

Programovací jazyky (C++)

Studijní materiál ke státnicím (I3)

Jakub Klesa

Pokryté předměty: NPRG041 (Programování v C++), NPRG051 (Pokročilé programování v C++)

Obsah

Úvod	2
1 Abstrakce, zapouzdření, polymorfismus, dědičnost	2
1.1 Třídy, zapouzdření, viditelnost	3
1.2 Dědičnost, polymorfismus, virtuální metody	3
1.3 Jednoduchá vs. vícenásobná dědičnost	5
1.4 Vhodné použití: dědičnost vs. kompozice	5
2 Primitivní a objektové typy a jejich reprezentace	9
2.1 Číselné a výčtové typy	9
2.2 Hodnota, ukazatel, reference	10
2.3 Vlastník vs. pozorovatel, zásobník vs. halda	10
3 Generické typy, funkcionální prvky, lambda funkce	11
3.1 Šablony (generické typy)	11
3.2 Statický vs. dynamický polymorfismus	11
3.3 Typy reprezentující funkce, lambda, funkcionální rozhraní	12
4 Manipulace se zdroji a ošetření chyb (výjimky)	17
4.1 RAII: vázání zdroje na životnost objektu	17
4.2 Výjimky: vyhození, zachycení, propagace	17
4.3 Bezpečnost vůči výjimkám; proč destruktory nehází	18
5 Životní cyklus objektů a správa paměti (GC, reference counting)	18
5.1 Alokace a inicializace	19
5.2 Destrukce, kopírování, přesun: pravidlo pěti	19
5.3 Chytré ukazatele a počítání referencí	20
5.4 Správa paměti: C++ vs. spravovaná prostředí	20
6 Vlákna a synchronizace	21
6.1 Reprezentace vlákna a spuštění	21
6.2 Časově závislé chyby a kritická sekce	21
6.3 Synchronizační prostředky	22
7 Implementace OO jazyků (tabulka virtuálních metod)	22
7.1 Reprezentace primitivních a složených typů	22
7.2 Tabulka virtuálních metod (vtable)	22

8	Nativní a interpretovaný běh, překlad a sestavení	23
8.1	Reprezentace programu, oddělený překlad	23
8.2	Statické a dynamické knihovny	24
8.3	Nativní vs. interpretovaný běh, AOT vs. JIT	24
8.4	Běhové prostředí a vazba na OS	24
	Shrnutí	25
	Zdroje	27

Úvod

Tento materiál pokrývá okruh **I3 – Programovací jazyky** ve variantě se zvoleným jazykem **C++** (předměty NPRG041 *Programování v C++* a NPRG051 *Pokročilé programování v C++*, oba přednáší David Bednárek na MFF UK). Výklad stojí na přednáškových slidech těchto kurzů; tam, kde slidy mlčí nebo kde se látka týká jazykově obecných pojmů (garbage collector, bytecode, JIT), doplňujeme výklad kontrastem se spravovanými běhovými prostředími (JVM, .NET CLR) a ověřujeme jej standardní referencí (cppreference). Terminologii i hloubku držíme na úrovni bakalářského kurzu.

Role důkazů. Okruh je praktický, požadavky v něm *nejmenují žádné věty ani důkazy*. Jde o pojmy a konstrukce jazyka a o jejich použití: navrhnout objektový model (hierarchii tříd a rozhraní, volbu mezi dědičností a kompozicí), zapsat ho v syntaxi zvoleného jazyka, použít vhodný koncept (polymorfismus, šablony, lambdy, chytré ukazatele, RAII, návrhový vzor) a umět volbu zdůvodnit. Systémová a běhová témata (tabulka virtuálních metod, překlad a sestavení, vlákna, správa paměti) požadavky formulují na úrovni porozumění konceptu a jeho popisu.

Do výkladu je zařazeno všech šest reálných otázek SZZ k okruhu z let 2022–2025 jako řešené příklady (jedna z nich, komponentový systém z léta 2023, formálně patřila jiné specializaci); jsou i v samostatném dokumentu *Příklady otázek*. Použité zdroje jsou uvedeny na konci. **Zadání otázek dovolují odpovět v C++, C# nebo Javě.** Výklad je v C++, ale u témat, kde je odpověď v C# výrazně jednodušší – typicky když se daná C++ konstrukce v C# vůbec nevyskytuje nebo má přímočařejší protějšek – na to upozorníme ve fialovém rámečku *Lehčí v C#*.

Značení. Třídy a typy píšeme *TaktoStrojopisem*, klíčová slova jazyka *takto*. Ukazatel na typ *T* značíme T^* , referenci $T\&$, r-value referenci $T\&\&$; chytré ukazatele `std::unique_ptr<T>` a `std::shared_ptr<T>`. „Metodou“ myslíme členskou funkci, „datovou položkou“ nestatický datový člen. V obrázcích kreslíme abstraktní třídy a rozhraní červeně (kurzívou název), konkrétní třídy modře; dědičnost (vztah IS-A) prázdnou trojúhelníkovou šipkou od potomka k předkovi, ukazatele plnou šipkou. Standard, na který se odkazujeme, je C++20/C++23 (idiomy moderního C++).

1 Abstrakce, zapouzdření, polymorfismus, dědičnost

Objektové programování staví na čtyřech pojmech: *abstrakce* (popsat objekt tím, co umí, ne jak je udělán), *zapouzdření* (skrýt implementaci za rozhraní), *dědičnost* (odvodit specializovanější typ z obecnějšího) a *polymorfismus* (zacházet s různými typy jednotně přes společné rozhraní). C++ tyto pojmy podporuje, ale na rozdíl od Javy či C# k nim *nenutí*: programátor sám volí, kdy je objektový model namístě a kdy stačí prostá hodnota nebo šablona.

1.1 Třídy, zapouzdření, viditelnost

Definice – Třída, datová položka, metoda

Třída je uživatelsky definovaný typ sdružující *datové položky* (stav objektu) a *metody* (operace nad stavem). *Objekt* je instance třídy. V C++ se třída deklaruje klíčovým slovem `class` nebo `struct`; jediný rozdíl je *implicitní viditelnost* členů: u `struct` jsou implicitně veřejné, u `class` privátní.

Definice – Viditelnost (zapouzdření)

Každý člen třídy má jednu ze tří úrovní viditelnosti:

- `public` – přístupný odkudkoli (tvoří *rozhraní* třídy),
- `protected` – přístupný jen ze třídy samé a z odvozených tříd,
- `private` – přístupný jen ze třídy samé (a jejích *friend*).

Zapouzdření znamená skrýt datové položky a pomocné metody jako `private` a ven vystavit jen veřejné rozhraní. Tím se implementace dá měnit, aniž se dotkne kódu, který třídu používá.

Intuice. Abstrakce a zapouzdření jsou dvě strany téže mince: abstrakce říká *co* objekt nabízí (veřejné metody), zapouzdření skrývá *jak* to dělá (privátní stav). Veřejné rozhraní je „smlouva“, na kterou se smí ostatní kód spoléhat; vše ostatní je volně měnitelné.

Poznámka. C++ rozlišuje *logickou* a *bitovou* konstantnost: metoda označená `const` (např. `double at(int i) const`) slibuje, že nezmění pozorovatelný stav objektu, a smí být volána na konstantních objektech. Položka označená `mutable` smí být změněna i z `const` metody (typicky cache).

1.2 Dědičnost, polymorfismus, virtuální metody

Definice – Dědičnost, vztah IS-A

Odvozená třída (potomek) vzniká z *báze* (předka) zápisem `class Derived : public Base { ... };`. Potomek přebírá členy předka a může přidat vlastní. Veřejná dědičnost vyjadřuje vztah IS-A: „každý `Derived` je zároveň `Base`“, takže ukazatel/referenci na potomka lze použít všude, kde se čeká předek (konverze ve směru *potomek* → *předek*).

Definice – Statické a dynamické typování, dynamický polymorfismus

C++ je *staticky typovaný* jazyk: typ každého výrazu je znám už při překladu a překladač jím kontroluje korektnost. U objektu za ukazatelem/referencí rozlišujeme *statický typ* (typ proměnné, např. `Base&`) a *dynamický typ* (skutečný typ objektu, např. `Derived`). (*Dynamický*) *polymorfismus* je schopnost zavolat podle dynamického typu tu „správnou“ verzi metody, i když k objektu přistupujeme přes statický typ předka.

Definice – Virtuální vs. nevirtuální metoda

Metoda označená `virtual` se při volání přes ukazatel nebo referenci vybírá podle *dynamického* typu objektu (pozdní vazba). Nevirtuální metoda se vybírá podle *statického* typu (časná vazba, rozhodne se při překladu). Potomek virtuální metodu *překrývá* (override); zápisem `override` necháme překladač ověřit, že v předkovi opravdu existuje virtuální metoda téže signatury, `final` zakáže další překrytí.

Intuice. Mechanismus virtuálních metod se uplatní **jen přes ukazatel nebo referenci** na bázi. Voláme-li metodu na objektu „přes hodnotu“ (`Base b = d; b.f();`), zavolá se vždy verze z báze, protože proměnná `b` je typu `Base`. Toto je zdroj jevu zvaného *slicing* (viz níže).

Definice — Abstraktní třída, rozhraní, čistě virtuální metoda

Čistě virtuální metoda se zapisuje `virtual void f() = 0;` a obvykle nemá tělo (mít ho ale může – viz poznámka). Třída s aspoň jednou čistě virtuální metodou je *abstraktní* a nelze ji samostatně instanciovat – slouží jako *rozhraní* (interface), které musí potomci implementovat. V C++ se rozhraní modeluje právě abstraktní třídou složenou z čistě virtuálních metod (jazyk nemá samostatné klíčové slovo interface).

i Poznámka. Zápis `= 0` dělá metodu *čistě virtuální* (a tím třídu abstraktní); je to nezávislé na tom, zda má metoda tělo. Tělo *mít může* – definuje se ovšem **mimo třídu** (samostatnou definicí `void Base::f()`, ne tělem přímo u deklarace `= 0`). Zavolat ho lze jen *kvalifikovaně* (nevirtuálně), např. `p->Base::f()` nebo z potomka jako `Base::f()`; (sdílená výchozí část implementace, kterou si override „dovolá“); přes virtuální volání `p->f()` se na něj nedostaneme a třída zůstává abstraktní. Klasický **povinný** případ: *čistě virtuální destruktork* `virtual ~Base() = 0;` tělo mít **musí**, protože destruktory potomků ho vždy volají (jinak chybí linkeru).

Lehčí v C# — rozhraní

C# má pro rozhraní *samostatné klíčové slovo* `interface`, takže ho nemusíte modelovat abstraktní třídou s čistě virtuálními metodami ani řešit virtuální destruktork (o uvolnění objektů se stará GC, sekce 5). Třída rozhraní jen „implementuje“:

```
interface IFilter { void Apply(Bitmap b); }           // jen kontrakt, bez těla
class Sharpness : IFilter {
    public void Apply(Bitmap b) { /* ... */ }         // implementace
}
```

Členové rozhraní jsou implicitně veřejní a abstraktní; implementující třída je naplní. Hodí se to ve všech návrhových otázkách okruhu (bitmapový editor, knihovna matic, komponenty, reprezentace typů), které stojí na rozhraních.

i Poznámka. [Slicing] Přiřazení objektu potomka do proměnné typu báze (`Base b = derived;`) zkopíruje jen „bázovou část“ a potomkovskou odřízne – tomu se říká *slicing*. Je to specifikum C++ plynoucí z hodnotové sémantiky; proto se polymorfní objekty drží *přes ukazatele* (`std::vector<std::unique_ptr<Base>>`), nikdy přes hodnotu (`std::vector<Base>`).

i Poznámka. [Virtuální destruktork] Má-li se objekt potomka rušit přes ukazatel na bázi (`Base* p = new Derived;` `delete p;`), musí mít báze **virtuální destruktork** (`virtual ~Base() = default;`); jinak se zavolá jen destruktork báze a vznikne nedefinované chování. Pravidlo: *každá polymorfní báze má mít virtuální destruktork*.

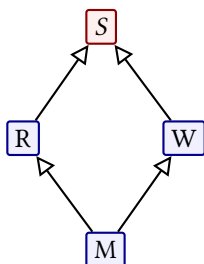
```
struct Shape {                                     // rozhraní (abstraktní třída)
    virtual ~Shape() = default;                     // polymorfní báze -> virtuální destruktork
    virtual double area() const = 0;                // čistě virtuální -> Shape je abstraktní
};
class Circle : public Shape {                       // Circle IS-A Shape
    double r_;
public:
    explicit Circle(double r) : r_(r) {}
    double area() const override { return 3.14159265 * r_ * r_; }
};
// použití přes bázi -> dynamický polymorfismus:
std::unique_ptr<Shape> s = std::make_unique<Circle>(2.0);
double a = s->area();                               // volá Circle::area (dynamický typ)
```

1.3 Jednoduchá vs. vícenásobná dědičnost

Jednoduchá dědičnost (jedna báze) modeluje hierarchii typů IS-A. *Vícenásobná* dědičnost (více bází) je v C++ povolena, ale přináší *diamantový problém*: dědí-li třída M ze dvou tříd R, W, které obě dědí ze společné báze S, vzniknou v objektu M *dvě kopie* podobjektu S a odkaz na S je nejednoznačný.

Definice – Virtuální dědičnost

Virtuální dědičnost (`class R : public virtual S`) zajistí, že všechny cesty k společné bázi sdílejí **jedinou** instanci podobjektu S. Cenou je nepřímý přístup k virtuální bázi (přes uložený offset, zjišťovaný za běhu). Používá se právě pro sdílená rozhraní.



(a) nevirtuální: objekt M obsahuje **dvě** kopie S



(b) virtuální: jen **jedna** sdílená S

Diamantová hierarchie `M : R, W` a `R, W : S`. Bez virtuální dědičnosti (a) má objekt M dvě nezávislé kopie S (přístup `m.S : x` je nejednoznačný). S virtuální dědičností (b) všechny cesty sdílejí jedinou S uloženou na konci rozložení objektu.

Lehčí v C# – bez vícenásobné dědičnosti

C# vícenásobnou dědičnost *tříd* nemá: každá třída má nejvýš **jednu** bazovou třídu. Celý diamantový problém (dvě kopie báze, virtuální dědičnost, dopočítávané offsety) v něm proto **nevzniká** a nemusíte ho vysvětlovat. „Více předků“ se modeluje jako jedna bazová třída plus libovolný počet *rozhraní* (viz výše): `class M : Base, I1, I2`. Rozhraní nenesou instanční data, takže se stav nemá jak zdvojit.

Pozn.: od C# 8 mohou mít rozhraní *výchozí implementaci* metod (DIM); i tak ale nenesou žádný instanční stav (duplicita stavu nehrozí) a případnou kolizi dvou stejně pojmenovaných metod řeší *explicitní implementace*.

1.4 Vhodné použití: dědičnost vs. kompozice

 **Intuice.** Dědičnost v C++ je silný, ale často zneužívaný nástroj. Bednárek uvádí jen dvě „správná“ použití: (1) **IS-A hierarchie** entit s vlastní identitou (jednoduchá nevirtuální veřejná dědičnost) a (2) **vztah rozhraní–implementace** (potomek implementuje abstraktní rozhraní; typicky vícenásobná virtuální veřejná dědičnost, rozhraní bez datových položek). Pro *sdílení/znovupoužití kódu* dědičnost *není* správný nástroj – na to se hodí **kompozice** (mít cizí objekt jako datovou položku).

 **Poznámka.** [Dva světy tříd] Kurz odlišuje *hodnotové typy* (bez dědičnosti a virtuálních metod – chovají se jako čísla, kopírují a přesouvají se (pravidlo pěti, sekce 5), ukládají se přímo, mají smysluplné operátory; např. `std::string`, `matice`) od *entitních typů s identitou* (dědičnost, virtuální metody, individuální dynamická alokace, drží se přes ukazatele na společnou bázi, nutný virtuální destruktork). Polymorfni kontejner je vždy `std::vector<std::unique_ptr<Base>>`, nikdy `std::vector<Base>` (slicing).

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

Přepracujte objektový návrh aplikace pro úpravu bitmapových obrázků. Aplikace umí mít otevřeno více obrázků; formát se pozná podle přípony souboru; na obrázek se postupně vybírají filtry, které se nakonec dávkově aplikují; obrázek lze uložit ve stejném či jiném formátu. *Nové filtry* se budou doplňovat přímo do kódu aplikace, kdežto *nové formáty* se budou přidávat jako *pluginy* (dynamicky linkované knihovny) načítané za běhu – proces načítání obrázku tedy musí být dostatečně flexibilní, aby uměl za běhu přijmout odkaz na nový formát. Navrhněte typy a jejich vztahy v syntaxi zvoleného jazyka a upravte Main. (Původní kostra je v C#; my odpovídáme v C++.)

Řešení.

Původní návrh měl chyby: Image mísil *data obrázku s načítáním/ukládáním formátu*, filtry byly statické funkce (nešlo je dávkovat ani snadno přidávat) a volba formátu byla natvrdo zadrátovaná v ImageLoader (nešel přidat plugin). Oddělíme proto tři odpovědnosti:

1. **Data obrázku** jako hodnotový typ Bitmap (rozměry + pixely; rozměry filtr nemění).
2. **Formát** jako *rozhraní* IImageFormat (Load/Save); každý formát (PNG, JPEG, plugin) je jeho implementace. Registr formátů (*Factory* podle přípony) lze za běhu rozšířit – sem se zaregistruje plugin.
3. **Filtr** jako rozhraní IFilter (Apply); konkrétní filtry nesou svá nastavení. Document drží seznam vybraných filtrů a umí je dávkově aplikovat.

```
struct Color { uint8_t r, g, b; };

class Bitmap {                                // hodnotový typ: jen data obrázku
    int w_, h_;
    std::vector<Color> px_;                    // w_*h_ pixelů
public:
    int width() const { return w_; }
    int height() const { return h_; }
    Color& at(int x, int y) { return px_[y * w_ + x]; }
};

class IImageFormat {                          // rozhraní formátu (implementuje i plugin)
public:
    virtual ~IImageFormat() = default;
    virtual Bitmap Load(const std::string& file) = 0;
    virtual void Save(const std::string& file, const Bitmap&) = 0;
};

class PngFormat : public IImageFormat { /* Load/Save pro PNG */ };
class JpegFormat : public IImageFormat { /* Load/Save pro JPEG */ };

// Registr formátů = Factory podle přípony; pluginy se sem zaregistrují za běhu.
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);
    }
    std::unique_ptr<IImageFormat> Create(const std::string& ext) const {
        auto it = reg_.find(ext);
        if (it == reg_.end()) throw std::runtime_error("neznámý formát");
        return it->second();           // vyrobí instanci formátu
    }
};

class IFilter {                               // rozhraní filtru
public:
```

```

    virtual ~IFilter() = default;
    virtual void Apply(Bitmap&) = 0;
};

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

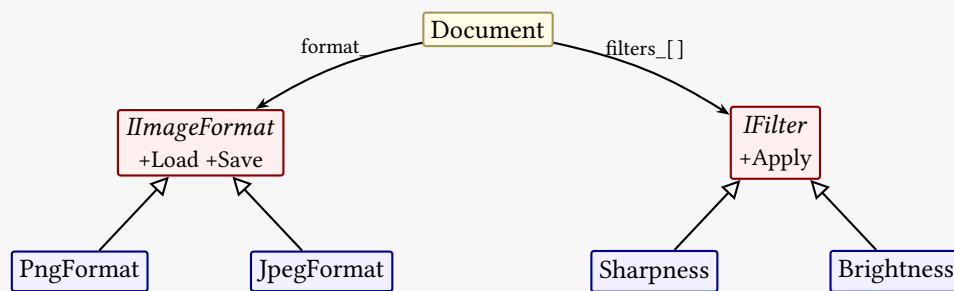
class Brightness : public IFilter {
    float amount_;
public:
    explicit Brightness(float a) : amount_(a) {}
    void Apply(Bitmap&) override { /* ... */ }
};

class Document {
    Bitmap data_;
    std::unique_ptr<IImageFormat> format_; // jak se ukládá
    std::vector<std::unique_ptr<IFilter>> filters_;
public:
    void AddFilter(std::unique_ptr<IFilter> f) { filters_.push_back(std::move(f)); }
    void ApplyAll() { for (auto& f : filters_) f->Apply(data_); filters_.clear(); }
    void Save(const std::string& file) { format_->Save(file, data_); } // v původním formátu
    // uložení v JINÉM formátu: formát se vybere podle cílové přípony (symetricky k Open)
    void SaveAs(const FormatRegistry& reg, const std::string& file) {
        reg.Create(file.substr(file.rfind('.')).->Save(file, data_);
    }
    // tovární metoda: od jména souboru k načtenému obrázku
    static Document Open(const FormatRegistry& reg, const std::string& file) {
        std::string ext = file.substr(file.rfind('.')); // přípona
        auto fmt = reg.Create(ext); // vybere formát
        Document d;
        d.data_ = fmt->Load(file);
        d.format_ = std::move(fmt);
        return d;
    }
};

int main() {
    FormatRegistry reg;
    reg.Register(".png", [] { return std::make_unique<PngFormat>(); });
    reg.Register(".jpg", [] { return std::make_unique<JpegFormat>(); });
    // (plugin by za běhu zavolal reg.Register pro svůj nový formát)
    Document doc = Document::Open(reg, "a.jpg");
    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 původním formátu
    doc.SaveAs(reg, "b.png"); // nebo v jiném podporovaném formátu (PNG)
}

```

Proč je návrh dobrý. 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` – a ten řádek může zavolat plugin za běhu, takže registr je flexibilní právě v požadovaném místě. Uložení v *jiném* formátu obstará `SaveAs`, jež vybere formát podle cílové přípony přes též registr (`Save` ukládá v původním). `Bitmap` je hodnotový typ, `IFilter/IImageFormat` jsou entitní (drženy přes `unique_ptr`); RAII (sekce 4) zajistí korektní úklid.



Přepracovaný návrh: Document drží jeden IImageFormat (jak ukládat) a seznam IFilter (co aplikovat). Obě jsou abstraktní rozhraní (červeně), konkrétní formáty i filtry z nich dědí (modře). Přidání filtru = nová třída pod IFilter; přidání formátu = nová třída pod IImageFormat plus registrace v FormatRegistry.

Příklad — SZZ jaro 2025 č. 2: Knihovna pro výpočty s maticemi

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

Zadání.

1. Definujte *kontrakt* (bez implementace) pro 2D matici: získání rozměrů, hodnoty na souřadnicích, aritmetické operace.
2. Vytvořte implementaci vhodnou pro velmi velké *řídke* (sparse) matice; implementujte jen rozměry a získání hodnoty a uveďte prostorovou složitost $O(\cdot)$.
3. S využitím vhodného návrhového vzoru implementujte prostředek, který dostane soubor a vrátí instanci kontraktu s ideální implementací.

Řešení.

(1) **Kontrakt** = abstraktní třída (rozhraní) s čistě virtuálními metodami; aritmetiku necháme vracet novou matici přes `unique_ptr<Matrix>` (polymorfní výsledek):

```

class Matrix {
public:
    virtual ~Matrix() = default;
    virtual int rows() const = 0;
    virtual int cols() const = 0;
    virtual double at(int r, int c) const = 0; // hodnota na souřadnicích
    virtual std::unique_ptr<Matrix> add(const Matrix&) const = 0;
    virtual std::unique_ptr<Matrix> mul(const Matrix&) const = 0;
};
  
```

(2) **Řídká implementace – formát CSR** (Compressed Sparse Row): tři pole – `vals_` (nenulové hodnoty po řádcích), `cols_` (jejich sloupce) a `rowPtr_` (index, kde v polích začíná každý řádek). Uloží se jen *nnz* nenulových hodnot, ne $r \cdot c$ nul. at prohledá příslušný úsek řádku:

```

class SparseMatrix : public Matrix {
    int r_, c_;
    std::vector<double> vals_; // nenulové hodnoty, seřazené po řádcích
    std::vector<int> cols_; // sloupcový index každé hodnoty
    std::vector<int> rowPtr_; // rowPtr_[i].. rowPtr_[i+1] = úsek řádku i
public:
    int rows() const override { return r_; }
    int cols() const override { return c_; }
    double at(int r, int c) const override {
        for (int k = rowPtr_[r]; k < rowPtr_[r + 1]; ++k) // jen nenuly v řádku r
            if (cols_[k] == c) return vals_[k];
        return 0.0; // jinak je tam nula
    }
    std::unique_ptr<Matrix> add(const Matrix&) const override { /* ... */ return {}; }
    std::unique_ptr<Matrix> mul(const Matrix&) const override { /* ... */ return {}; }
};
  
```

Paměť drží jen nenulové prvky a ukazatele na řádky, takže **prostorová složitost je** $O(nnz + r)$, kde nnz je počet nenulových hodnot a r počet řádků (pole `rowPtr_` má délku $r + 1$). To je proti hustému $O(r \cdot c)$ obrovská úspora, je-li většina prvků nula.

(3) Factory podle obsahu souboru. Prostředek načte soubor, odhadne hustotu a vrátí ideální implementaci za rozhraním `Matrix` (tovární metoda vracející `unique_ptr<Matrix>`; konstruktory `SparseMatrix/DenseMatrix` z cesty k souboru zde nevypisujeme):

```
std::unique_ptr<Matrix> createMatrix(const std::string& path) {  
    double density = estimateDensity(path); // podíl nenulových prvků  
    if (density < 0.1) return std::make_unique<SparseMatrix>(path);  
    else return std::make_unique<DenseMatrix>(path);  
}
```

Volající pracuje jen s rozhraním `Matrix`, konkrétní implementace je skrytá – to je smysl návrhového vzoru *Factory*.

i Poznámka. Náčrt v PDF zmiňuje i variantu s *přetíženými operátory* (`operator+`, `operator*`) místo metod `add/mul` – v C++ idiomatické pro hodnotové matice. U polymorfního kontraktu drženého přes ukazatel je ale obvyklejší pojmenovaná virtuální metoda; operátory by se daly přidat jako nevirtuální obal volající `add/mul`.

2 Primitivní a objektové typy a jejich reprezentace

C++ má dvě velké skupiny typů: *primitivní* (vestavěné: čísla, znaky, `bool`, výčty, ukazatele) a *složené/objektové* (třídy, struktury, pole, kontejnery). Liší se tím, jak se s nimi zachází a jak vypadají v paměti.

2.1 Číselné a výčtové typy

☰ Definice – Číselné typy

Celá čísla: `int`, `short`, `long`, `long long` a jejich `unsigned` varianty; znaménková se ukládají ve *dvojkovém doplňku*. Čísla *s plovoucí řádovou čárkou:* `float` (32 bitů), `double` (64 bitů), na běžných platformách podle normy IEEE 754 (standard C++ ji nevyžaduje). Velikost typu v bajtech vrací operátor `sizeof`. Pevné velikosti zaručují typy z `<cstdint>` (`int32_t`, `uint8_t`, ...).

i Poznámka. Přesné bitové reprezentace (dvojkový doplněk, IEEE 754, zarovnání) jsou doménou okruhu **I4**; zde stačí vědět, že primitivní typy mají *pevnou velikost danou platformou* a ukládají se přímo (žádná dynamická alokace, žádná hlavička objektu).

☰ Definice – Výčtový typ

Výčet je typ s konečnou množinou pojmenovaných hodnot. Moderní C++ preferuje *silně typovaný* výčet `enum class`:

```
enum class Color { Red, Green, Blue }; // silně typovaný (scoped)  
Color c = Color::Red; // nutná kvalifikace Color::  
int i = static_cast<int>(c); // konverze jen explicitně  
enum class Status : uint8_t { Ok, Error }; // lze zvolit podkladový typ
```

💡 Intuice. Starší prostý `enum` se implicitně převádí na `int` a jeho názvy „vytékají“ do okolního jmenného prostoru (riziko kolizí). `enum class` obojí zakazuje: hodnoty jsou v jmenném prostoru výčtu (`Color::Red`) a konverze na číslo je jen explicitní. Proto je v moderním C++ výchozí volbou.

2.2 Hodnota, ukazatel, reference

C++ pojmem *odkaz* zastřešuje několik prostředků, jak se dostat k objektu jinde v paměti. Volba odkazu **signalizuje úmysl** programátora (kdo vlastní, kdo jen čte, jak dlouho).

Definice – Hodnota, ukazatel, reference

- *Hodnota* (τ x ;) – proměnná je objektem; přístup přes `.`; přiřazení *kopíruje* obsah.
- *Ukazatel* (τ^* p ;) – drží *adresu* objektu; přístup přes `*p` a `p->m`; lze ho přesměrovat i nastavit na `nullptr`; umožňuje ukazatelovou aritmetiku v polích.
- *Reference* ($\tau\&$ $r = x$;) – *alias* existujícího objektu; musí být inicializována, **nelze ji přesměrovat**; přístup přes `.` jako u hodnoty.

Definice – const reference, r-value reference

Konstantní reference `const` $\tau\&$ váže libovolný výraz (i dočasný) a nedovolí jej měnit – typický *vstupní parametr* předávaný bez kopie. *R-value reference* $\tau\&\&$ váže jen *dočasné* (r-value) objekty, „jejichž majitel se o ně už nezajímá“; je základem *move sémantiky* (přesun obsahu místo kopie, viz sekce 5).

Intuice. V Javě a C# se k objektům tříd (tzv. referenčním typům) přistupuje vždy přes referenci; C++ naopak u každého typu dává na výběr mezi hodnotou, ukazatelem a referencí. `const` $\tau\&$ jako parametr znamená „jen se podívám, nezdržuj kopírováním“; $\tau\&$ „budu měnit tvůj objekt“; τ^* „možná žádný objekt není (`nullptr`)“; $\tau\&\&$ „tenhle dočasný objekt si rozeberu“.

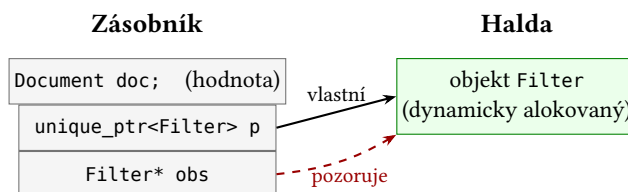
2.3 Vlastník vs. pozorovatel, zásobník vs. halda

Klíčový myšlenkový model kurzu: každý odkaz je buď *vlastník* (owner), který je zodpovědný za zrušení objektu, nebo *pozorovatel* (observer), který objekt jen používá a nesmí ho rušit.

Definice – Vlastník a pozorovatel

Vlastník drží objekt a uvolní ho, až přestane být potřeba: `std::unique_ptr<T>` (výhradní vlastník), `std::shared_ptr<T>` (sdílený vlastník). *Pozorovatel* jen odkazuje, nevlastní: holý τ^* (měnič), `const` τ^* (jen čtoucí) nebo reference. V moderním C++ **holý ukazatel nikdy nereprezentuje vlastnictví**.

Poznámka. [Kde objekt žije] *Statická* doba života: globální a `static` proměnné (žijí po celý běh). *Automatická* (zásobník): lokální proměnné, rušené automaticky na konci bloku – rychlé, ale životnost vázaná na blok. *Dynamická* (halda): objekty vytvořené `new` / `make_unique` / `make_shared`, žijí, dokud je někdo neuvolní; pomalejší (cache), nutné jen tam, kde životnost nesouvisí s voláním funkcí nebo kde je objekt polymorfní.



Zásobník drží malé „úchyty“ s automatickou dobou života; samotné objekty leží na haldě. `unique_ptr` `p` haldový objekt *vlastní* (zruší ho ve svém destrukturu, plná šipka). Holý ukazatel `obs` též objekt jen *pozoruje* (přerušovaná šipka) – nevlastní ho a nesmí ho rušit; musí přestat pozorovat dřív, než vlastník objekt zruší. Hodnotový objekt `doc` žije přímo na zásobníku.

3 Generické typy, funkcionální prvky, lambda funkce

Generické programování umožňuje napsat kód jednou a použít ho pro mnoho typů. C++ k tomu má *šablony* (templates), které pracují *za překladač* – to je základ *statického polymorfismu*. Funkcionální prvky (ukazatele na funkce, funktory, lambdy, `std::function`) umožňují předávat „chování“ jako hodnotu.

3.1 Šablony (generické typy)

Definice – Šablona

Šablona je generická deklarace s *formálními parametry vyhodnocenými za překladač*. Parametrem může být typ (`template<typename T>`), celočíselná konstanta (`template<std::size_t N>`) nebo jiná šablona. Rozlišujeme šablonu funkce a šablonu třídy:

```
template <typename T>                // šablona funkce
T max(T a, T b) { return a < b ? b : a; }

template <typename T, std::size_t N> // šablona třídy
class array { T data_[N]; /* ... */ };

int m = max(3, 7);                    // instanciaci: T = int (odvozeno z argumentů)
double d = max<double>(1, 2);         // T zadáno explicitně
array<int, 10> a;                     // instanciaci třídy: T = int, N = 10
```

Definice – Instanciaci, specializace

Instanciaci = použití šablony s konkrétními parametry; teprve tehdy překladač vygeneruje kód. U funkcí se typ často *odvodí* z argumentů, u tříd se zadává explicitně (od C++17 i u tříd lze odvodit, CTAD). *Specializace* dává pro konkrétní parametry jinou definici (úplná `template<> struct hash<MyType>`, nebo částečná – ta jen u šablon tříd, ne u funkcí, např. `template<class A> class vector<bool, A>`).

i Poznámka. Definice šablony musí být **viditelná v místě instanciaci**, proto šablony bývají celé v hlavičkových souborech (nelze je oddělit do `.cpp` jako běžné funkce). Šablony nemají *žádnou běhovou režii* – vygenerovaný kód je stejný, jako by byl napsán ručně pro daný typ; cenou je možné nafouknutí velikosti kódu (každá instanciaci je samostatná).

3.2 Statický vs. dynamický polymorfismus

Polymorfismus je schopnost přizpůsobit chování kódu. C++ ho má ve dvou podobách, které je dobré umět porovnat:

	Statický (šablony)	Dynamický (virtuální)
Kdy se rozhodne	při překladač (instanciaci)	za běhu (vtable)
Společná báze	netřeba (<i>duck typing</i>)	nutná abstraktní třída
Signatura	libovolná kompatibilní	přesně daná bázi
Volání	přímé, lze <i>inlinovat</i>	nepřímé přes ukazatel
Rychlost	rychlejší	pomalejší
Velikost kódu	větší (více instanciaci)	menší (jeden kód)
Přepínání za běhu	ne	ano

 **Intuice.** *Duck typing* (řádek „společná báze“ v tabulce): šabloně stačí, že typ má použité operace (metody/operátory správného jména a typu), nemusí dědit ze společné báze – „když to kváká jako kachna, ber to jako kachnu“. Statický polymorfismus je tím volnější než dynamický, který společnou abstraktní bázi vyžaduje. Doporučení kurzu: **preferuj statický polymorfismus, dynamický jen když je nutný** – tedy když se chování musí volit až za běhu (polymorfni kontejnery různých typů, GUI prvky, zásuvné moduly). Statický je „zadarmo“ (inlinuje se), ale neumí za běhu přepínat; dynamický to umí, ale platí nepřímým voláním přes tabulku virtuálních metod (sekce 7).

3.3 Typy reprezentující funkce, lambda, funkcionální rozhraní

Často chceme předat *chování* (porovnávací funkci, operaci na prvku) jako argument. C++ k tomu má tři prostředky:

Definice – Ukazatel na funkci, funktor, `std::function`

- Ukazatel na funkci `void(*) (int)` – nejstarší, beze stavu, pevná signatura.
- Funktor (funkční objekt) – třída s přetíženým `operator()`; může nést *stav* v datových položkách (uzávěr: tělo + nabalená data).
- `std::function<void(int)>` – typové úložiště pro libovolné volatelné s danou signaturou; dražší (případná dynamická alokace, nepřímé volání), použít, až když je potřeba míchat/ukládat různá volatelná.

Definice – Lambda funkce

Lambda je výraz `[capture](params){ telo }`, který překladač přeloží na *anonymní funktor*. Seznam *capture* určuje, které okolní proměnné lambda zachytí a jak: `[x]` hodnotou (kopie), `[&x]` odkazem, `[=]`/`[&]` vše *použité* hodnotou/odkazem, `[this]` přístup k objektu.

```
std::vector<int> v = {3, 1, 2};
int prah = 2;
// lambda zachytí proměnnou prah hodnotou; std::count_if čeká volatelné:
auto kolik = std::count_if(v.begin(), v.end(),
    [prah](int x){ return x > prah; });
// lambda = anonymní funktor; uloží se do proměnné typu, který zná jen překladač:
auto add = [](int a, int b){ return a + b; };
// pro uložení/míchání různých lambd se stejnou signaturou slouží std::function:
std::function<int(int,int)> op = add;
```

 **Intuice.** Každá lambda má *vlastní anonymní typ*, takže dvě lambdy téže signatury nelze přímo zaměnit ani uložit do jedné proměnné – na to je právě `std::function`. Funkce přijímající lambda proto bývají *šablony* (`template<typename F> void g(F f)`): chování se předá v *typu*, vyřeší se za překladač a inlinuje (statický polymorfismus). *Funkcionálním rozhraním* se v C++ míní právě „cokoli volatelné s danou signaturou“ – ukazatel, funktor, lambda i `std::function`.

Příklad – SZZ podzim 2022 č. 1: Objektové rozhraní pro grafy

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

Zadání.

Navrhněte (jen deklarujte) generické objektové rozhraní pro neorientované grafy $G = (V, E)$ (hrany spojují právě dva vrcholy, mezi dvěma vrcholy nejvýš jedna hrana), aby je mohly implementovat různé reprezentace (seznam sousedů, matice sousednosti, ...) a algoritmy se psaly jen jednou. Rozhraní pokrývá 3 entity: *vrchol* (přístup k incidentním hranám), *hrana* (její dva vrcholy) a *graf* (seznam všech vrcholů a hran, pouze pro čtení). Každý vrchol i hrana nese *tag* (např. příznak navštívení

u vrcholu, váha u hrany); typy tagů volí uživatel, ale v rámci jedné instance mají všechny vrcholy stejný typ tagu a všechny hrany taktéž (rozhraní je tedy generické). Dále napište tělo funkce, která pro daný graf a reálné číslo r rozloží graf na komponenty souvislosti, přičemž hrany s tagem (délkou) větším než r ignoruje, a index komponenty každého vrcholu uloží do jeho tagu.

Řešení.

(1) Tři generická rozhraní. Parametry VTag/ETag nesou typy tagů; v C++ jsou to abstraktní třídy-šablony (čistě virtuální metody), takže algoritmus je nezávislý na vnitřní reprezentaci grafu:

```
template <class VTag, class ETag> struct IEdge;           // dopředná deklarace

template <class VTag, class ETag>
struct IVertex {                                         // vrchol
    virtual ~IVertex() = default;
    virtual std::vector<IEdge<VTag,ETag>*> incidentEdges() = 0; // incidentní hrany
    virtual VTag getTag() const = 0;
    virtual void setTag(VTag) = 0;
};

template <class VTag, class ETag>
struct IEdge {                                           // hrana
    virtual ~IEdge() = default;
    virtual IVertex<VTag,ETag>* endpoint1() = 0;         // jeden koncový vrchol
    virtual IVertex<VTag,ETag>* endpoint2() = 0;         // druhý koncový vrchol
    virtual ETag getTag() const = 0;
    virtual void setTag(ETag) = 0;
};

template <class VTag, class ETag>
struct IGraph {                                         // graf je pouze pro čtení
    virtual ~IGraph() = default;
    virtual std::vector<IVertex<VTag,ETag>*> vertices() = 0;
    virtual std::vector<IEdge<VTag,ETag>*> edges() = 0;
};
```

(2) Komponenty souvislosti. BFS z každého dosud neoznačeného vrcholu; šíříme se jen po hranách s tagem $\leq r$, ostatní (delší) ignorujeme. Vrcholům dosaženým v jednom průchodu nastavíme stejný index, čítač pak zvýšíme. Tagy vrcholů jsou `int` (index komponenty, -1 = neoznačeno), tagy hran `float` (délka):

```
void connectedComponents(IGraph<int,float>& g, float r) {
    for (auto* v : g.vertices()) v->setTag(-1);          // -1 = neoznačeno
    int component = 0;
    for (auto* start : g.vertices()) {
        if (start->getTag() != -1) continue;             // již přiřazen
        std::queue<IVertex<int,float>*> q;               // BFS z vrcholu start
        start->setTag(component); q.push(start);
        while (!q.empty()) {
            auto* u = q.front(); q.pop();
            for (auto* e : u->incidentEdges()) {
                if (e->getTag() > r) continue;           // příliš dlouhá hrana, ignoruj
                auto* w = (e->endpoint1() == u) ? e->endpoint2() : e->endpoint1();
                if (w->getTag() == -1) { w->setTag(component); q.push(w); }
            }
        }
        component++;                                     // další komponenta
    }
}
```

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



(Reálná otázka SZZ: 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.

Navrhněte komponentový herní framework. Každý objekt scény je `GameObject` (GO), což je množina *komponent*. Od každého *druhu* komponenty má GO nejvýš jednu instanci. Potřeba je: (1) metoda pro připojení komponenty (stejný druh nahradí předchozí) a pro získání komponenty zvoleného druhu (s indikací, že chybí); metoda `Update(timeDelta)` zavolá `Update` na těch komponentách, které ji mají (autor komponenty se rozhodne, zda ji má). (2) Komponenty `Position` a `Health`. (3) Efekty otravy a krvácení (vyžadují `Health`), parametrizované úbytkem zdraví za sekundu; funkce, která zvolený efekt aplikuje na seznam GO.

Řešení.

(1) Identifikace druhu komponenty. Druh komponenty reprezentujeme jejím *typem* a klíčujeme podle `std::type_index` (běhový identifikátor typu). `GameObject` drží mapu z typu na komponentu. Generická metoda `Get<T>` vrátí ukazatel na komponentu typu `T`, nebo `nullptr`, když chybí – to je ono „rozumné indikování“. Možnost mít/nemít `Update` řešíme volitelným rozhraním `IUpdatable`.

```
struct Component { virtual ~Component() = default; }; // báze všech komponent
struct IUpdatable { // volitelné rozhraní
    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> // připojení (nahradí stejný druh)
    void Add(std::unique_ptr<T> c) { comps_[std::type_index(typeid(T))] = std::move(c); }

    template <typename T> // získání, nebo nullptr když chybí
    T* Get() {
        auto it = comps_.find(std::type_index(typeid(T)));
        return it == comps_.end() ? nullptr : static_cast<T*>(it->second.get());
    }

    void Update(float dt) { // zavolá Update na komponentách, co ji mají
        for (auto& [type, c] : comps_)
            if (auto* u = dynamic_cast<IUpdatable*>(c.get())) u->Update(dt);
    }
};
```

(2) Konkrétní komponenty.

```
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_); } // hlídá 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 na svém GO (potřebuje zpětný odkaz na vlastníka) a ubírá zdraví. Otrava i krvácení sdílejí logiku, liší se jen typem: protože GameObject klíčuje komponenty podle typu, musejí to být *dva různé typy* (jinak by druhý efekt přepsal první), aby mohly být na GO obě současně. Volbu „otrava vs. krvácení“ ve funkci řešíme *šablonovým* parametrem typu efektu:

```
class Effect : public Component, public IUpdatable {
protected:
    GameObject* owner_; float perSec_; // pozorovatel vlastníka + parametr
public:
    Effect(GameObject* o, float p) : owner_(o), perSec_(p) {}
    void Update(float dt) override {
        if (auto* hp = owner_->Get<Health>()) // efekt nic nedělá, chybí-li Health
            hp->SetCurrent(hp->Current() - perSec_ * dt);
    }
};

struct Poison : Effect { using Effect::Effect; }; // dva různé typy,
struct Bleeding : Effect { using Effect::Effect; }; // aby šly na GO obě naráz

template <typename E> // volba efektu = typový 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));
}

// použití: na kámen a trola zkusíme krvácení 1.2/s (kámen nemá Health -> přeskočí se)
applyEffect<Bleeding>({kamen.get(), trol.get()}, 1.2f);
```

Komentář k volbám. Klíčování podle typu (type_index) splní „nejvýš jedna instance daného druhu“ zdarma; držení přes unique_ptr zaručí, že komponenta patří nejvýš jednomu GO. Health je zapouzdřená, takže invariant $0 \leq \text{cur} \leq \text{max}$ ze zadání hlídá jediné místo (SetCurrent). Get<T> vrací nullptr při absenci – vhodná indikace pro dotaz „má GO tuto komponentu?“. Volitelnost Update přes IUpdatable + **dynamic_cast** respektuje, že ne každá komponenta Update má. Volbu efektu jsme dali do *šablonového* parametru (statický polymorfismus) – alternativně by šel běhový příznak nebo tovární funkce.

Příklad — SZZ podzim 2024 č. 2: Reprezentace typů (návrh + Visitor)

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

Zadání.

(1) Pro IDE navrhnete třídy/rozhraní pro uchování informací o typech a jejich elementech (metody, datové položky, vnořené typy) tak, aby šlo snadno rozšiřovat; jen deklarace. (2) Implementujte metodu, která na typ a všechny jeho elementy aplikuje zadanou „operaci“ – pokud použijete obecně známé řešení průchodu, pojmenujte ho; pro operaci použijte vhodný koncept jazyka. (3) Zavolejte metodu s operací „vytiskni název elementu, jen jde-li o datovou položku nebo typ“.

Řešení.

(1) Hierarchie elementů. Společná abstraktní báze Element, z níž dědí Type, Method, Field. Typ obsahuje své elementy (kompozice); vnořené typy jsou opět Type, takže struktura je rekurzivní (strom).

```
struct Visitor; // dopředná deklarace
struct Element {
    virtual ~Element() = default;
    std::string name;
    virtual void accept(Visitor& v) = 0; // Visitor: double dispatch
};

struct Field : Element { std::string type; void accept(Visitor&) override; };
```

```
struct Method : Element { std::string signature; void accept(Visitor&) override; };
struct Type : Element { // typ drží své elementy (i vnořené typy)
    std::vector<std::unique_ptr<Element>> members;
    void accept(Visitor&) override;
};
```

(2) **Průchod = návrhový vzor Visitor.** Operaci pojmem jako *návštěvníka* s metodou `visit` pro každý druh elementu; metoda `accept` na elementu zavolá správnou `visit` (dvojitá dispatch). Průchod typem aplikuje návštěvníka na typ i rekurzivně na všechny elementy:

```
struct Visitor {
    virtual void visit(Field&) = 0;
    virtual void visit(Method&) = 0;
    virtual void visit(Type&) = 0;
};
void Field::accept(Visitor& v) { v.visit(*this); }
void Method::accept(Visitor& v) { v.visit(*this); }
void Type::accept(Visitor& v) {
    v.visit(*this);
    for (auto& m : members) m->accept(v); // rekurzivně na elementy (i vnořené typy)
}
```

(3) **Konkrétní operace.** Operaci „vytiskni název jen u datové položky nebo typu“ realizujeme konkrétním návštěvníkem (a v `main` ho použijeme). Pro pohodlí by šly metody `visit` delegovat na uloženou *lambdu*, ale s rozhraním `Visitor` je nejpřímější odvodit třídu:

```
struct PrintNamesVisitor : Visitor {
    void visit(Field& f) override { std::cout << f.name << "\n"; } // tiskne
    void visit(Type& t) override { std::cout << t.name << "\n"; } // tiskne
    void visit(Method&) override { /* metodu netiskne */ }
};
void run(Type& root) {
    PrintNamesVisitor pv;
    root.accept(pv); // projde typ i všechny jeho elementy
}
```

i Poznámka. Náčrt v PDF jmenuje právě *Visitor* (případně „sealed“ typy + pattern matching v jazycích, co to mají); operace může být rozhraní, funkcionální typ, anonymní třída či *lambda*. V C++ je idiomatická buď hierarchie `Visitor` (zde), nebo `std::variant` elementů + `std::visit` s *lambdou* (pattern matching). Rozšiřitelnost o nový druh elementu při `Visitoru` znamená přidat metodu `visit` do rozhraní (a doimplementovat ji ve všech existujících návštěvnících) – `Visitor` se proto hodí pro stabilní hierarchie.

Lehčí v C# – pattern matching místo Visitoru

U předchozí otázky náčrt v PDF přímo zmiňuje „sealed typy + pattern matching“ – a pattern matching je v C# běžný idiom. Místo dvojité dispatche (`accept/visit`) stačí jeden `switch` se vzory typů; nepotřebujete rozhraní `Visitor` ani metodu `accept` v každém typu. (Listové třídy lze označit `sealed`, tím se ale v C# jen zakáže další dědění; úplnost `switch` přes celou hierarchii překladač nekontroluje. To umí až Java: uzavřené hierarchie `sealed ... permits` má od verze 17, `switch` s typovými vzory a kontrolou úplnosti od verze 21.)

```
void Visit(Element e) {
    switch (e) {
        case Field f: Console.WriteLine(f.name); break; // tiskni
        case Type t: Console.WriteLine(t.name);
                     foreach (var m in t.members) Visit(m); break; // rekurze
        case Method: break; // metodu netiskni
    }
}
```

```
}
```

Přidat *operaci* je pak triviální (nová funkce se **switch**). Cenou je opačný kompromis než u Visitoru: přidání nového *druhu* elementu si vynutí úpravu všech **switch** (kdežto u Visitoru se mění rozhraní). Pro typické zadání „aplikuj operaci na strom typů“ je ale C# verze výrazně kratší.

4 Manipulace se zdroji a ošetření chyb (výjimky)

Zdroj je něco, co se musí po použití uvolnit: paměť, otevřený soubor, zámek, síťové spojení. V C++ se o uvolnění stará *RAII*, a to spolehlivě i tehdy, když výpočet přeruší *výjimka*. Tato dvě témata spolu úzce souvisí.

4.1 RAII: vázání zdroje na životnost objektu

Definice — RAII

RAII (Resource Acquisition Is Initialization) je idiom, kdy **získání zdroje** svážeme s *konstrukcí* objektu a **uvolnění zdroje** s jeho *destrukcí*. Objekt zdroj *vlastní*; jeho destruktorka zdroj uvolní automaticky na konci platnosti objektu, ať se z bloku odejde jakkoli (normálně, **return**, nebo výjimkou).

Intuice. RAII je v C++ *hlavní* mechanismus správy zdrojů a nahrazuje konstrukci try/finally z jiných jazyků (kterou C++ **nemá**). Místo „nezapomeň v každé větvi uvolnit“ se zdroj uvolní sám v destruktorku. Příkladem jsou `std::unique_ptr` (vlastní paměť), `std::lock_guard` (vlastní zámek), `std::fstream` (vlastní soubor).

```
void prenes(Ucet& z, Ucet& na, int castka) {
    std::lock_guard<std::mutex> g(z.mtx); // zamkne v konstruktoru
    z.zustatek -= castka;
    na.zustatek += castka;
} // lock_guard() odemkne mutex i kdyby zde výpočet skončil výjimkou
```

4.2 Výjimky: vyhození, zachycení, propagace

Definice — Výjimka, throw/try/catch

Výjimka je objekt vyhozený příkazem **throw** při chybovém stavu. Vyhození ukončí běžný tok a hledá nejbližší obklopující blok **try** s **catch**, jehož typ odpovídá (podle dědičnosti) typu vyhozené hodnoty. Nenajde-li se žádný, zavolá se `std::terminate` a program skončí.

Definice — Odvíjení zásobníku (stack unwinding)

Při hledání odpovídajícího **catch** se postupně opouštějí bloky a funkce na zásobníku; přitom se **volají destruktory všech lokálních objektů** v opouštěných rámcích. Tomuto procesu se říká *odvíjení zásobníku*. Právě díky němu RAII uvolní zdroje i při výjimce.

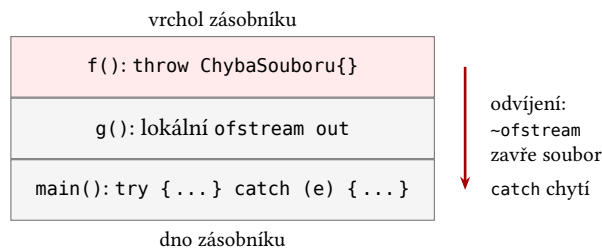
```
struct ChybaSouboru : std::runtime_error { // vlastní výjimka dědí z std::
    using std::runtime_error::runtime_error;
};
void f() { // vyhodí výjimku
    throw ChybaSouboru("nelze zapsat");
}
void g() {
    std::ofstream out("a.txt"); // RAII zdroj v g()
    f(); // f() vyhodí; pak ofstream
```

```

} // při opuštění g() se out korektně zavře (unwinding)
int main() {
    try {
        g();
    } catch (const std::exception& e) { // chytá i potomky (dědičnost)
        std::cerr << "chyba: " << e.what() << "\n";
    }
}

```

i Poznámka. Standardní výjimky dědí z `std::exception` (metoda `what()` vrací popis); dělí se na `std::logic_error` (chyba programátora: `out_of_range`, `invalid_argument`) a `std::runtime_error` (problém za běhu). Vlastní výjimky je dobré odvozovat z `std::exception`. Chytáme *referenci* (`const std::exception&`), aby fungovala dědičnost a nedošlo ke slicing; `throw`; bez argumentu výjimku *přehodí* dál (rethrow).



Schematický řetězec volání `main` → `g` → `f`. Funkce `f()` vyhodí výjimku; ta se propaguje zpět volajícími rámci (`g`, pak `main`) k prvnímu odpovídajícímu `catch` v `main()`. Při opuštění mezilehlého rámce `g()` se zavolá destruktorka jeho lokálního `ofstream`, který zavře soubor – tak RAII zaručí úklid zdrojů i při chybě (C++ nepotřebuje `finally`).

4.3 Bezpečnost vůči výjimkám; proč destruktory nehází

Definice – Úrovně záruky bezpečnosti vůči výjimkám

- *Základní* (basic): i při výjimce zůstanou data v *konzistentním* stavu (žádné úniky, platné invarianty).
- *Silná* (strong): operace buď proběhne celá, nebo nezmění pozorovatelný stav (*commit-or-rollback*).
- *Nothrow* (`noexcept`): operace výjimku nikdy nevyhodí.

i Poznámka. **Destruktorka nesmí vyhodit výjimku** a je implicitně `noexcept`. Důvod: při odvíjení zásobníku už jedna výjimka „letí“; kdyby destruktorka vyhodila druhou, nelze obě zpracovat a volá se `std::terminate`. Proto se v destruktorech nedělají operace, které mohou selhat. Související idiom: *move* operace značíme `noexcept`, jinak je kontejnery (např. `std::vector` při realokaci) z bezpečnostních důvodů nevyužijí a raději kopírují.

5 Životní cyklus objektů a správa paměti (GC, reference counting)

Životní cyklus objektu má tři fáze: *alokace* (kde vznikne), *inicializace* (konstruktor) a *destrukce* (destruktorka). C++ dává programátorovi kontrolu nad všemi třemi a nemá garbage collector – paměť se uvolňuje deterministicky (RAII, chytré ukazatele).

5.1 Alokace a inicializace

i Poznámka. [Tři druhy alokace] *Statická* (globální/**static**): jediná instance, žije po celý běh. *Na zásobníku* (automatická): lokální proměnné, vznikají a zanikají s blokem, velmi rychlé. *Na haldě* (dynamická): **new**/make_unique/make_shared, žijí nezávisle na voláních funkcí, dokud je někdo neuvolní.

i Poznámka. [Explicitní uvolňování a proč se mu vyhýbáme] Haldy lze obsluhovat ručně: $T^* p = \text{new } T$; alokuje, **delete** p ; uvolní. Ruční pár je ale křehký – zapomenutý **delete** je únik paměti, dvojí **delete** nebo **delete** po výjimce je nedefinované chování. Proto moderní C++ alokaci na haldě skoro vždy svěřuje chytrým ukazatelům (make_unique/make_shared), které uvolní automaticky (RAII, sekce 4) – explicitní **delete** v aplikačním kódu je dnes spíš výjimkou. To je protějšek *garbage collectoru* ze spravovaných jazyků (viz níže): tam o uvolnění rozhoduje běhové prostředí, v C++ deterministicky vlastní zdroj.

Definice – Konstruktor, inicializační seznam, pořadí

Konstruktor uvede objekt do platného stavu. Datové položky a předky inicializuje *inicializační seznam* ($T(\text{int } a) : \text{base_}(a), x_ (a) \{ \}$). Pořadí konstrukce je pevné: **nejprve báze** (v pořadí dědění), **pak datové položky** (v pořadí deklarace), nakonec tělo konstruktoru. Destrukce probíhá v *přesně opačném* pořadí.

```
struct Base { Base(int) { /* 1. */ } };
struct Derived : Base {
    Member m_;
    Derived(int a) : Base(a), m_(a) { /* 3. tělo */ }
    // pořadí: Base(a) -> m_(a) -> tělo; destrukce: tělo Derived -> m_ -> Base
};
```

💡 Intuice. Inicializace v seznamu je efektivnější než přiřazení v těle (položka se rovnou zkonstruuje správně, místo aby se nejdřív default-zkonstruovala a pak přepsala). Pořadí podle *deklarace* (ne podle pořadí v seznamu) je častý zdroj chyb, závisí-li jedna položka na druhé.

5.2 Destrukce, kopírování, přesun: pravidlo pěti

Definice – Destruktor; finalizátor

Destruktor $\sim T()$ se zavolá při zániku objektu (konec bloku, **delete**, konec vlastnictví chytrým ukazatelem) a uvolní zdroje. Volá se *deterministicky* a v *přesně daný okamžik*. To je rozdíl proti *finalizátoru* ve spravovaných jazycích (Java `finalize`, v C# metoda zapisovaná $\sim T()$, dříve nazývaná destruktorem), který volá *garbage collector* *nedeterministicky* (nebo vůbec) – proto se na něj nelze spolehnout pro včasné uvolnění zdrojů.

Definice – Copy/move, pravidlo pěti

O kopírování a přesun objektu se stará pět zvláštních metod: *destruktor*, *copy-konstruktor* $T(\text{const } T\&)$, *copy-přiřazení* **operator** $=(\text{const } T\&)$, *move-konstruktor* $T(T\&\&)$ a *move-přiřazení* **operator** $=(T\&\&)$. *Pravidlo pěti*: musíš-li napsat jednu z nich (typicky kvůli vlastněnému zdroji), nejspíš musíš rozmyslet všech pět. *Kopie* duplikuje obsah; *přesun* (z dočasněho objektu) jen „ukradne“ vnitřní data a zdroj vyprázdní (přesně: zaručeně prázdné zůstanou chytré ukazatele; obecně, i u většiny standardních typů, zůstane zdroj v platném, ale nespecifikovaném stavu) – je rychlý a nealokuje.

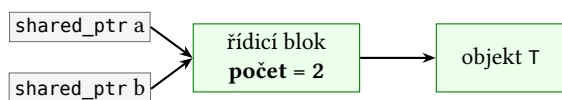
Intuice. Nejlepší je pravidlu pěti se *vyhnout*: drží-li třída zdroje jen přes `std::unique_ptr`/`std::shared_ptr`/kontejnery, generuje překladač všech pět metod správně sám (*pravidlo nuly*). Ručně se píšou jen u tříd, které spravují holý zdroj.

5.3 Chytré ukazatele a počítání referencí

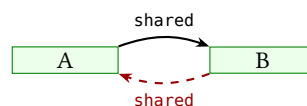
Definice — `unique_ptr`, `shared_ptr`, `weak_ptr`

`std::unique_ptr<T>` – *výhradní* vlastník, nelze kopírovat (jen přesunout), bez režie. `std::shared_ptr<T>` – *sdílený* vlastník; drží *počet referencí* (reference count) v řídicím bloku, objekt se uvolní, až počet klesne na nulu. `std::weak_ptr<T>` – slabý odkaz, který *nezvyšuje* počet referencí (řeší cykly).

Poznámka. [Počítání referencí a problém cyklů] Kopie `shared_ptr` počet referencí zvýší, zánik sníží. Tak se objekt uvolní přesně tehdy, když ho nikdo nevlastní – bez garbage collectoru. **Slabina:** *cyklus* dvou objektů, které se vzájemně drží přes `shared_ptr`, má počet referencí trvale ≥ 1 a *nikdy se neuvolní* (únik paměti). Řešení: jeden směr odkazu udělat `weak_ptr`.



(a) sdílené vlastnictví: oba ukazatele drží též objekt, počet = 2



(b) cyklus: počet nikdy neklesne na 0 $\Rightarrow$ únik; opravit jeden směr na `weak_ptr`

Počítání referencí u `shared_ptr`: řídicí blok eviduje, kolik vlastníků objekt drží (a); při zániku posledního (počet $\rightarrow 0$) se objekt uvolní. Vzájemný odkaz dvou objektů přes `shared_ptr` (b) udržuje počet trvale kladný, takže se neuvolní nikdy (únik paměti) – proto se jeden směr odkazu volí jako `weak_ptr`, který počet nezvyšuje.

5.4 Správa paměti: C++ vs. spravovaná prostředí

Intuice. Spravované jazyky (Java, C#) mají *garbage collector* (GC): běhové prostředí samo najde a uvolní objekty, na které už nevede žádný odkaz. Výhoda: programátor neřeší uvolňování. Nevýhody: nedeterministické pauzy, vyšší spotřeba paměti, pozdní uvolnění *ne-paměťových* zdrojů (soubory, zámky) – proto i ony zavádějí explicitní mechanismy (`using`, `try-with-resources`). C++ jde opačnou cestou: **deterministická** správa přes RAII a chytré ukazatele – zdroj se uvolní v přesně daný okamžik (konec platnosti vlastníka), bez GC a jeho režie. Reference counting (`shared_ptr`) je „lokální“ náhrada GC s přiznanou cenou a omezením (cykly).

Lehčí v C# – GC místo ručního životního cyklu

Otázky, které kladou důraz na *garbage collector*, *finalizátory* nebo prostou správu paměti, jsou v C# přímočařejší: paměť uvolní GC, takže odpadá *pravidlo pěti*, chytré ukazatele i ruční `delete`. Ne-paměťové zdroje (soubor, zámek, spojení) se uklízejí *deterministicky* přes rozhraní `IDisposable` a příkaz `using` – to je protějšek RAII (sekce 4) a Javovského `try-with-resources`:

```
using (var f = File.OpenText("a.txt")) { /* ... */ } // Dispose() na konci bloku
```

Pozn.: na rozdíl od RAII (úklid je automaticky v destruktoru) je `using` *explicitní* a finalizátor `~T()` volá GC nedeterministicky, takže se na něj pro včasné uvolnění zdrojů nelze spolehnout.

6 Vlákna a synchronizace

Vlákno (thread) je samostatný tok výpočtu uvnitř procesu; vlákna téhož procesu sdílejí paměť. C++ je reprezentuje objektem `std::thread` (hlavička `<thread>`). Sdílení paměti přináší *časově závislé chyby*, proti nimž slouží *synchronizační prostředky*.

6.1 Reprezentace vlákna a spuštění

Definice — `std::thread`

`std::thread` je objekt spravující jedno běžící vlákno operačního systému. Konstruktor dostane *volatelné* (funkci, lambda, funktor) a jeho argumenty a vlákno **ihned spustí**. Vlákno je nutné před zánikem objektu buď `join()` (počkat na jeho dokončení), nebo `detach()` (osamostatnit). Zanikne-li `std::thread` ještě `joinable()`, zavolá se `std::terminate`.

```
#include <thread>
void prace(int id) { /* ... */ }

std::thread t1(prace, 1);           // spustí vlákno s funkcí + argumentem
std::thread t2([]{ /* ětlo */ });   // nebo s lambdou
unsigned n = std::thread::hardware_concurrency(); // počet HW vláken
t1.join();                         // počká na dokončení t1
t2.join();
```

i Poznámka. Funkce vykonávaná vláknem se předá jako kterékoli volatelné; argumenty se vláknu *kopírují* (pro předání odkazem je nutné `std::ref`). Výjimka vyhozená ve vlákně se v jiném vlákně přímo chytit nedá – nezachytí-li ji vlákno samo, volá se `std::terminate`. Přenést ji do jiného vlákna lze jen explicitně (`std::exception_ptr`, případně ji přeneseme `std::async/std::future` při volání `get()`).

6.2 Časově závislé chyby a kritická sekce

Definice — Souběh / datový závod, kritická sekce

Časově závislá chyba (race condition) nastává, když výsledek závisí na nedeterministickém prokládání vláken plánovačem OS. *Datový závod* je její typický případ: dvě vlákna přistupují ke sdílené proměnné a aspoň jedno zapisuje, bez synchronizace. *Kritická sekce* je úsek kódu, který smí v jeden okamžik provádět nejvýš jedno vlákno.

```
int citac = 0;           // sdílený stav
void inc() { ++citac; }   // ++ NENÍ atomické: načti, zvyš, ulož
// Dvě vlákna volají inc() 1000x. Bez synchronizace může výsledek být < 2000:
// obě načtou stejnou hodnotu, obě uloží +1, jeden zápis se "ztratí".
```

💡 Intuice. Operace `++citac` se přeloží na tři kroky (čti, zvyš, zapiš). Proloží-li plánovač dvě vlákna mezi čtením a zápisem, oba načtou např. 5, oba zapíšou 6 – jeden přírůstek zmizí. Proto je výsledek nedeterministický a může být menší, než by měl být (v C++ je datový závod navíc nedefinované chování).

6.3 Synchronizační prostředky

Definice – Mutex, lock_guard

`std::mutex` (vzájemné vyloučení) zajistí, že kritickou sekci provádí nejvýš jedno vlákno: vlákno mutex *zamkne* (lock) a po dokončení *odemkne* (unlock). RAII obal `std::lock_guard` zamkne v konstruktoru a **odemkne v destruktoru** – nelze zapomenout odemknout ani při výjimce.

```
std::mutex m;
int citac = 0;
void inc() {
    std::lock_guard<std::mutex> g(m); // zamkne;odemkne se na konci bloku
    ++citac;                         // kritická sekce: jen jedno vlákno naráz
}
```

Lehčí v C# – lock místo mutex + lock_guard

C# má pro kritickou sekci přímo příkaz `lock`: zápis `lock (zamek) { ++citac; }` zamkne objekt na dobu bloku a spolehlivě odemkne i při výjimce (přeloží se na `Monitor.Enter/Exit` v `try/finally`). Je stručnější než dvojice `std::mutex + std::lock_guard`, kterou ovšem v podstatě jen zapouzdřuje. Vlákno se spustí přes `new Thread(prace).Start()` nebo úlohou `Task.Run(...)`; vyšší úroveň nabízí `async/await`.

i Poznámka. [Uvážnutí a další prostředky] *Uvážnutí* (deadlock) nastane, když dvě vlákna čekají navzájem na zámky, které drží to druhé. Pomáhá zamykat více mutexů *najednou* přes `std::scoped_lock` (zamkne je v bezpečném pořadí). Další prostředky: `std::condition_variable` (vlákno čeká na podmínku a je probuzeno notifikací – producent/konzument), `std::atomic<T>` (operace bez zámku, např. atomický ++). Podrobnosti *paměťového modelu* atomických operací jdou nad rámec požadavků okruhu.

7 Implementace OO jazyků (tabulka virtuálních metod)

Tato sekce ukazuje, *jak* se objektové konstrukce realizují v paměti – zvlášť dynamický polymorfismus přes *tabulku virtuálních metod*.

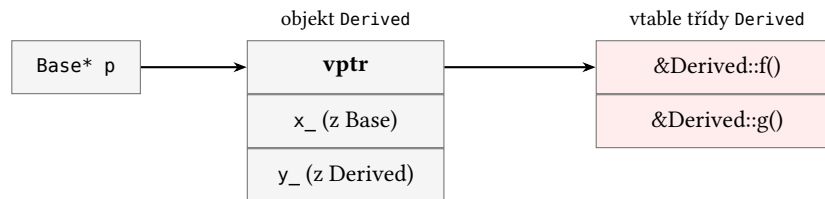
7.1 Reprezentace primitivních a složených typů

i Poznámka. *Primitivní* typy mají pevnou velikost a v paměti leží přímo (bez hlavičky). *Složený* objekt (`struct/class`) je v paměti *posloupnost* svých datových položek v pořadí deklarace; mezi ně překladač vkládá *výplň* (padding) kvůli zarovnání (alignment). Objekt odvozené třídy obsahuje *podobný* báze (její položky) následovaný vlastními položkami. Třída **bez** virtuálních metod (a bez virtuálních bází) nemá v objektu žádnou skrytou režii – proto C++ objekt může být stejně úsporný jako čisté C.

7.2 Tabulka virtuálních metod (vtable)

Definice – vtable, vptr

Má-li třída aspoň jednu virtuální metodu, překladač pro ni vytvoří *tabulku virtuálních metod* (vtable) – pole ukazatelů na konkrétní implementace virtuálních metod této třídy. Každý objekt takové třídy nese skrytý ukazatel *vptr* na vtable své (nejodvozenější) třídy. Volání virtuální metody přes ukazatel/referenci pak proběhne nepřímou: *dereferencuj vptr → najdi v tabulce položku metody → zavolej*.



Rozložení objektu Derived (báze Base): na začátku skrytý **vp**, pak položky báze a potomka. **vp** míří na vtable třídy Derived, kde jsou adresy skutečných implementací virtuálních metod. Volání `p->f()` přes `Base*`: přečti **vp**, vyber položku `f`, zavolej `Derived::f` – to je *dynamic* výběr podle skutečného typu objektu, ne podle typu ukazatele.

Poznámka. Cena dynamického polymorfismu = jedna nepřímost navíc (přes vtable) a skrytý **vp** v objektu; proto se v C++ za virtuální metody „platí“ jen tam, kde se použijí. Vedle adres metod nese vtable i pomocné položky (zejména ukazatel na `type_info` pro RTTI – běhovou typovou informaci – a `dynamic_cast`), které ve skutečnosti leží *před* adresami metod; pro princip je ale podstatné právě pole adres metod, na jehož začátek **vp** ukazuje. U *vícenásobné* dědičnosti má objekt více **vp** (po jednom na každou polymorfní bázi); u *virtuální* dědičnosti se navíc poloha sdílené báze zjišťuje přes uložené offsety. Konstrukce vtable se dělá automaticky; je to běžný způsob, jak objektové jazyky (C++, ale i Java/C#) implementují virtuální volání.

8 Nativní a interpretovaný běh, překlad a sestavení

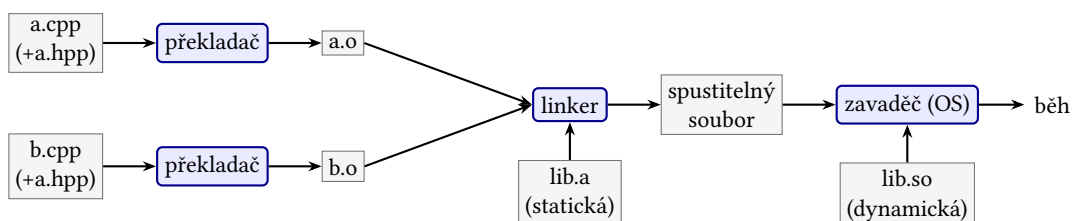
Poslední téma je „cesta od zdrojového kódu ke spuštěnému programu“: jak se C++ překládá a sestavuje a čím se to liší od interpretovaných / bytecode jazyků.

8.1 Reprezentace programu, oddělený překlad

Definice – Hlavička, překladová jednotka, deklarace vs. definice

Hlavičkový soubor (.hpp) obsahuje *rozhraní* – deklarace (co existuje); *zdrojový soubor* (.cpp) obsahuje *implementaci* – definice (jak to funguje). *Překladová jednotka* (translation unit) je jeden .cpp po vložení všech `#include`. *Oddělený překlad* znamená, že každá překladová jednotka se přeloží **nezávisle** na *objektový soubor* (.o); změna jednoho .cpp nevyžaduje překlad ostatních.

Poznámka. *Deklarace* oznamuje existenci jména (smí jich být víc); *definice* ho zavádí (*One Definition Rule*, ODR: nejvýš jedna v celém programu; výjimkou jsou definice tříd, `inline` funkce a šablony, jejichž shodná definice smí být v každé překladové jednotce). `#include` je čistě textové vložení (preprocesorem). Šablony musí být celé v hlavičce (definice musí být viditelná v místě instanciaci), proto nejdou oddělit do .cpp.



Sestavení programu v C++: každá překladová jednotka (a.cpp, b.cpp) se přeloží *nezávisle* na objektový soubor (.o). *Linker* slije objektové soubory a *statické* knihovny (.a) do spustitelného souboru. *Zavaděč* operačního systému ho při spuštění nahraje do paměti a připojí *dynamické* (sdílené) knihovny (.so).

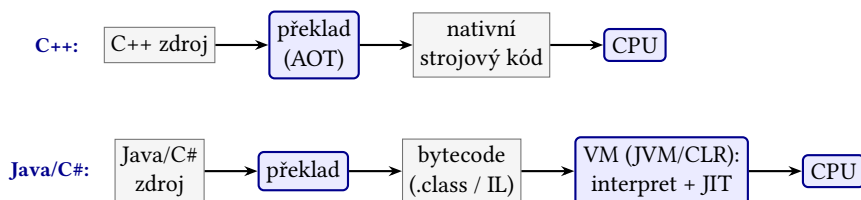
8.2 Statické a dynamické knihovny

Poznámka. *Statická* knihovna (.a / .lib) se **při linkování zkopíruje** (její potřebné části) do spustitelného souboru – program je samostatný, ale větší a knihovnu nelze aktualizovat bez relinkování. *Dynamická / sdílená* knihovna (.so / .dll) se připojí **až za běhu** zavaděčem – spustitelný soubor je menší, jednu kopii kódu v paměti sdílí více procesů a knihovnu lze aktualizovat zvlášť (cenou je závislost na přítomnosti správné verze). Program si může sdílenou knihovnu načíst i sám za běhu (dlopen v POSIX, LoadLibrary ve Windows) a vyhledat v ní symboly podle jména – tak fungují *pluginy* (viz otázka jaro 2023).

8.3 Nativní vs. interpretovaný běh, AOT vs. JIT

Definice – Nativní/AOT vs. interpretovaný/bytecode/JIT

C++ je překládán *dopředeně* (AOT, ahead-of-time) přímo do *nativního strojového kódu* cílové platformy; za běhu nepotřebuje interpret ani překladač. Naproti tomu jazyky jako Java či C# se překládají do *bytecode* (mezikódu: .class u Javy, IL u .NET), který běží ve *virtuálním stroji* (JVM, .NET CLR): ten ho *interpretuje*, nebo *překládá za běhu* (JIT, just-in-time) na nativní kód. JVM typicky kombinuje obojí (interpretuje a JIT překládá jen často prováděná místa), .NET CLR metody před prvním voláním JIT překládá. I pro tyto jazyky existuje AOT překlad do nativního kódu (např. GraalVM Native Image, .NET Native AOT); AOT tedy není vlastnost jazyka, ale způsobu jeho implementace.



Nativní (AOT) běh C++ vs. bytecode/JIT běh spravovaných jazyků. C++ se přeloží dopředeně na nativní strojový kód, který procesor vykonává přímo. Java/C# se přeloží na přenositelný bytecode, jež virtuální stroj za běhu interpretuje nebo JIT-překládá na nativní kód. AOT dává předvídatelný výkon a žádnou běhovou režii navíc; JIT umožňuje přenositelnost a optimalizace podle skutečného běhu za cenu rozběhové latence a běhového prostředí.

8.4 Běhové prostředí a vazba na OS

Poznámka. Spustitelný soubor zavádí do paměti *zavaděč* operačního systému, předá řízení startovacímu kódu běhového prostředí (inicializace, konstruktory globálních objektů) a ten zavolá *main*. *Běhové prostředí* (runtime) C++ je minimální (na rozdíl od JVM/CLR) – v omezených prostředích (embedded, jádro OS) může program běžet skoro bez něj. C/C++ je proto jazykem operačních systémů a běhových knihoven jiných jazyků.

Příklad – SZZ léto 2022 č. 2: Překlad konstrukcí vyššího jazyka

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

Zadání.

Procesor inspirovaný MIPS má registry \$R0–\$R31 (\$R0 = 0) a PC; instrukce LD reg=[mem], ST [mem]=reg, ADD r1=r2, r3, J label, BEQ/BNE r1, r2, label (skok při rovnosti / nerovnosti) a SLT r1=r2, r3 (r1 = (r2 < r3)). Dána je posloupnost:

```
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
```

Úkoly: (1) vybrat z nabídnutých úryvků v C ty, které posloupnosti odpovídají, a připsat k instrukcím čísla řádků odpovídajících příkazů; (2) jak se C kód změní, nahradíme-li BEQ za BNE (stejně operandy); (3) upravit posloupnost tak, aby realizovala podmínku rozšířenou o $i < b$.

Řešení.

(1) Test cyklu `SLT $r4=$r1,$r2 + BNE $r4,$r0,lab2` běží *dokud* $i < n$ (řídící i je $r1$, start q , krok inc). V těle dá `SLT $r5=$r5,$r4` hodnotu $r5 = (b < a+i)$ a `BEQ $r5,$r0,lab3` *přeskočí* uložení, když $r5 == 0$, tj. $a+i \leq b$; $b = a+i$ se tedy provede právě když $a+i > b$. Posloupnost realizuje (a odpovídají *obě* tyto nabízené varianty):

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

Čísla řádků: načtení q , n , inc , skok J lab1, přičtení inc a test SLT/BNE na konci patří k hlavičce `for`; načtení a , b , součet, SLT a BEQ k `if`; ST k přiřazení (u varianty s `while` obdobně, přehled je v dokumentu *Příklady otázek*).

(2) Záměna `BEQ`→`BNE` obrátí podmínku skoku: přeskočení nastane, když $r5 \neq 0$ (tj. $a+i > b$), takže uložení proběhne v opačném případě. Podmínka se změní na `if (a+i <= b) b = a+i;`.

(3) Rozšíření o $i < b$: před `ST [b]=$r4` vložíme druhý test ($i < b$); když neplatí, přeskočíme uložení na lab3:

```
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 neplatí
ST  [b]=$r4          ; ulozi se jen kdyz a+i > b AND i < b
```

Shrnutí

i Poznámka. Klíčové pojmy a fakta (přehled k SZZ). Hlavní témata okruhu I3: OO návrh a rozhraní, C++ idiomy (RAII, šablony, chytré ukazatele), implementace runtime (vtable, překlad). Zkratky: UB = undefined behaviour; RAII = Resource Acquisition Is Initialization; ODR = One Definition Rule.

Abstrakce, zapouzdření, polymorfismus, dědičnost.

- Abstrakce = skrytí implementace za rozhraní; zapouzdření = řízený přístup přes `public/protected/private`. Dědičnost IS-A (veřejná); potomek rozšiřuje a může překrývat.
- Virtuální metoda (`virtual`): dynamický dispatch za běhu přes `vptr`. Nevirtuální: statická vazba za překladu. Čistě virtuální ($= 0$): potomek musí překrýt; třída s aspoň jednou $= 0$ = abstraktní (nelze instanciovat) = C++ „rozhraní“.
- Polymorfní báze **musí mít virtuální destruktory**; jinak `delete` přes ukazatel na bázi = UB.
- Přetěžování** (overloading): stejné jméno, různé signatury – volba staticky za překladu. **Překrývání** (overriding): `virtual + override` – volba dynamicky za běhu. `override` zachytí překlep v signatuře.
- Pozor na **slicing**: přiřazení polymorfního objektu *hodnotou* ořízne potomkovy členy.
- SZZ vzory**: IFilter/IImageFormat (bitmapový editor, jaro 2023); IGraph/IVertex/IEdge (podzim 2022); Matrix (jaro 2025); hierarchie Element/Type/Method/Field (podzim 2024).

Vícenásobná dědičnost, virtuální dědičnost, rozhraní.

- Diamantový problém: dvě kopie báze S (přes R , W) → dvojznačnost přístupu a redundantní data.
- Virtuální dědičnost (`class R : public virtual S`): jedna sdílená instance S .
- C++ nemá klíčové slovo `interface`; „rozhraní-abstraktní třída“ (čistě virtuální metody + virtuální

destruktor, žádné datové položky).

- Vhodné použití: IS-A → dědičnost; HAS-A → **kompozice** (upřednostnit); obecný kontrakt → abstraktní třída.

Primitivní a složené typy; ukazatele, reference, const.

- Číselné: `int/long/float/double/bool/char`; `enum class`: typově bezpečný výčet (nevzniká implicitní převod na `int`).
- Ukazatel `T*`: může být `nullptr`, lze rebindovat, aritmetika. Reference `T&`: alias, nesmí být null, nelze rebindovat (může ale „viset“, zanikne-li odkazovaný objekt dřív než reference).
- `const T&`: přijme dočasný i pojmenovaný. `T&`: r-value reference; základ pro přesun bez kopírování (move semantics).
- `const` na metodě: garantuje nemodifikaci `*this` (kromě `mutable` položek). Vlastník vs. pozorovatel: `unique_ptr/shared_ptr` vlastní; holý `T*` nebo reference pozoruje (nesmí přežít vlastníka).

Generické typy a statický polymorfismus.

- Šablony (templates): instanciaci za překladu pro každý použitý typ; bez běhové rezie. Požadavky na typ jsou implicitní (duck typing – chybějící operátor/metoda = chyba za překladu).
- Specializace: úplná funguje u šablon funkcí i tříd; částečná **jen u šablon tříd** (ne u šablon funkcí).
- Statický polymorfismus (šablony) vs. dynamický (virtuální metody, vtable): statický = nulová nepřímost za běhu; dynamický = heterogenní kolekce přes ukazatele/reference na bázi.
- `std::type_index`: klíč odvozený od RTTI (`typeid`); klíčování typů v mapě za běhu – viz SZZ léto 2023 (`unordered_map<type_index, unique_ptr<Component>>`).

Typy reprezentující funkce; lambda, funkcionální rozhraní.

- Ukazatel na funkci `R(*) (Args...)`: nejjednodušší; jen prosté funkce (ne lambdy se stavem).
- Funktor (třída s `operator()`): nese stav; základ, z něhož kompilátor generuje lambda.
- Lambda: `[capture](params){body}` = anonymní funktor; `[=]` zachytí vše použité hodnotou; `[&]` referencí; `[this]` ukazatel na objekt.
- `std::function<R(Args...)>`: type erasure (typ volatelného se „vymaže“) – přijme ukazatel, funktor i lambda; mírná běhová rezie.
- *SZZ podzim 2024*: průchod strukturou typu = vzor Visitor (`accept/visit`, dvojitá dispatch); operace = konkrétní návštěvník. Alternativy: operace jako lambda/`std::function`, nebo `std::variant` elementů + `std::visit`.

RAII a výjimky.

- RAII: zdroj (soubor, mutex, paměť) = životnost objektu; konstruktor získá, destruktor uvolní. Správa funguje i při výjimce. Nahrazuje `finally` z Javy/C#.
- Výjimky: `throw/try/catch`, hierarchie `std::exception`. Odvíjení zásobníku (stack unwinding) při výjimce volá destruktory všech lokálních objektů ve všech rámcích na cestě.
- Záruky: *basic* (invariant zachován, žádný únik), *strong* (commit-or-rollback), *nothrow*. Dstruktor by **nikdy** neměl házet (`noexcept` od C++11 implicitně).
- *SZZ jaro 2023*: `unique_ptr<IFilter>` a `unique_ptr<IImageFormat>` = RAII pro polymorfní objekty; `std::function` jako Factory pro formáty načítané za běhu z pluginu (.dll).

Životní cyklus a správa paměti.

- Alokace: statická (`static`/globální), zásobník (lokální proměnné), halda (`new/delete`, lépe `make_unique/make_shared`).
- Pořadí init: báze → datové položky → tělo konstruktoru; destrukce přesně opačně. Položky inicializuje inicializační seznam (ne tělo konstruktoru); pořadí určuje pořadí deklarace, ne pořadí v seznamu.
- **Pravidlo pěti**: vlastní-li třída zdroj, definuj (nebo = `delete`) destruktor, kopírovací konstruktor, kopírovací přiřazení, přesouvací konstruktor, přesouvací přiřazení. **Pravidlo nuly**: spolehni se na RAII členy, nic nedefinuj.
- `unique_ptr`: exkluzivní vlastnictví (move-only). `shared_ptr`: sdílené vlastnictví + ref-count. Cyklus `shared_ptr` ⇒ únik → `weak_ptr` (nezvyšuje počet).

- C++ nemá GC: deterministická správa, destruktory volán na konci životnosti. Java/C#: GC nedeterministický; finalizátor (`Finalize`) není totéž co destruktory.

Vlákna a synchronizace.

- `std::thread(f, args...)`: spustí `f` v novém OS vlákně; `join()` čeká na dokončení; `detach()` odpojí (vlákno běží dál, nelze `joinovat`).
- Race condition: paralelní neatomická čtení-modifikace-zápis (neatomické ++) → nedeterministický výsledek; datový závod (data race) je v C++ UB.
- Kritická sekce: `std::mutex + lock_guard` (RAII – zamkne při vzniku, odemkne při destrukci). Deadlock = kruhové čekání na zámky. `std::atomic<T>`: atomické operace bez mutexu.
- `std::condition_variable`: čekání na podmínku (vlákno čeká, probuzeno notifikací – vzor producent/konzument); musí být doprovázena mutexem.

Implementace OO (vtable, vptr, RTTI).

- Objekt = posloupnost položek (nejdříve báze, pak potomkovy) + skrytý **vptr** na začátku (u polymorfni třídy).
- vtable = statické pole adres virtuálních metod třídy; vptr ukazuje na vtable; virtuální volání = dereference vptr + nepřímý skok.
- RTTI (Runtime Type Information) leží ve vtable před adresami metod (Itanium ABI: g++, clang); `dynamic_cast` a `typeid` přistupují přes RTTI.
- `dynamic_cast<D*>(p)`: za běhu ověří RTTI; vrátí `nullptr` při selhání (pro reference hodí `bad_cast`). Vyžaduje polymorfni hierarchii (aspoň jedna `virtual`).

Překlad, sestavení, AOT vs. JIT.

- Pipeline: preprocesor (`#include`, `#define`) → kompilátor (`.cpp` → `.o`) → linker (`.o` + knihovny → spustitelný) → zavadač OS (+ dynamické knihovny).
- Statická knihovna (`.a` / `.lib`): kód se zkopíruje do exe (větší, soběstačný). Dynamická (`.so` / `.dll`): sdílená za běhu; lze dodat jako plugin za běhu – viz SZZ jaro 2023 (formáty obrázků jako `.dll` pluginy).
- ODR: každý symbol smí mít nejvýš jednu definici v programu. Třídy, `inline` funkce a šablony smějí být v hlavičce (vícenásobné definice se sloučí).
- C++ = nativní/AOT: `.cpp` → strojový kód; žádný interpret ani GC. Java/C# = bytecode (`.class` / `IL`) + VM (JVM/CLR) + JIT (přeloží kód za běhu; JVM hlavně často prováděná místa); AOT je i u nich možný (GraalVM, .NET Native AOT). AOT výhody: deterministická latence, žádný warmup; JIT výhody: portabilita, run-time optimalizace dle dat.

Kdy je lehčí C# – viz fialové boxy v textu.

- Rozhraní: klíčové slovo `interface` místo abstraktní třídy s čistě virtuálními metodami.
- Vícenásobná dědičnost: C# ji nemá ⇒ žádný diamantový problém; jedna třídni báze + libovolné rozhraní.
- Visitor: switch se vzory typů (pattern matching) místo `accept/visit` – viz SZZ podzim 2024.
- Správa paměti: GC, žádné pravidlo pěti; `using/IDisposable` pro deterministické uvolnění zdrojů.
- Kritická sekce: příkaz `lock(obj) {...}` místo `mutex + lock_guard`.

Zdroje

- D. Bednárek: *Programování v C++* (NPRG041), přednáškové slidy, MFF UK, 2025/26 (primární zdroj výkladu: ukazatele a reference, copy/move, chytré ukazatele, šablony, lambdy, dědičnost a vtable, výjimky, překlad a modularita).
<https://teaching.mff.cuni.cz/nprg041-web/>
- D. Bednárek: *Pokročilé programování v C++* (NPRG051), přednáškové slidy, MFF UK, 2025/26 (bezpečnost vůči výjimkám, šablony do hloubky, vlákna a synchronizace, atomické operace).
<https://teaching.mff.cuni.cz/nprg051-web/>
- *cplusplusreference.com*: referenční příručka jazyka C++ a standardní knihovny (ověření sémantiky)

jazykových konstrukcí a knihovních tříd).

<https://en.cppreference.com/w/>

- *Itanium C++ ABI*: specifikace binárního rozhraní používaného překladači g++ a clang (rozložení objektů a tabulek virtuálních metod).

<https://itanium-cxx-abi.github.io/cxx-abi/abi.html>

- Microsoft: *C# documentation* (Microsoft Learn), dokumentace jazyka C# (rámečky *Lehčí v C#*).

<https://learn.microsoft.com/en-us/dotnet/csharp/>

- MFF UK: požadavky ke státní závěrečné zkoušce (Informatika, Bc.) a zadání otázek z minulých termínů.

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