

Příklady otázek – Programovací jazyky

Reálné otázky ze SZZ 2022–2025 (I3)

Jakub Klesa

Pokryté předměty: NPRG013 (Java) / NPRG035 (C#) / NPRG041 (C++) - dle volby jazyka

Obsah

1	Překlad konstrukcí vyššího jazyka	1
2	Programování	4
3	Objektový návrh	6
4	Komponentový systém	10
5	Reprezentace typů	13
6	Knihovna pro výpočty s maticemi	14

Poznámka. Tento dokument obsahuje reálné otázky z minulých termínů SZZ (2022–2025) zařazené do okruhu I3. U otázek s obrázkem je grafika oříznutá z originálního PDF (viz odkaz na zdroj). Text, číselná data a tabulky jsou přepsané věrně. Zadání i oficiální náčrty řešení pocházejí ze sad zveřejněných na webu MFF UK:

<https://www.mff.cuni.cz/cs/studenti/bakalarske-studium/statni-zaverecne-zkousky/bakalarske-statni-zkousky-studijniho-programu-informatika>

Náčrty označené jako doplněné jsou vlastní, oficiální sada je k dané otázce neobsahuje; vlastní řešení jsou v C++ (u otázky podzim 2022 v Javě) a jejich kód byl ověřen překladem. Pojmy odpovídají přednáškám D. Bednářka *Programování v C++* a *Pokročilé programování v C++* (<https://teaching.mff.cuni.cz/nprg041-web/>). Fialové rámečky *Lehčí v C#* ukazují kratší odpověď v C#, který zadání také dovolují.

1 Překlad konstrukcí vyššího jazyka

(Reálná otázka: léto 2022, č. 2 – Překlad konstrukcí vyššího jazyka)

Zadání.

Mějme jednoduchý procesor s architekturou inspirovanou MIPS. Procesor má 32 32-bitových celočíselných obecných registrů `$R0–$R31` a jeden 32-bitový registr `PC` ukazující na začátek následující instrukce. Hodnota registru `$R0` je vždy 0. V popisu operandů značí `label` symbolické návěští, `[mem]` adresu paměti a `reg` obecný registr.

Instrukce procesoru:

- LD `reg=[mem]` – načte celočíselnou proměnnou z adresy `mem` do registru (`reg = *mem`).
- ST `[mem]=reg` – uloží registr do proměnné na adrese `mem` (`*mem = reg`).

- `ADD reg1=reg2,reg3` – `reg1 = reg2 + reg3`.
- `J label` – nepodmíněný skok (`PC = label`).
- `BEQ reg1,reg2,label` – skok, pokud `reg1 == reg2`.
- `BNE reg1,reg2,label` – skok, pokud `reg1 != reg2`.
- `SLT reg1=reg2,reg3` – `reg1 = (reg2 < reg3)` (jinak 0).

Mějme následující posloupnost instrukcí:

```

LD $r1=[q]
LD $r2=[n]
LD $r3=[inc]
J lab1
lab2:
LD $r4=[a]
LD $r5=[b]
ADD $r4=$r4,$r1
SLT $r5=$r5,$r4
BEQ $r5,$r0,lab3
ST [b]=$r4
lab3:
ADD $r1=$r1,$r3
lab1:
SLT $r4=$r1,$r2
BNE $r4,$r0,lab2

```

1. Vyberte ze zadaných úryvků v jazyce C všechny, které odpovídají výše uvedené posloupnosti instrukcí (neznámé identifikátory považujte za někde deklarované 32-bitové celočíselné proměnné). Pro zvolené řešení připište k instrukcím čísla řádků odpovídajících příkazů v jazyce C. Nabízené úryvky (v PDF je každý rozepsán na očíslované řádky, jeden příkaz nebo hlavička na řádek: úryvky (a)–(d) mají řádky 1–3, úryvek (e) řádky 1–6):

- (a) `for (i=q; i<n; i+=inc) if (a+i <= b) b = a+i;`
- (b) `if (a+i > b) for (i=q; i<n; i+=inc) b = a+i;`
- (c) `for (i=q; i<n; i+=inc) if (a+i > b) b = a+i;`
- (d) `if (a+i < b) for (i=q; i<n; i+=inc) b = a+i;`
- (e) `i = q; while (i<n) { if (a+i > b) b = a+i; i += inc; }`
- (f) Žádný z uvedených úseků neodpovídá (napište vlastní kód).

2. Napište (stačí jeden řádek), jak se původní kód v jazyce C změní, pokud instrukci `BEQ` nahradíme instrukcí `BNE` se stejnými operandy.
3. Předpokládejme, že podmínka v příkazu `if` je rozšířena o část `&& i < b`, tj. bude mít tvar `if (původní podmínka && i < b)`. Upravte původní sekvenci instrukcí tak, aby tuto rozšířenou podmínku realizovala.

Náčrt řešení.

(V PDF bez náčrtu řešení; náčrt doplněn k výkladu okruhu, logika ověřena simulací.)

Nejprve rozluštíme posloupnost. Před cyklem se načte `q`, `n`, `inc` (do `$r1`, `$r2`, `$r3`) a skočí se na `lab1`. Tam je test cyklu: `SLT $r4=$r1,$r2` nastaví `$r4 = (i < n)` (řídící proměnná `i`

je v \$r1, počáteční hodnota q) a BNE \$r4,\$r0,lab2 skočí do těla, je-li \$r4 != 0, tj. *dokud platí* i < n. V těle se spočítá \$r4 = a + i, pak SLT \$r5=\$r5,\$r4 dá \$r5 = (b < a+i) a BEQ \$r5,\$r0,lab3 *přeskočí* uložení ST [b]=\$r4, když \$r5 == 0, tj. když a+i <= b. Uložení b = a+i se tedy provede právě tehdy, když a+i > b. Na lab3 se i zvýší o inc a běh pokračuje testem cyklu.

(1) Posloupnost realizuje

```
for (i=q; i<n; i+=inc)
    if (a+i > b) b = a+i;
```

a ekvivalentně její while-rozpis

```
i = q;
while (i<n) { if (a+i > b) b = a+i; i += inc; }
```

Odpovídají tedy *obě* varianty: (c) for s podmínkou a+i > b i (e) ekvivalentní while. Varianty (a) (opačná podmínka <=), (b)/(d) (zaměněné pořadí for a if, resp. ostré <) neodpovídají.

Čísla řádků k instrukcím. Úryvek (c): řádek 1 for (i=q; i<n; i+=inc), 2 if (a+i > b), 3 b = a+i;. Úryvek (e): 1 i = q;, 2 while (i<n) {, 3 if (a+i > b), 4 b = a+i;, 5 i += inc;, 6 }.

instrukce	(c)	(e)
LD \$r1=[q]	1	1
LD \$r2=[n], LD \$r3=[inc]	1	2, 5
J lab1	1	2
LD \$r4=[a], LD \$r5=[b], ADD \$r4=\$r4,\$r1, SLT \$r5=\$r5,\$r4, BEQ	2	3
ST [b]=\$r4	3	4
ADD \$r1=\$r1,\$r3	1	5
SLT \$r4=\$r1,\$r2, BNE	1	2

Součet a+i v \$r4 spočítaný pro podmínku se znovu použije jako přiřazovaná hodnota na řádku s b = a+i (proto je ADD \$r4=\$r4,\$r1 vlastně společný podmínce i přiřazení). Test i<n je v posloupnosti *na konci* cyklu a J lab1 na něj skočí hned na začátku, což je obvyklý překlad cyklu for/while.

(2) Záměna BEQ za BNE (se stejnými operandy \$r5,\$r0,lab3) obrátí podmínku skoku: přeskočení uložení nastane, když \$r5 != 0, tj. když a+i > b. Uložení se proto provede v opačném případě, takže podmínka v C se změní na

```
if (a+i <= b) b = a+i;
```

(3) Rozšíření o && i < b: před instrukci ST [b]=\$r4 vložíme druhý test. Načteme b (do \$r6), spočítáme \$r6 = (i < b) a BEQ \$r6,\$r0,lab3 přeskočí uložení, když i < b neplatí. Uloží se tedy jen při splnění obou podmínek (a+i > b a zároveň i < b). Upravená posloupnost (komentáře bez diakritiky):

```
lab2:
    LD $r4=[a]
    LD $r5=[b]
    ADD $r4=$r4,$r1
    SLT $r5=$r5,$r4 ; r5 = (b < a+i)
    BEQ $r5,$r0,lab3 ; preskoc, kdyz a+i <= b
    LD $r6=[b] ; pridany test i < b
    SLT $r6=$r1,$r6 ; r6 = (i < b)
    BEQ $r6,$r0,lab3 ; preskoc, kdyz i < b neplati
    ST [b]=$r4 ; ulozi se jen kdyz a+i > b AND i < b
lab3:
    ADD $r1=$r1,$r3
```

2 Programování

(Reálná otázka: podzim 2022, č. 1 – Programování (objektové rozhraní pro grafy))

Zadání.

Pro tuto otázku si vyberte libovolný mainstreamový objektově orientovaný staticky typovaný jazyk (C++, C# nebo Java) a svůj výběr napište na začátek řešení.

1. Naimplementujte (nadeklarujte) vhodné objektové rozhraní (pomocí *interfaces* nebo abstraktních tříd) pro neorientované grafy $G = (V, E)$ ve své nejběžnější podobě (hrany spojují právě dva vrcholy, mezi dvěma vrcholy vede nejvýš jedna hrana). Motivací je, aby toto rozhraní mohly implementovat různé reprezentace grafů (seznam sousedů, matice sousednosti, ...), zatímco programátor grafových algoritmů (BFS, hledání minimální kostry, ...) může napsat svůj kód pouze jednou a bude fungovat se všemi reprezentacemi.

Rozhraní pokrývá 3 entity (vrchol, hranu a graf): vrchol nabídne přístup ke všem incidentním hranám, hrana ke svým dvěma vrcholům, graf k seznamu všech vrcholů a všech hran. Graf je pouze pro čtení (bez modifikací).

Každý vrchol a každá hrana má příznak (*tag*) – u vrcholu např. zda byl navštíven, u hrany např. váhu. Tagy mohou mít různé datové typy, ale pro danou instanci grafu mají všechny vrcholy stejný typ tagu a všechny hrany taktéž. Rozhraní má být generické (uživatel zvolí typy tagů vrcholů a hran) a nabídne pro každý vrchol/hranu prostředky tag přečíst a nastavit.

2. Uvažme implementaci, kde tagy vrcholů jsou `int` (index komponenty souvislosti, od 0) a tagy hran jsou `float` (délky hran, na počátku inicializované). Napište tělo funkce využívající vaše rozhraní, která dostane graf a reálné číslo r a provede rozklad grafu na komponenty souvislosti tak, že každý vrchol bude mít index své komponenty uložený jako tag (konkrétní hodnoty indexů nejsou podstatné, jen že vrcholy téže komponenty mají stejný tag). Hrany s délkou (tagem) větší než r se neberou v potaz (jako by neexistovaly).

Náčrt řešení.

(V PDF bez náčrtu řešení; náčrt doplněn k výkladu okruhu.)

Zvolme **Java**. Generika nesou typy tagů: `VTag` pro vrcholy, `ETag` pro hrany.

1. Tři spolupracující generická rozhraní (pouze deklarace, bez těl):

```
interface IVertex<VTag, ETag> {
    Iterable<IEdge<VTag, ETag>> incidentEdges(); // hrany incidentni s vrcholem
    VTag getTag();
    void setTag(VTag tag);
}

interface IEdge<VTag, ETag> {
    IVertex<VTag, ETag> endpoint1(); // jeden z dvou koncovych vrcholu
    IVertex<VTag, ETag> endpoint2(); // druhy koncovy vrchol
    ETag getTag();
    void setTag(ETag tag);
}

interface IGraph<VTag, ETag> { // graf je pouze pro cteni
    Iterable<IVertex<VTag, ETag>> vertices();
    Iterable<IEdge<VTag, ETag>> edges();
}
```

Díky generikám je rozhraní nezávislé na konkrétním typu tagu i na vnitřní reprezentaci grafu (seznam sousedů, matice sousednosti, ...). Algoritmus se proti rozhraní napíše jednou a běží nad libovolnou implementací.

2. BFS z každého dosud neoznačeného vrcholu; šíříme se jen po hranách s tagem $\leq r$, ostatní (delší) hrany ignorujeme, jako by neexistovaly. Všem vrcholům dosaženým v jednom průchodu nastavíme stejný index komponenty, čítač pak zvýšíme.

```
static void connectedComponents(IGraph<Integer, Float> g, float r) {
    final int UNVISITED = -1;
    for (IVertex<Integer, Float> v : g.vertices()) {
        v.setTag(UNVISITED); // inicializace: nikdo neoznačen
    }

    int component = 0;
    for (IVertex<Integer, Float> start : g.vertices()) {
        if (start.getTag() != UNVISITED) {
            continue; // již přiřazen do komponenty
        }
        // BFS z vrcholu start, vse dosazene dostane index "component"
        Queue<IVertex<Integer, Float>> queue = new ArrayDeque<>();
        start.setTag(component);
        queue.add(start);
        while (!queue.isEmpty()) {
            IVertex<Integer, Float> u = queue.remove();
            for (IEdge<Integer, Float> e : u.incidentEdges()) {
                if (e.getTag() > r) {
                    continue; // příliš dlouhá hrana, ignoruj
                }
                // určí druhý konec hrany (graf je neorientovaný)
                IVertex<Integer, Float> w =
                    (e.endpoint1() == u) ? e.endpoint2() : e.endpoint1();
                if (w.getTag() == UNVISITED) {
                    w.setTag(component);
                    queue.add(w);
                }
            }
        }
        component++; // další komponenta
    }
}
```

Hrany s tagem $> r$ se v BFS přeskočí, takže rozklad odpovídá podgrafu jen s hranami délky $\leq r$. Protože pracujeme výhradně přes rozhraní (`vertices()`, `incidentEdges()`, `getTag/setTag`), je tělo funkce stejné pro každou reprezentaci grafu.

Lehčí v C# — generická rozhraní grafu

Stejný návrh je v C# přímočařejší: rozhraní mají vlastní klíčové slovo `interface`, tag je *vlastnost* s `get/set` místo dvojice metod a kolekce se vrací jako `IEnumerable` (procházené přes `foreach`). Tělo BFS zůstává nezávislé na reprezentaci grafu.

```
interface IVertex<VTag, ETag> { VTag Tag { get; set; }
    IEnumerable<IEdge<VTag, ETag>> IncidentEdges { get; } }
interface IEdge<VTag, ETag> { ETag Tag { get; set; }
    IVertex<VTag, ETag> Endpoint1 { get; } IVertex<VTag, ETag> Endpoint2 { get; } }
```

```
interface IGraph<VTag, ETag> { IEnumerable<IVertex<VTag, ETag>> Vertices { get; }  
    IEnumerable<IEdge<VTag, ETag>> Edges { get; } } // jen pro ctení
```

Proti C++ odpadá virtuální destruktork i ruční správa životnosti uzlů a hran (uklízí GC).

3 Objektový návrh

(Reálná otázka: jaro 2023, č. 2 – Objektový návrh (bitmapový editor, polymorfismus, pluginy/DLL))
Zadání.

Předpokládejte, že implementujeme aplikaci pro úpravu bitmapových obrázků. Uživatel bude moci mít otevřených více obrázků najednou. Nový obrázek si uživatel otevře v uživatelském rozhraní aplikace (UI) výběrem jména souboru – aplikace dle přípony souboru rozhodne o jeho formátu. U otevřeného obrázku si uživatel může postupně vybírat některé z nabízených filtrů – v UI aplikace si uživatel vždy vybere filtr a jeho konkrétní nastavení, ten se uloží do seznamu vybraných filtrů. Nakonec bude mít uživatel možnost všechny filtry najednou aplikovat. Každý otevřený soubor si uživatel může kdykoliv uložit buď v původním nebo jiném podporovaném bitmapovém formátu. Pro jednoduchost předpokládejte, že jednomu formátu obrázku odpovídá právě jedna přípona souboru (tj. např. pro JPEG formát jen přípona .jpg a nikoliv už .jpeg), a že všechny filtry mohou libovolně měnit obrazová data, ale rozměry obrázku v pixelech vždy zůstávají zachované.

Zatím máme v C# připravenou níže uvedenou kostru hlavních tříd našeho programu, kde třída Image reprezentuje načtená obrazová data (další metadata jako původní jméno souboru atd. si budeme pamatovat jinde; poznámka: [,] v C# znamená dvojrozměrné pole), a metoda Program.Main představuje jednoduchý scénář využití uvedených tříd:

```
struct Color { public byte R, G, B; }  
  
static class Filters {  
    public static void ApplySharpness(Image image, float amount, float radius) { ... }  
    public static void ApplyBrightness(Image image, float amount) { ... }  
}  
  
abstract class Image {  
    public Color[,] bitmap = null;  
    public abstract void Load(string fileName);  
    public abstract void Save(string fileName);  
}  
  
class PngImage : Image {  
    public override void Load(string fileName) { ... }  
    public override void Save(string fileName) { ... }  
}  
  
class JpegImage : Image {  
    public override void Load(string fileName) { ... }  
    public override void Save(string fileName) { ... }  
}  
  
static class ImageLoader {  
    public static Image Load(string fileName) {  
        Image image;  
        if (fileName.EndsWith(".png")) {  
            image = new PngImage();  
        } else if (fileName.EndsWith(".jpg")) {
```

```

        image = new JpegImage();
    } else {
        throw new ArgumentException();
    }
    image.Load(fileName);
    return image;
}
}

class Program {
    public static void Main() {
        var fileName = "a.jpg";
        var image = ImageLoader.Load(fileName);
        Filters.ApplySharpness(image, 9.0f, 1.3f);
        Filters.ApplyBrightness(image, 1.7f);
        image.Save(fileName);
    }
}

```

Přepracujte celý objektový návrh tak, aby lépe umožňoval implementovat, udržovat a dále rozšiřovat uvedenou aplikaci. Novou kostru kódu запиšte v C#, C++, nebo Javě. Soustředte se na návrh typů a jejich vztahů (vše zapisujte v syntaxi zvoleného jazyka), deklaraci metod a pouze klíčových vlastností (properties) a datových položek. Implementaci většiny metod uvádět nemusíte – pouze upravte implementaci `Program.Main` tak, aby vhodně využívala váš OO návrh. Ve svém řešení uveďte také implementaci kódu, který bude ve vašem návrhu implementovat posloupnost kroků od jména obrazového souboru k výběru formátu souboru a načtení obrázku v tomto formátu.

Ve svém návrhu zohledněte nejen snadnou možnost implementace všech aspektů výše načrtnutého scénáře používání aplikace, ale také to, že do budoucna budeme chtít přidávat velké množství různých filtrů – implementaci nového filtru ale vždy plánujeme doplňovat do hlavních zdrojových kódů aplikace; nepředpokládáme, že bychom nové filtry dodávali ve formě pluginů. Tedy ve svém řešení zohledněte pouze možnost batchové aplikace předvybraných filtrů, ale nemusíte řešit žádnou komplikovanou infrastrukturu pro jejich obecný popis.

Formou pluginů naopak budeme chtít přidávat podporu pro nové obrazové formáty. Předpokládejte tedy, že každý nově podporovaný obrazový formát budeme implementovat ve formě nového pluginu naší aplikace (který budeme za běhu načítat jako dynamicky linkovanou knihovnu) – chceme tedy, aby proces načítání obrázků ze souboru byl dostatečně flexibilní a umožňoval nám za běhu dodat odkaz na nový formát souborů, který bychom našli v nějakém pluginu (samotné načítání pluginů a podporu pro načítání dynamicky linkovaných knihoven, ani přímo hledání popisu formátu v pluginu neřešte).

Náčrt řešení.

(V PDF bez náčrtu řešení; náčrt doplněn k výkladu okruhu: vlastní vzorové řešení v C++, ověřené překladem.)

Slabiny původní kostry: filtry jsou `static` metody (nejdou rozšiřovat bez zásahu do třídy `Filters`), výběr formátu je napevno v `ImageLoader` (nejde dodat formát z pluginu za běhu) a `Image` míchá obrazová data se čtením/zápisem. Přepracovaný návrh oddělí tři odpovědnosti do rozhraní a zavede registr formátů, do něhož smí plugin přidat formát za běhu.

(1) Obrazová data jako hodnotový typ `Bitmap` (rozměry se nemění):

```

struct Color { unsigned char r, g, b; };

class Bitmap { // hodnotovy typ: drzi jen pixely
    int w_, h_;

```

```

    std::vector<Color> px_; // w_*h_ pixelu, rozmery se nemeni
public:
    Bitmap(int w, int h) : w_(w), h_(h), px_(w * h) {}
    int width() const { return w_; }
    int height() const { return h_; }
    Color& at(int x, int y) { return px_[y * w_ + x]; }
    const Color& at(int x, int y) const { return px_[y * w_ + x]; }
};

```

(2) Filtr a formát jako rozhraní (entitní typy, drží se přes unique_ptr):

```

struct IFilter { // jeden filtr = jedna trida
    virtual ~IFilter() = default;
    virtual void Apply(Bitmap& bmp) const = 0;
};

struct IImageFormat { // cteni/zapis jednoho formatu
    virtual ~IImageFormat() = default;
    virtual Bitmap Load(const std::string& file) const = 0;
    virtual void Save(const std::string& file, const Bitmap& bmp) const = 0;
};

class Sharpness : public IFilter {
    float amount_, radius_;
public:
    Sharpness(float a, float r) : amount_(a), radius_(r) {}
    void Apply(Bitmap&) const override { /* ... */ }
};

class Brightness : public IFilter {
    float amount_;
public:
    explicit Brightness(float a) : amount_(a) {}
    void Apply(Bitmap&) const override { /* ... */ }
};

class PngFormat : public IImageFormat { /* Load, Save */ };
class JpegFormat : public IImageFormat { /* Load, Save */ };

```

(3) **Registr formátů** – klíč je přípona, hodnota je továrna na formát; plugin zaregistruje nový formát za běhu jediným voláním:

```

class FormatRegistry {
    std::map<std::string, std::function<std::unique_ptr<IImageFormat>()>> reg_;
public:
    void Register(const std::string& ext,
                  std::function<std::unique_ptr<IImageFormat>()>> factory) {
        reg_[ext] = std::move(factory); // sem smí sahnout i plugin za behu
    }
    std::unique_ptr<IImageFormat> ForFile(const std::string& file) const {
        auto it = reg_.find(file.substr(file.rfind('.')));
        if (it == reg_.end()) throw std::runtime_error("neznamy format");
        return it->second();
    }
};

```

(4) Dokument drží data, formát a seznam vybraných filtrů:

```

class Document {
    Bitmap bmp_;
    std::unique_ptr<IImageFormat> format_;
    std::vector<std::unique_ptr<IFilter>> filters_;
public:

```

```

Document(Bitmap b, std::unique_ptr<IImageFormat> f)
    : bmp_(std::move(b)), format_(std::move(f)) {}
void AddFilter(std::unique_ptr<IFilter> f) { filters_.push_back(std::move(f)); }
void ApplyAll() { for (auto& f : filters_) f->Apply(bmp_); }
void Save(const std::string& file) { format_->Save(file, bmp_); } // v puvodnim
    formatu
// ulozeni v jinem formatu: format vybrán podle cilove prípony (symetrické k
    ForFile)
void SaveAs(const FormatRegistry& reg, const std::string& file) { reg.ForFile(file)
    ->Save(file, bmp_); }
};

```

(5) Scénář (cesta od jména souboru přes výběr formátu k načtení a úpravám):

```

int main() {
    FormatRegistry reg;
    reg.Register(".png", []{ return std::make_unique<PngFormat>(); });
    reg.Register(".jpg", []{ return std::make_unique<JpegFormat>(); });

    std::string file = "a.jpg";
    auto fmt = reg.ForFile(file); // dle prípony vyber format
    Bitmap bmp = fmt->Load(file); // nacti obrazek v tom formatu
    Document doc(std::move(bmp), std::move(fmt));
    doc.AddFilter(std::make_unique<Sharpness>(9.0f, 1.3f));
    doc.AddFilter(std::make_unique<Brightness>(1.7f));
    doc.ApplyAll();
    doc.Save("a.jpg"); // v puvodnim formatu
    doc.SaveAs(reg, "b.png"); // nebo v jinem podporovanem formatu
}

```

Nový filtr = nová třída implementující `IFilter` (přidá se do kódu, nic dalšího se nemění).
 Nový formát = nová třída implementující `IImageFormat` plus jeden řádek `reg.Register`, který
 smí zavolat i plugin za běhu (požadovaná flexibilita načítání formátů). Uložení v jiném formátu
 řeší `SaveAs`, jež vybere formát podle cílové přípony přes týž registr (`Save` ukládá v původním
 formátu). `Bitmap` je hodnotový typ, `IFilter/IImageFormat` entitní (drženy přes `unique_ptr`);
 o úklid se stará RAII.

Lehčí v C# — rozhraní + zásuvné moduly

Tatáž architektura je v C# kratší a bezpečnější. Filtry i formáty jsou `interface` (bez
 abstraktní třídy a virtuálního destruktora), o úklid se stará GC (žádné `unique_ptr`).
 Klíčová výhoda u *pluginů*: formát z DLL se načte reflexí, ne ručním `dlopen / LoadLibrary`
 a továrnami:

```

interface IFilter { void Apply(Bitmap b); }
interface IImageFormat { string Extension { get; } // napr. ".png"
    Bitmap Load(string path); void Save(Bitmap b, string path); }

// nacteni pluginu za behu: najdi vsechny IImageFormat v DLL a zaregistruj
foreach (var asm in Directory.GetFiles("plugins", "*.dll").Select(Assembly.
    LoadFrom))
    foreach (var t in asm.GetTypes()
        .Where(t => typeof(IImageFormat).IsAssignableFrom(t) && !t.
            IsAbstract))
    { var f = (IImageFormat)Activator.CreateInstance(t!);
        registry.Register(f.Extension, f); // prípona -> format
    }

```

Nový filtr/formát = nová třída implementující rozhraní; plugin se přidá bez rekompile jádra.

4 Komponentový systém

(Reálná otázka: léto 2023, č. 18 – Komponentový systém (herní framework, generika, komponenty))
(Pozn.: v rámci současné akreditace nešlo o společnou otázku (q1–q4), ale o otázku jiné specializace (specializační sekce, ne UI). Do společného okruhu I3 tedy formálně nepatří; tematicky sem ovšem spadá a je výborná k procvičení.)

Zadání.

Naprogramujte v C++, C# nebo Javě všechny níže zmíněné a další potřebné typy (včetně zmíněných metod) pro implementaci následujícího komponentového herního frameworku. Ve hře vytvořené pomocí vašeho frameworku má každý objekt ve scéně (např. ležící kámen, nebo odpočívající trol) být reprezentovaný instancí typu `GameObject` (dále jen GO). Samotný GO má reprezentovat množinu komponent – kde komponenta je pro nás libovolný vhodný typ (dle vašeho návrhu), jehož instance přidává k instanci GO nějaká silně typovaná data, a/nebo nějakou další funkcionalitu. Od každého druhu komponenty může být na instanci GO připojena právě 0 nebo 1 instance daného druhu komponenty. Každá instance nějaké komponenty může být připojena na maximálně 1 GO.

1. Na GO by měla být vhodná metoda pro připojení instance nějaké komponenty ke GO – pokud GO komponentu stejného druhu již má, tak nová instance nahradí instanci původní.

Na GO by měla být vhodná metoda pro vrácení instance komponenty zvoleného druhu (požadovaný druh si nějak vhodně může zvolit volající). Pokud na GO žádná instance požadovaného typu komponenty připojená není, tak to má funkce nějak rozumně indikovat (funkci budeme chtít běžně používat pro detekci, zda daný typ komponenty na GO je, nebo není).

Na GO má být metoda `Update` s parametrem `timeDelta` jako 32-bitové float číslo, které reprezentuje čas v sekundách od předchozího volání metody `Update` na stejném GO. Metoda `Update` na GO má projít všechny instance připojených komponent, a pro komponenty, které mají metodu `Update` se stejným parametrem, se má tato zavolat. Pořadí volání metod `Update` na komponentách GO není důležité. Ve svém objektovém návrhu vhodně zohledněte, že autor komponenty má mít možnost se rozhodnout, zda komponenta bude metodu `Update` mít nebo nikoliv.

2. Napište implementaci komponenty `Position`, která ke GO přidává 32-bitové float souřadnice X a Y reprezentující polohu GO na 2D herní ploše. Napište implementaci komponenty `Health`, která ke GO přidává 32-bitové float hodnoty maximálního a aktuálního zdraví (aktuální zdraví je vždy v rozsahu 0 až max včetně).

Napište příklad kódu, který vytvoří instanci GO pro kámen, který má komponentu `Position`, a instanci GO pro trola, který má komponenty `Position` a `Health` (max = 30, aktuální = 20).

3. Doplňte vhodné komponenty reprezentující efekt otravy a efekt krvácení – oba efekty vyžadují, aby GO, ke kterému jsou připojené, měl komponentu `Health` (pokud GO takovou komponentu nemá, tak efekt nic nedělá). Dále jsou oba efekty parametrizované 32-bit float hodnotou, která reprezentuje, kolik aktuálního zdraví efekt odečítá každou sekundu. Pokud je efekt připojený k nějakému GO, tak má dojít k úpravě zdraví GO dle parametru efektu. Na jednom GO může být maximálně 1 efekt otravy a nezávisle maximálně 1 efekt krvácení (tedy GO může být bez efektu, nebo mít 1 otravu a žádné krvácení, nebo žádnou otravu a 1 krvácení, nebo 1 otravu a 1 krvácení). Do budoucna budeme přidávat další podobné efekty.

Napište implementaci funkce, která dostane seznam GO a hodnotu zranění za sekundu – funkce projde všechny předané GO a pokud GO má komponentu `Health`, tak ke GO připojí novou instanci efektu otravy nebo krvácení parametrizované předanou hodnotou zranění za sekundu. Volající musí mít navíc možnost si nějak vhodně zvolit, zda funkce aplikuje na předané GO efekt otravy nebo efekt krvácení.

Napište příklad kódu, který na výše vytvořený kámen a trola zkusí aplikovat efekt krvácení s hodnotou zranění 1.2 za sekundu.

Náčrt řešení.

(V PDF bez náčrtu řešení; náčrt doplněn k výkladu okruhu: vlastní vzorové řešení v C++, ověřené překladem a během.)

Klíčová myšlenka: `GameObject` drží komponenty v mapě klíčované *typem* komponenty (`std::type_index`). Tím je „nejvýš jedna instance daného druhu“ zdarma a vrácení komponenty podle typu je triviální. Volitelnou metodu `Update` řeší zvláštní rozhraní `IUpdatable`.

(1) Komponenta, rozhraní `IUpdatable` a `GameObject`:

```
struct Component { virtual ~Component() = default; };
struct IUpdatable { // volitelné chování
    virtual ~IUpdatable() = default;
    virtual void Update(float dt) = 0;
};

class GameObject {
    std::unordered_map<std::type_index, std::unique_ptr<Component>> comps_;
public:
    template <typename T> void Add(std::unique_ptr<T> c) {
        comps_[std::type_index(typeid(T))] = std::move(c); // nahradí starý druh
    }
    template <typename T> T* Get() const {
        auto it = comps_.find(std::type_index(typeid(T)));
        return it == comps_.end() ? nullptr // indikace "nemam"
            : static_cast<T*>(it->second.get());
    }
    void Update(float dt) {
        for (auto& [type, c] : comps_)
            if (auto* u = dynamic_cast<IUpdatable*>(c.get())) // jen kdo Update má
                u->Update(dt);
    }
};
```

(2) Konkrétní komponenty a scéna (kámen, trol):

```
struct Position : Component { float x = 0, y = 0; };
class Health : public Component { // invariant: 0 <= cur_ <= max_
    float max_, cur_;
public:
    Health(float max, float cur) : max_(max), cur_(std::clamp(cur, 0.0f, max)) {}
    float Max() const { return max_; }
    float Current() const { return cur_; }
    void SetCurrent(float v) { cur_ = std::clamp(v, 0.0f, max_); } // hlída rozsah
};

auto kamen = std::make_unique<GameObject>();
kamen->Add(std::make_unique<Position>());
auto trol = std::make_unique<GameObject>();
trol->Add(std::make_unique<Position>());
trol->Add(std::make_unique<Health>(30.0f, 20.0f)); // max = 30, aktuální = 20
```

(3) Efekty. Efekt je komponenta *a zároveň* `IUpdatable`; v `Update` si najde `Health` svého GO a ubírá zdraví. Otrava a krvácení sdílejí logiku, ale musejí to být *dva různé typy* (mapa klíčeje podle typu, jinak by se přepsaly), aby mohly být na GO obě naráz. Volbu efektu předáme šablonovým parametrem typu:

```
class Effect : public Component, public IUpdatable {
protected:
    GameObject* owner_; float perSec_; // pozorovatel vlastnika + parametr
public:
    Effect(GameObject* o, float p) : owner_(o), perSec_(p) {}
    void Update(float dt) override {
        if (auto* hp = owner_->Get<Health>()) // nic, chybi-li Health
            hp->SetCurrent(hp->Current() - perSec_ * dt);
    }
};
struct Poison : Effect { using Effect::Effect; }; // dva ruzne typy,
struct Bleeding : Effect { using Effect::Effect; }; // aby sly na GO obe naraz

template <typename E> // volba efektu = typovy parametr
void applyEffect(const std::vector<GameObject*>& gos, float perSec) {
    for (GameObject* go : gos)
        if (go->Get<Health>()) // jen na GO s komponentou Health
            go->Add(std::make_unique<E>(go, perSec));
}
// pouziti: na kamen a trola zkusime krvaceni 1.2/s (kamen nema Health -> preskoci)
applyEffect<Bleeding>({kamen.get(), trol.get()}, 1.2f);
```

Klíčování podle typu (`type_index`) splní „nejvýš jedna instance daného druhu“ zadarmo; držení přes `unique_ptr` zaručí, že komponenta patří nejvýš jednomu GO; zapouzdřená `Health` hlídá invariant $0 \leq \text{aktuální} \leq \text{max}$ ze zadání na jediném místě; `Get<T>()` vrací `nullptr` při absenci (vhodná indikace pro dotaz „má GO tuto komponentu?“); volitelnost `Update` přes `IUpdatable` a `dynamic_cast` respektuje, že ne každá komponenta `Update` má. Volba efektu v šablonovém parametru je statický polymorfismus; alternativou je běhový příznak nebo tovární funkce.

Lehčí v C# — komponenty přes generika

Komponentový systém je idiomatický v C# (styl Unity). `Get<T>()` přes generika s omezením vrací komponentu nebo `null`; volba efektu je generický typový parametr. Bez hlaviček, bez instanciačních chyb šablon, bez ruční správy paměti:

```
class GameObject {
    readonly Dictionary<Type, Component> comps = new();
    public T? Get<T>() where T : Component // null, chybi-li komponenta
        => comps.TryGetValue(typeof(T), out var c) ? (T)c : null;
    public void Add(Component c) => comps[c.GetType()] = c; // klic = skutecny typ
}
abstract class Effect : Component, IUpdatable {
    public GameObject Owner { get; set; } = null!;
    public float PerSec { get; set; }
    public void Update(float dt) { var h = Owner.Get<Health>(); if (h != null) h.
        Current -= PerSec * dt; }
}
sealed class Poison : Effect { } sealed class Bleeding : Effect { }
void ApplyEffect<E>(IEnumerable<GameObject> gos, float perSec) where E : Effect,
    new() {
    foreach (var go in gos)
        if (go.Get<Health>() != null) // jen na GO s komponentou Health
```

```

    go.Add(new E { Owner = go, PerSec = perSec });
}

```

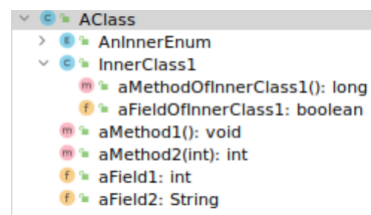
Klíčování slovníkem podle Type dá „nejvýš jedna instance druhu“ zdarma; volitelný Update řeší rozhraní IUpdatable a test if (c is IUpdatable u) místo dynamic_cast. Health.Current je vlastnost, jejíž set ořízne hodnotu do rozsahu 0 až max (není vypsáno).

5 Reprezentace typů

(Reálná otázka: podzim 2024, č. 2 – Reprezentace typů (návrh tříd/interfaců pro IDE))

Zadání.

1. Představte si, že vyvíjíte integrované vývojové prostředí (IDE). V okně projektu je potřeba zobrazit typy a elementy typů (tj. jejich metody, datové položky, vnitřní/vnořené typy – viz obrázek níže). Ve zvoleném jazyce (Java, C++, C#) navrhnete vhodnou sadu tříd/interfaců, které vám umožní uchovávat informace o typech tak, aby mohly být zobrazeny v okně v IDE. Informace zahrnují alespoň názvy typů, jejich druhy, metody typů a jejich signatury, datové položky a vnitřní/vnořené typy. Navrhnete vaše řešení tak, aby bylo v budoucnu snadno rozšiřitelné. Vytvořte pouze deklarace tříd/interfaců, tj. bez těl jejich metod.



2. Implementujte metodu, která bere jako parametry (i) typ (jak jste jej definovali pro část otázky výše) a (ii) „operaci“. Metoda aplikuje danou operaci na daný typ a všechny elementy typu. Pokud založíte procházení struktury typu na nějakém obecně známém řešení, explicitně toto řešení pojmenujte. Pro operaci použijte vhodný koncept, který nabízí vámi zvolený jazyk.
3. Napište kód, který zavolá vaší metodu z předchozího bodu. Operace předaná metodě je: „vytiskne název elementu, ale pouze v případě, že element reprezentuje datovou položku nebo typ (včetně vnitřních/vnořených typů)“.

Náčrt řešení.

1. Existuje více správných řešení. Typické řešení je obvyklá hierarchie tříd, např. abstraktní třída `Element`, a dále pak třídy, které od `Element` dědí, tj. třída `Type`, třída `Method`, třída `Field`, atd.
2. Existuje více správných řešení. Průchod strukturou může být založen např. na vzoru „Visitor“, nebo mohou být typy deklarovány jako „sealed“ (pokud to jazyk umožňuje) a průchod používá pattern matching, atd.
Pro reprezentaci operace lze použít interface nebo funkcionální typ apod.
3. V závislosti na reprezentaci bude operace definována jako anonymní třída, lambda výraz atd.

Lehčí v C# — sealed + pattern matching místo Visitoru

Náčrt sám zmiňuje „sealed“ a pattern matching. Pattern matching mají C# i Java: průchod je `switch` s typovými vzory — bez psaní celé hierarchie `Visit/Accept` jako v C++. (V C# `sealed` jen zakáže další dědění z listových typů; skutečně uzavřenou hierarchii má Java (`sealed interface ... permits`, od verze 17) a `switch` s typovými vzory nad ní kontroluje úplnost od verze 21.)

```
abstract record Element(string Name);
sealed record TypeElem(string Name, IReadOnlyList<Element> Members) : Element(Name);
sealed record Field(string Name) : Element(Name);
sealed record Method(string Name) : Element(Name);

void Walk(Element e, Action<Element> op) {
    switch (e) { // pattern matching = nahrada Visitoru
        case TypeElem t: op(t); foreach (var m in t.Members) Walk(m, op); break;
        case Field or Method: op(e); break;
    }
}
// volani: vytiskni jen datove polozky a typy
Walk(root, x => { if (x is Field or TypeElem) Console.WriteLine(x.Name); });
```

Operace se předá jako lambda (delegát `Action`) — není nutná anonymní třída.

6 Knihovna pro výpočty s maticemi

(Reálná otázka: jaro 2025, č. 2 – Knihovna pro výpočty s maticemi (kontrakt/rozhraní))

Zadání.

Poznámky, jak odpovídat na otázku:

- Pro odpověď si vyberte jeden z jazyků Java, C++ nebo C#.
- Snažte se použít co možná nejnovější syntaxi (a idiomy, atd.) vybraného jazyka.

Představte si, že vyvíjíte **knihovnu pro výpočty s maticemi**.

1. Definujte kontrakt (bez implementace) pro reprezentaci dvourozměrné matice. Tento kontrakt musí poskytovat *prostředky* (tj. metody, funkce, vlastnosti apod. podle zvoleného jazyka) alespoň pro:

- získání velikosti jednotlivých rozměrů,
- získání hodnoty z matice na základě souřadnic dané hodnoty,
- provádění aritmetických operací (sčítání s jinou maticí, odčítání, násobení, atd.).

2. Vytvořte implementaci svého kontraktu, která je vhodná pro velmi velké, ale **řídké** (tzv. *sparse*) matice (tj. matice obsahující převážně nuly). Okomentujte svou implementaci tak, aby bylo zřejmé, jak funguje. Z metod (funkcí, atd.) implementujte pouze ty, které vrací velikosti rozměrů a hodnotu z matice na základě souřadnic.

Definujte prostorovou složitost své implementace ve stylu $O(\text{something})$.

3. Představte si, že máte více implementací svého kontraktu pro matice (např. pro řídké matice a pro husté matice). Dále si představte, že máte metodu (nebo funkci, atd.), která dostane soubor s hodnotami matice a určí, která implementace je pro danou matici ideální.

S využitím vhodného *návrhového vzoru* implementujte *prostředek*, který dostane soubor a vrátí instanci kontraktu matice s ideální implementací.

Náčrt řešení.

1. Rozhraní nebo abstraktní třída s metodami podle otázky, přetíženými operátory (pokud jsou podporovány), vlastnostmi (pokud jsou podporovány) atd.
2. Existuje mnoho možných implementací (pro matici $M \times N$ určitě není správnou implementací dvourozměrné pole $M \times N$, které obsahuje všechny hodnoty). Správné řešení je uložit pouze nenulové hodnoty (plus nějaká „management”data). Možná řešení: (i) seznam obsahující trojice [řádek, sloupec, hodnota] se seřazením podle řádkového indexu a pak podle sloupcového indexu, (ii) seznam seznamů obsahující dvojice [sloupec, hodnota] a seřazený, (iii) Compressed Sparse Row formát (CSR), tj. tři pole, z nichž jedno drží nenulové hodnoty, druhé drží sloupec, kde se daná hodnota nachází, a třetí drží index (v hodnotách a sloupcích), kde daný řádek začíná, (iv) atd. Implementace metody `getValue` závisí na zvolené reprezentaci. Pro CSR to může být například:

```
public double getValue(int row, int col) {
    checkBounds(row, col);
    int start = rowPointers[row];
    int end = rowPointers[row + 1];
    for (int idx = start; idx < end; idx++) {
        if (colIndices[idx] == col) {
            return values[idx];
        }
    }
    return 0.0;
}
```

Složitost závisí na zvolené implementaci, ale měla by být podobná $O(\text{pocetNenulovychHodnot} + \text{neco})$.

3. Odkazuje to na implementaci na základě návrhového vzoru *Factory*, například:

```
public static Matrix create(Path path) {
    return switch (getSuitableType(path)) {
        case "sparse": new SparseMatrix(path);
        case "dense": new DenseMatrix(path);
        default throw new RuntimeException("unknown_implementation");
    }
}
```

Doplňující vysvětlení (nad rámec oficiálního náčrtu): úryvek v oficiálním náčrtu míchá dvě syntaxe `switch` a v Javě by se nepřeložil. Výraz `switch` (Java 14+) se šipkovými větvemi vypadá takto:

```
return switch (getSuitableType(path)) {
    case "sparse" -> new SparseMatrix(path);
    case "dense" -> new DenseMatrix(path);
    default -> throw new RuntimeException("unknown implementation");
};
```

Jde o *tovární metodu* (Factory): volající dostane instanci kontraktu `Matrix` a o konkrétní implementaci (řídka/hustá) rozhoduje jediné místo, které lze rozšířit o další reprezentace, aniž se změní kód volajících. Prostorová složitost CSR je $O(nnz + M)$ (nnz nenulových hodnot a sloupcových indexů, $M + 1$ ukazatelů na začátky řádků).

Lehčí v C# — rozhraní matice + přetížení operátorů

Více reprezentací (hustá, řádká/CSR) za jedním rozhraním je v C# čisté; navíc C# (na rozdíl od Javy) umí *přetížení operátorů*, takže `a + b` funguje jako v C++:

```
interface IMatrix {
    int Rows { get; } int Cols { get; }
    double this[int r, int c] { get; } // indexer místo getValue(r,c)
}
sealed class CsrMatrix : IMatrix {
    public int Rows { get; } public int Cols { get; }
    public double this[int r, int c] { get => /* prohledej radek r v colIndices */
        0.0; }
    public static CsrMatrix operator +(CsrMatrix a, CsrMatrix b) => /* ... */ a;
}
static IMatrix Create(string path) => GetSuitableType(path) switch { // tovární
    metoda
    "sparse" => new CsrMatrix(path),
    "dense" => new DenseMatrix(path),
    _ => throw new ArgumentException("unknown format"),
};
```