systém (není potřeba např. sudo, nerozbijeme nic v systému)
  - Ale občas je potřeba něco doinstalovat – pak poradí přesně jak
- Spolupracuje s nástrojem Cmake **(ujistěte se, že vcpkg je v PATH)**:
  - CMake musíme při konfiguraci říct, kde hledat package (přidáme option): **-DCMAKE_TOOLCHAIN_FILE=[vcpkgroot]/scripts/buildsystems/vcpkg.cmake**
  - Pak můžeme pohodlně používat find_package(…) s packagi ve vcpkg
  - **vcpkg install <package>** nám poradí, co asipřidat do CMakeLists.txt
    - I po instalaci to můžeme použít na ten hint

Testování

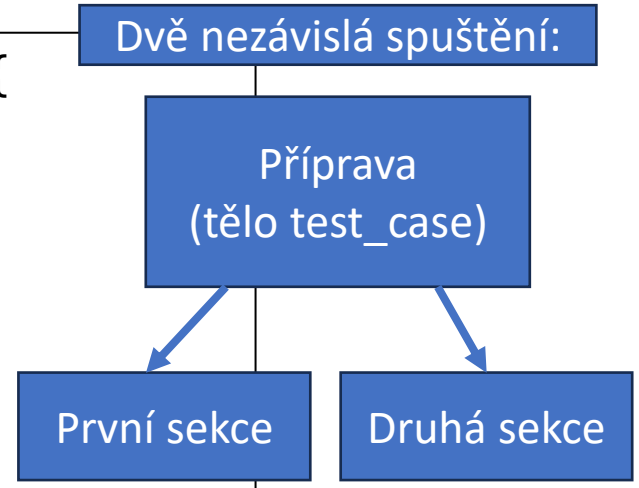
- Existuje spousta knihoven, které lze doporučit
  - Boost.Test, Catch2, Google test, mnoho dalších…
- My se zaměříme na Catch2 (čistě osobní volba, testy vyberte podle potřeb nebo metody vývoje)
  - Podporuje unit testing i test-driven development, umí benchmarky, …
  - V různých .cpp definujeme bloky označené **TEST_CASE(<name>, <tags>)**
    - To definuje jeden konkrétní test (různé testy jedné featury definujeme v jednom .cpp)
  - V tom testu pak použijeme **REQUIRE(condition_expression)**
    - Jako assert, ale ukončí pouze ten daný test case
    - Dokáží rozebrat zadaný expression: REQUIRE(2 == 1) řekne přímo „2 není 1“
  - Test case můžeme rozdělit na různé **SECTION(name)** bloky
    - Každý se interpretuje zvlášť – když na jedné věci chceme testovat 2 situace
      - Každý section jde větvit na další subsekce – nemusíme psát mnoho drobných testů

Catch2 example

```
TEST_CASE( "vectors can be sized and resized", "[vector]" ) {
    std::vector<int> v( 5 );

    REQUIRE( v.size() == 5 );
    REQUIRE( v.capacity() >= 5 );

    SECTION( "resizing bigger changes size and capacity" ) {
        v.resize( 10 );
        REQUIRE( v.size() == 10 );
        REQUIRE( v.capacity() >= 10 );
    }
    SECTION( "resizing smaller changes size but not capacity" ) {
        v.resize( 0 );
        REQUIRE( v.size() == 0 );
        REQUIRE( v.capacity() >= 5 );
    }
}
```



Pátá úloha:
CMake projekt s externími dependencemi

- Sestavíme program s 2D grafikou a networkingem, napíšeme k němu testy
- Použité externí dependence:
  - **sfml**: ./vcpkg install sfml
  - **boost-asio**: ./vcpkg install boost-asio
  - **catch2**: ./vcpkg install catch2
- https://gitlab.mff.cuni.cz/teaching/nprg041/klepl/cmake-project
  - Clone it in the **/labs** folder and **delete .git** (or register it as a git submodule)
- TODO jsou v README.md
  - Většina projektu už je předpřipravená