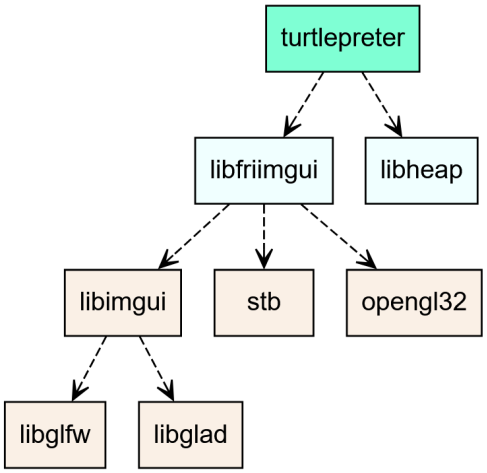
Cvičenie 5: Linkovanie knižníc

1 Obsah

Cvičenie otvára projekt, v ktorom sa budeme venovať vykresľovaniu jednoduchých obrázkov pomocou korytnačky. Správanie korytnačky – a teda aj výsledný obrázok – bude určené stromovou štruktúrou obsahujúcou príkazy pre korytnačku. Korytnačka bude riadená interpreterom stromovej štruktúry. Spojením týchto dvoch skutočností vznikol názov projektu – **Turtlepreter**. Strom správania, ktorý na cvičeniach implementujeme je zjednodušenou verziou knižnice Behavior Tree, ktorá ma veľmi široké uplatnenie.

2 Knižnice

Na vykresľovanie trajektórie korytnačky budeme používať grafické používateľské rozhranie (GUI). Toto rozhranie je už pre vás pripravené. Medzi materiálmi pre dnešné cvičenie nájdete súbor **03-turtlepreter.zip**. Tento archív si stiahnite a priečinok `03-turtlepreter` umiestnite do priečinku `cvicenia` vo vašom repozitári. Priečinok obsahuje pripravenú súborovú štruktúru s potrebnými knižnicami a nastaveným build systémom. Priečinok `turtlepreter` obsahuje `CMakeLists.txt` definujúci cieľ `turtlepreter`, v ktorom budeme programovať interpreter.



Obr. 1: Knižnice a ich závislosti v projekte Turtlepreter

Priečinok `lib` obsahuje knižnice, ktoré cieľ `turtlepreter` potrebuje. Knižnica heap je veľmi jednoduchá – obsahuje iba `heap_monitor`, ktorý používame na detekciu únikov pamäte. Knižnica `libfriimgui` vytvára používateľské rozhranie pre našu korytnačku. Na pozadí využíva state-of-the-art knižnicu Dear ImGui. Na Obr. 1 môžeme vidieť stručný prehľad použitých knižníc a závislostí medzi nimi. Tieto závislosti sú zachytené v súboroch `CMakeLists.txt` – napríklad v súbore definujúcom cieľ `turtlepreter` nájdeme riadok:

```
target_link_libraries(turtlepreter PRIVATE friimgui heap)
```

ktorý definuje, že cieľ `turtlepreter` je závislý na cieľoch `friimgui` a heap, ktoré sú definované v príslušných `CMakeLists.txt` súboroch v priečinkoch `lib/libfriimgui` a `lib/libheap`. CMake na základe závislostí generuje konfiguráciu pre build systém, ktorá automaticky preloží ciele v správnom poradí. Závislý cieľ (`turtlepreter`) navyše dedí z cieľov (heap a `friimgui`) užitočné vlastnosti ako napríklad cesty k hlavičkovým súborom. Toto nám umožní používať hlavičkový súbor `heap_monitor.hpp` rovnakým spôsobom – direktívou:

```
1 #include "heap_monitor.hpp"
```

napriek tomu, že sa nachádza v inom priečinku.

Priečinok `lib/external` obsahuje open-source knižnice glad, glfw, imgui a stb. Tieto knižnice budeme prekladať priamo z ich zdrojových kódov (angl. build from source). Ich zdrojový kód sa automaticky stiahne po spustení nástroja CMake.

Na začiatok skúsime projekt nakonfigurovať a preložiť. Použijeme na to štandardnú postupnosť príkazov spustenú v koreňovom priečinku projektu s rozšíreným volaním nástroja CMake:

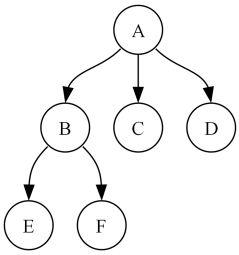
```
mkdir build
cd build
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON -DCMAKE_BUILD_TYPE=Debug ..
ninja
```

V príkaze `cmake` sme použili dva parametre:

- `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON` – CMake pri generovaní konfiguračných súborov pre build systém vygneneruje aj súbor `compile_commands.json` t, ktorý obsahuje volania kompilátora pre každý zdrojový súbor. Tento súbor býva spracovávaný nástrojmi (napríklad) a vývojovými prostrediami pre zlepšenie intellisense. Zvyčajne ho umiestňujeme v koreňovom priečinku projektu (`mv compile_commands.json ..`) a zahrnieme v súbore `.gitignore`.
- Doposiaľ sme generovanie informácií pre ladenie (debugovanie) spustiteľného súboru zapínali parametrom kompilátora `-g` nastavenom príkazom `target_compile_options`. Nevýhodou tohto prístupu je, že v budúcnosti môžeme chcieť projekt preložiť s nastaveniami pre maximálne optimalizácie (angl. release build) – pri takýchto nastaveniach nie je možné spustiteľný súbor ladiť (aspoň nie štandardným spôsobom). Nastavenie úrovne optimalizácií a informácií pre ladenie preto nenastavujeme priamo v súboroch `CMakeLists.txt`, ale práve parametrom `CMAKE_BUILD_TYPE`.

3 Strom príkazov

Správanie korytnačky budeme popisovať stromom príkazov. Strom je v informatike veľmi dôležitá údajová štruktúra. Zároveň je to špeciálny typ súvislého grafu, v ktorom je každá hrana mostom. Strom je hierarchická údajová štruktúra zložená z vrcholov (angl. node). Každý vrchol má najviac jedného *rodiča* (angl. parent) a, vo všeobecnosti, neobmedzený počet *synov*. Vrchol, ktorý nemá rodiča, nazývame *koreň* stromu (angl. root) a vrchol, ktorý nemá žiadnych synov, nazývame *list* (angl. leaf). Vrchol, ktorý nie je ani koreň ani list, nazývame *vnútorný* vrchol. Ukážku stromu môžeme vidieť na Obr. 2. Na obrázku je koreňom stromu vrchol A, vrcholy C, D, E a F sú listy a vrchol B je vnútorný vrchol.

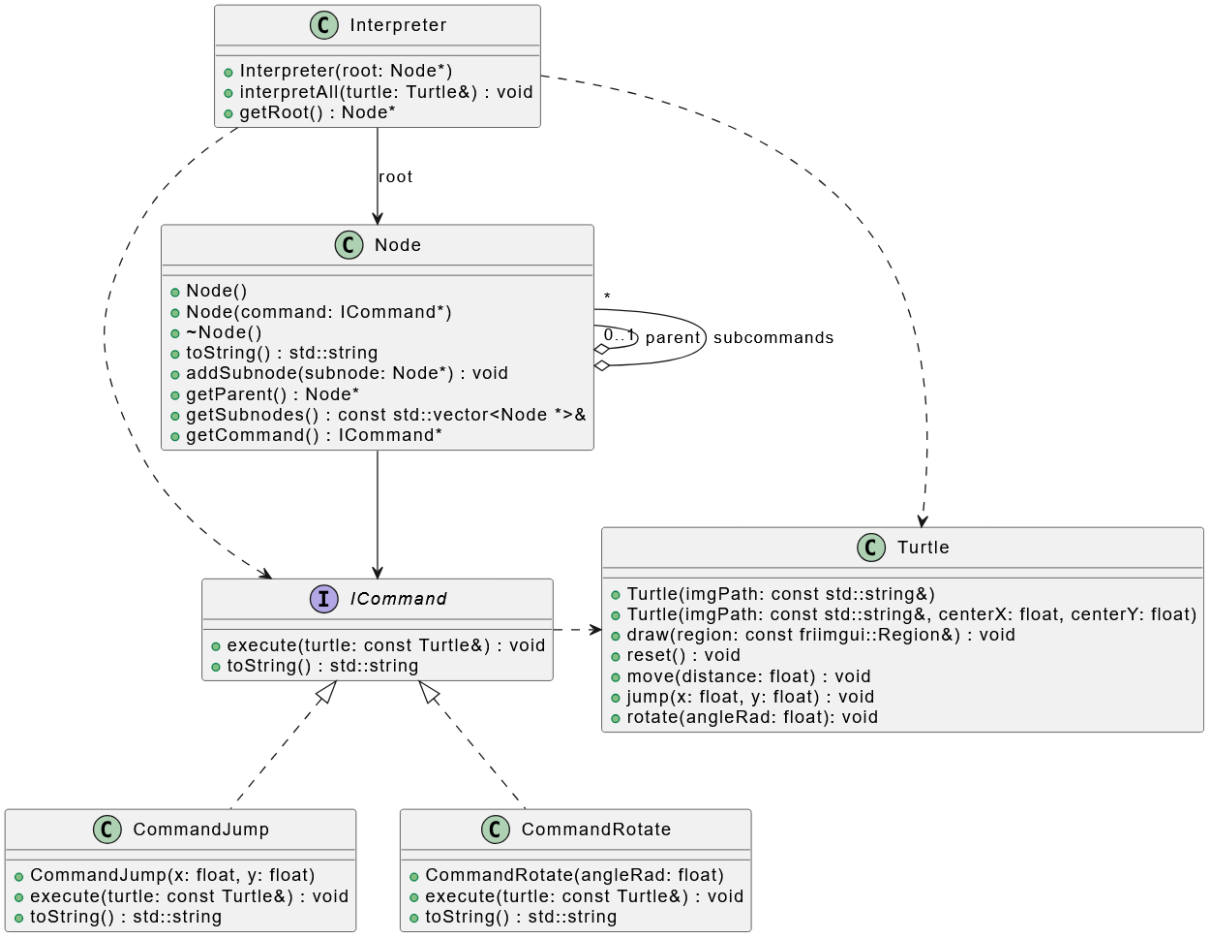


Obr. 2: Príklad stromu so šiestimi vrcholmi

Vrchol stromu budeme reprezentovať triedou `Node`. Táto bude uchovávať ukazovateľ na otca a zoznam ukazovateľov na synov. Na ich uloženie použije kontajner `std::vector`[1] – ekvivalent kontajnera `ArrayList` z jazyka Java. UML diagram triedy môžeme vidieť na Obr. 3.

Úloha 1: *Implementujte triedu* `Node` *podľa popisu, UML diagramu tried a dokumentačných komentárov. Triedu implementujte v súboroch* `interpreter.hpp` *a* `interpreter.cpp`.

Úloha 2: *Vo funkcii* `main` *vytvorte tri vrcholy – otca a jeho dvoch synov. S použitím metód triedy* `Node` *nastavte tieto vzťahy. Ukazovateľ na otca pošlite ako parameter konštruktora inštancie triedy* `Interpreter` *vo funkcii* `main`.



Obr. 3: UML diagram tried projektu Turtlepreter

[1] Názov hlavičkového súboru, ktorý je potrebné vložiť nájdete v dokumentácii

3.1 Zobrazenie stromu

Po implementácii triedy Node môžeme program preložiť a spustiť. V ľavej časti okna – pod tlačidlom "Run" – by sme chceli vidieť reprezentáciu nášho stromu. Aktuálne vidíme iba koreň stromu. Vykresľovanie rozhrania je zodpovednosťou triedy `TurtleGUI`. Vizualizácia stromu sa vytvára v jeho súkromnej metóde `buildLeftPanel`, ktorá volá metódu `populateTreeNodes` nad koreňom stromu, ktorý získa od inštancie triedu `Interpreter`.

Úloha 3: *Preskúmajte metódu* `populateTreeNodes`*. Upravte ju tak, aby bola zavolaná pre každý vrchol stromu.*

4 Príkazy

Reprezentáciu topológie stromu sme implementovali v triede Node. Pri prehliadke stromu budeme chcieť v jeho listoch vykonávať nejaké akcie – príkazy pre korytnačku. Algoritmus, ktorý sa má vykonať pri navštívení vrcholu preto potrebujeme v tomto vrchole uložiť. Zároveň chceme udržať reprezentáciu stromu nezávislú na akciách, ktoré sa v jeho vrcholoch vykonávajú.

Akcie budeme reprezentovať triedami, ktoré budú potomkami abstraktnej triedy `ICommand`[2] (Obr. 3). Do triedy Node pridáme členskú premennú typu ukazovateľ na `ICommand`. Abstraktná trieda `ICommand` obsahuje dve metódy:

```
1  /**
2   * Vykoná akciu s korytnačkou @p turtle
3   */
4  virtual void execute(Turtle &turtle) = 0;
5
6  /**
7   * Vráti reťazec popisujúci príkaz.
8   */
9  virtual std::string toString() = 0;
```

Na cvičení implementujeme príkazy `CommandJump`, `CommandRotate` a `CommandMove`. Implementácie príkazov budú vždy veľmi jednoduché – ich úlohou je zavolať zodpovedajúcu metódu korytnačky. Okrem toho musia implementácie uchovávať parametre danej operácie, ktoré prevezmú vo svojich konštruktoroch, uložia ich vo svojich atribútoch a použijú ich pri volaní metód korytnačky.

Úloha 4: *Implementujte triedy* `CommandJump`, `CommandRotate` *a* `CommandMove`*. Implementácie umiestnite do súborov* `interpreter.hpp` *a* `interpreter.cpp`*. Vo funkcii* `main` *vytvorte dve inštancie príkazu* `Jump` *a umiestnite ich do listov stromu.*

4.1 Korytnačka

Samotný algoritmus akcií `Jump` a `Rotate` implementujeme v metódach v triede `Turtle`. Realizácie rozhrania `ICommand` plnia iba úlohu "prostredníka", ktorý zavolá správnu metódu. Pozícia korytnačky v priestore je určená je **transformáciou**, ktorá sa skladá z trochu zložiek:

[2] Keďže jazyk C++ nepodporuje rozhrania, použijeme abstraktnú triedu. V jazyku Java by bola trieda ICommand rozhraním.

- **translácia** – relatívny posun od bodu $(0, 0)$
- **rotácia** – natočenie korytnačky
- **škála** – škálovanie korytnačky

Metóda `rotate` musí preto upraviť zložku **rotácia** nastavením jej hodnoty na novú hodnotu danú parametrom. Metóda `jump` musí upraviť zložku **translácia** nastavením jej hodnoty na hodnotu danú parametrami. Parametrami metódy sú súradnice x a y novej pozície, teda vektor (x, y). V knižnici Dear ImGui reprezentujeme dvojrozmerné vektory typom `ImVec2`:

```
1 const ImVec2 dest(x, y);
```

Hodnotu `dest` nastavíme do zložky **translácia**. V implementácii tejto metódy je však potrebné zohľadniť ešte jednu skutočnosť – korytnačka má svojim pohybom kresliť čiary. Vykresľovanie grafického rozhrania knižnicou Dear ImGui prebieha v slučke, v ktorej je vždy potrebné vykresliť celé rozhranie. Z rozhrania vykresleného v predchádzajúcom opakovaní slučky nezostáva nič. Korytnačka si preto musí pamätať históriu svojho pohybu, aby ju vedela celú vykresliť (v jej metóde `draw`). Cestu (históriu) uchováva v členskej premennej `m_path` typu `std::vector<ImVec4>`. Cesta je zložená zo segmentov, pričom jeden segment je reprezentovaný štvorrozmerným vektorom typu `ImVec4`[3] (prvé dve zložky pre začiatok – `pos.x` a `pos.y`, druhé dve pre koniec segmentu `dest.x` a `dest.y`). Pred zmenou translácie preto najprv získame jej aktuálnu hodnotu:

```
1 const ImVec2 pos = m_transformation.translation.getValueOrDef();
```

a spojenie zložiek `pos` a `dest` vložíme do zoznamu segmentov. Až následne zmeníme hodnotu translácie na `dest`. Škálu korytnačky zatiaľ meniť nebudeme.

Vykresľovanie korytnačky zabezpečuje jej metóda `draw`. V tejto metóde je potrebné vykresliť všetky segmenty cesty uložené v členskej premennej `m_path`. Segment vykreslíme jeho pridaním do zoznamu `drawList` metódou `AddLine`. Metóda obsahuje dokumentačný komentár popisujúci jej parametre. Niektoré parametre sú už v metóde pripravené.

Úloha 5: *Implementujte metódy triedy* `Turtle` *podľa vyššie uvedeného popisu. Metódy korytnačky a príkazy overte ich manuálnym spustením vo funkcii* `main`.

5 Interpretovanie stromu

Poslednou časťou cvičenia je implementovanie automatického vykonania všetkých príkazov po stlačení tlačidla `"Run"` v grafickom rozhraní. V súbore `turtle_gui.cpp:40` vidíme, že po stlačení tlačidla sa zavolá metóda triedy `Interpreter`. Táto trieda predstavuje posledný diel (aj keď ho ešte v budúcnosti rozšírime) univerzálneho návrhu rozdeleného do troch nezávislých častí:

- štruktúra stromu (topológia) – trieda `Node`
- akcie vykonávanie vo vrcholoch – trieda `ICommand` jej potomkovia
- riadenie vykonávania akcií – trieda `Interpreter`

[3] zložky tohto vektora majú názvy x, y, z, w

Na začiatok implementujeme veľmi jednoduché spustenie všetkých príkazov v metóde `interpretAll`. Táto metóda iba zavolá súkromnú metódu `interpterSubtreeNodes` s koreňom stromu a korytnačkou. Úlohou tejto metódy je postupne zavolať všetky príkazy uložené v listoch stromu. Môžete si všimnúť (a využiť) podobnosť s metódou `populateTreeNodes`).

6 Commit a Push

Na konci cvičenia nezabudnite na **commit** a **push** vašej práce. Váš repozitár by mal na konci cvičenia vyzerať nasledovne:

```
informatika3/
`-- cvicenia/
    |-- 01/
    |-- 02/
    `-- 03-turtlepreter/
        |-- .vscode/
        |-- lib/
        |-- resources/
        |-- turtlepreter/
        |   |-- CMakeLists.txt
        |   |-- interpreter.cpp
        |   |-- interpreter.hpp
        |   |-- main.cpp
        |   |-- turtle_gui.cpp
        |   |-- turtle_gui.hpp
        |   |-- turtle.cpp
        |   `-- turtle.hpp
        |-- CMakeLists.txt
        `-- .gitignore
```

Pre získanie bodov za commit je potrebné:

- aby bolo projekt možné preložiť štandardnou postupnosťou príkazov spustenou v koreňovom priečinku projektu (pozri Cvičenie 2)
- aby boli funkčné všetky metódy triedy `Node` uvedené v UML diagrame tried na Obr. 3.