NPRG 041 – cvičení Programování v C++

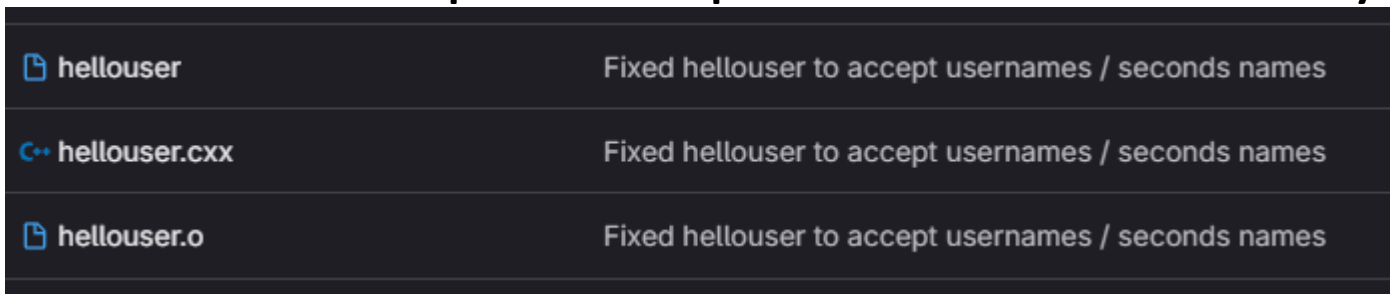
Jiří Klepl

mail:
klepl@d3s.mff.cuni.cz

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

Problémy s Hello world – feedback

- Někteří z vás do repozitáře pushli binární soubory:

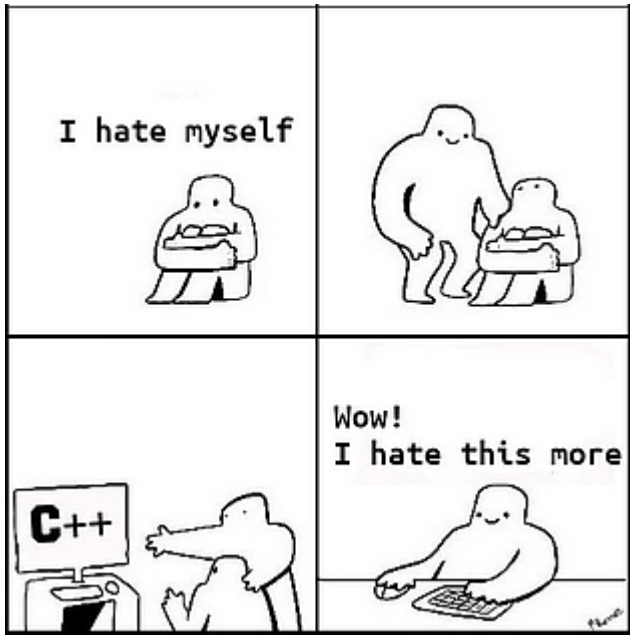


  - Řešení: např. vytvoření .gitignore, kde na prvním řádku bude *.o a na druhém hellouser
    - Souborů .gitignore může být víc, každý platí ve své složce a všech uvnitř

Prostor na otázky z přednášky

Agenda

- **První úloha: Násobilka**
  - Stav programu patří do třídy, ne do globální proměnné
    - Dělení na komponenty
  - Referenční vs hodnotová sémantika, předávání parametrů
  - for loop s iterátory, range-based for
  - std::array, std::vector a std::span (a jejich jednotlivé výhody)



Násobilka

- Vycházejte z kódu: https://www.ksi.mff.cuni.cz/teaching/nprg041-klepl-web/data/sources/nasobilka.cpp
- **Zadání**: Vypište malou násobilku z čísel na příkazové řádce (cmd args)
  - Meziúkol: vypište parametry příkazové řádky, pak pomůže std::stoi
  - `nasobilka 5`

```
1 * 5 = 5
2 * 5 = 10
…
10 * 5 = 50
```

- **Rozšíření**: přidejte optiony (můžete předpokládat, že budou před skutečnými argumenty)
  - -f N ≈ start counting from N (default 1)
  - -t M ≈ count to M (default 10)
  - ! Oba optiony jsou nepovinné
  - ! Jejich vzájemné pořadí je libovolné
  - `nasobilka –f 3 -t 5 7 9`

```
3 * 7 = 21
4 * 7 = 28
5 * 7 = 35
3 * 9 = 27
4 * 9 = 36
5 * 9 = 45
```

Doprovodné slajdy ke kódu

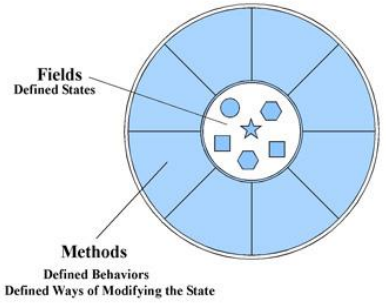
Jak řešit program state

- If there is some program state shared by many functions,
  -> **encapsulate the state in a class** (and **avoid global variables**!)
- Why?
  - Classes allow **access control** -> **private**/**protected** data fields and methods
    - When? – If there is some **invariant on the data**
      - For methods, if it is just a helper/inner detail
    - Create sensible setters that **preserve the invariants**
  - The code is **safer** and **easier to understand** <- the methods do not access some random variable

Jak řešit program state

- If there is some program state shared by many functions,
  -> **encapsulate the state in a class** (and **avoid global variables**!)
- Why?
  - Classes allow **access control** -> **private/protected** data fields and methods
    - When? – If there is some **invariant on the data**
      - For methods, if it is just a helper/inner detail
    - Create sensible setters that **preserve the invariants**
  - The code is **safer** and **easier to understand** <- the methods do not access some random variable

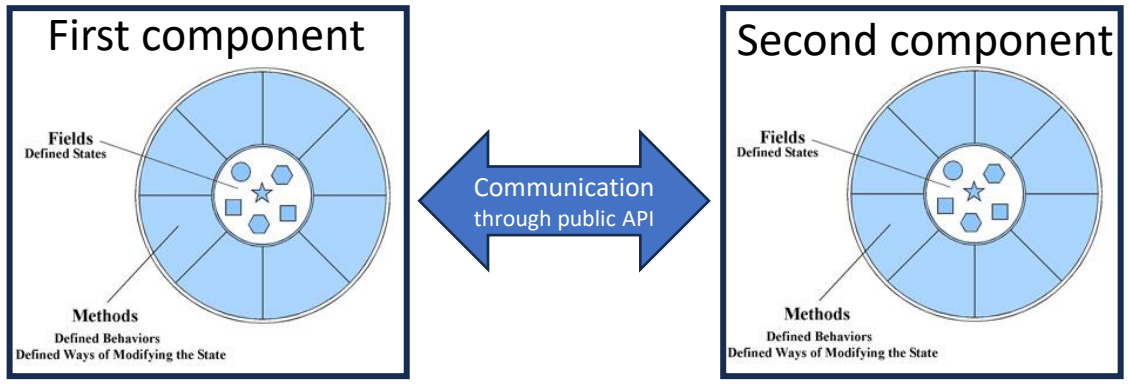
Simple program with state:



Jak řešit program state

- If there is some program state shared by many functions,
  -> **encapsulate the state in a class** (and **avoid global variables**!)
- Why?
  - Classes allow **access control** -> **private/protected** data fields and methods
    - When? – If there is some **invariant on the data**
      - For methods, if it is just a helper/inner detail
    - Create sensible setters that **preserve the invariants**
  - The code is **safer** and **easier to understand** <- the methods do not access some random variable

More complicated program:



Hodnotová vs referenční sémantika

- V C#/Javě: primitivní typy a structy = hodnotová, classy = referenční
- V C++: **každý objekt má hodnotovou sémantiku**
  - Co to znamená? – přiřazení `T b = a` **vždy okopíruje** objekt a do b
  - Pokud nechceme kopírovat, musíme explicitně brát referenci: **(const) T&**
    - **Bohužel, v místě volání to nepoznáme – zde pomáhají chytrá IDE/intellisense** 
  - "Ale pointery mají referenční sémantiku" – pointer je jen adresa a my ji hodnotově kopírujeme – *pointer jen reprezentuje referenci*
- **Některé objekty reprezentují referenci** (předání nekopíruje data):
  - Pointery a iterátory, std::span, std::string_view <- vše toto něco „observuje"
    - Iterátor = pohled na prvek kontejneru, span = pohled na array, string_view = string view
  - **Data jsou někde jinde** (typicky v jiném objektu)
  - Když před takový objekt dáme `const`, tak se neaplikuje na data 

Předávání parametrů v C++ a kdy použít jaký druh

- void foo(**Object object**) - **Používat pro triviální objekt (např. číslo)**
  - Funkce, která bere kopii objectu; volání foo(object) začne kopírováním parametru object
  - Pokud zapíšeme do objectu, neuvidíme změnu na předávaném parametru
  - object je z pohledu funkce lokální proměnná
- void foo(**Object& object**) - **Pro objekt, který chceme měnit**
  - Funkce, která bere referenci na object; při volání se nebude kopírovat
  - Po volání foo(object) bude stav objectu odpovídat tomu, jak vypadal ve foo
- void foo(**const Object& object**) - **Vždy, když to jde (u netriviální věci)**
  - To samé, ale do objectu foo nechce zapisovat (měnit jeho stav)

Časté chyby předávání a jak je opravit

- void foo(std::string text)
  - Okopíruje text, pokud budeme pracovat s textem často, tak za to docela dost zaplatíme
- void foo(std::vector<Object> objects)
  - Okopíruje všechny objekty
  - Pokud by měla ty objekty nějak měnit, tak "venku" jejich změnu neuvidíme
- void foo(Object& object) [foo nemění stav objectu]
  - Pokud foo nemodifikuje object, tak její API neodpovídá realitě, protože z její deklarace vidíme, že jej měnit může
- void foo(const int &number)
  - Zde je úplně zbytečné brát referenci (pointer na číslo není jednodušší na předání)

Časté chyby předávání a **jak je opravit**

- void foo(const std::string& text)
  - Neokopíruje text, předá se jen reference
  - Možná alternativa: void foo(std::string_view text)
    - O tom se budeme bavit později, ale je to pohled (chytrá reference) na nějaký string
- void foo(std::vector<Object> &objects)
  - Nic se nekopíruje; ve foo(std::span<Object> objects) bychom podporovali I předání částí std::vectoru a dalších (jako string_view, ale u arrayů a vectorů)
- void foo(const Object& object) [foo nemění stav objectu]
  - Funkce správně naznačuje, že object nebude měnit (a pokud v jejím těle object omylem změníme, compiler nás praští přes prsty)
- void foo(int number)
  - Číslo nemá smysl brát readonly referencí – často bývá samo jednodušší než pointer

Poslední příklad předávání objektů

- void foo(**Object&& object**)
  - Stejně jako u foo(**Object& object**), předáváme referenci
    - Ale na object, který od volání foo už nikdy nechceme použít
  - Budeme se tomu věnovat později

TO BE CONTINUED...

for loop s iterátorem (stručný úvod):
for (auto it = vec.begin(); it != vec.end(); ++i)

- Výhody oproti Cčkařské iteraci s i:

  - Nemusíme indexovat, hodnota se získá dereferencí: *it
  - Funguje na strukturách, kde je indexace pomalá (např. stromy, linked-listy)
  - Jasně vyjadřuje, že procházíme strukturu
- Inspirace pointery z C: for (char** arg = argv; arg != nullptr; ++argv)

  - Pointerem procházíme cmd argumenty (vždy končí nullem)
  - Argument přečteme dereferencí: *arg
- Co je to iterátor? Je to druh pseudo-pointeru (na prvky kontejneru)
  - Chová se jako pointer: jde ho inkrementovat, porovnávat, dereferencovat
  - Ale funguje i na stromových strukturách atd.

Range-based for: for (auto&& arg : args)

- Stejně efektivní jako správně napsaná C-style smyčka, ale jednodušší!
  - Interně používá for loopu s iterátory (viz na předchozím slajdu)
- Pattern, co znáte z jazyků jako C#, Python, atd.
- Chytře prochází prvky v `args` **bez hrozby out-of-bounds accessu**
  - **Nikde nemusíme řešit okraje kontejneru, správnou indexaci, …**
- "**auto&&**" znamená "udělej správnou referenci na hodnotu“
- Funguje i s dalšími kontejnery: např. for (auto&& [key, value] : map)
  - C++ mapy jsou asociativní key-value kontejnery (v C#/Pythonu: dictionary)
- Mnohem větší **readability** – říká, co chceme: **projít každý prvek args**

C and C++ arrays

Containers	Static array	Dynamic array
C version 	T array[N]	T *array = new T[N] // in C: T *array = malloc(sizeof(T) * N)
C++ version 	**std::array<T, N> array;** #include <array>	**std::vector<T> array(N)** **std::vector<T> array{n_1, n_2, …, n_N}** #include <vector>
Features	• Data allocated on the **stack** • The length is a compiler constant • Good if the array is quite small	• Data allocated on the **heap** (the container object itself has just a few bytes) • The length is in the container object

- The C++ versions have methods like .size(), .begin(), .end(), etc., that are often very useful
- std::vectors have .push_back(new_last), .back() to retrieve the last element and .pop_back()
  - They automatically grow in O(1), or they can be .resize(N)'d or .reserve(N)'d manually
- In a typical C++ program, **std::vector is usually the best choice** of an array type

std::span – chytrá reference na (C/C++) arraye

- Chová se jako array, ale data jsou spravována jiným objektem
- Příklad jednoduchého použití (automatické přetypování z vectoru):
  void foo(std::span<T> args) { … }
  int main() { vector<std::string> args{"hi", "hello"}; foo(args); }
- Výhody oproti normální referenci: void foo(array_type &args)
  1. Je obecnější: akceptuje různé array typy, může se koukat i jen na část arraye
  2. Nemusíme řešit na jaké části arraye funkce má pracovat: pošlem jen tu část
  3. Zpřístupňuje jen prvky, ne metody předávané struktury (např. push_back)
- Složitější použití (reference jen na část arraye):
  begin/end: int main() { … foo({args.begin() + 3, args.end()}); }
  pointery: int main() { … foo({&args[3], &args[args.size()]}); }

Odevzdání násobilky

- Soubory úlohy dejte do složky „nasobila“ v adresáři „labs“ vašeho repozitáře
  - Necommitujte binární soubory ani žádné jiné vygenerované při sestavování
  - Úloha předpokládá 3 zdrojové soubory, ale součástí mohou být projektové soubory Visual Studia, CMakeList.txt a nebo README.md
- Doma si nepovinně vypracujte EXTRA TODO a dále experimentujte s C++
  - Příště se na EXTRA TODO podíváme společně