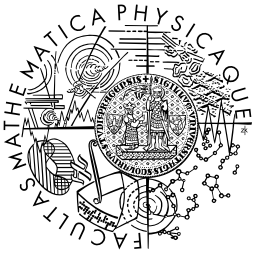


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁRSKA PRÁCA

Peter Fačko

Systém pre introšpekciu v C++

Katedra softwarového inženýrství

Vedúci bakalárskej práce: RNDr. David Bednárek, Ph.D.

Študijný program: Informatika

Študijný odbor: Programování a softwarové systémy

Praha 2021

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne a výhradne s použitím citovaných prameňov, literatúry a ďalších odborných zdrojov. Táto práca nebola využitá k získaniu iného alebo rovnakého titulu.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona v platnom znení, najmä skutočnosť, že Univerzita Karlova má právo na uzavretie licenčnej zmluvy o použití tejto práce ako školského diela podľa §60 odst. 1 autorského zákona.

V ..... dňa ....

Podpis autora

Ďakujem vedúcemu tejto práce RNDr. Davidovi Bednárkovi, Ph.D. za ochotu a čas, ktorý mi pri písaní tejto práce venoval.

Ďalej ďakujem svojej rodine a priateľom za podporu počas celého štúdia.

Názov práce: Systém pre introšpekciu v C++

Autor: Peter Fačko

Katedra: Katedra softwarového inženýrství

Vedúci bakalárskej práce: RNDr. David Bednárek, Ph.D., Katedra softwarového inženýrství

Abstrakt: Introšpekcia a reflexia sú silné programovacie nástroje umožňujúce všeobecné riešenie problematických úloh, ako napríklad multiple dispatch alebo serializácia. V C++ je podpora pre introšpekciu veľmi slabá a reflexia nie je možná vôbec. Štandard jazyka sa za posledné obdobie značne rozvinul, vďaka čomu je už možné do jazyka tieto mechanizmy doplniť vlastným rozšírením. Výsledkom tejto práce je knižnica pre introšpekciu pracujúca s metadátami vygenerovanými zo zdrojového textu a generátor týchto metadát ako nástroj nad prekladačom. Funkčnosť celého systému je demonštrovaná implementáciou mechanizmu multiple dispatch, ktorý nie je jednoducho implementovateľný v štandardnom C++.

Kľúčové slová: introšpekcia C++ analýza zdrojových textov

Title: A system for introspection in C++

Author: Peter Fačko

Department: Department of Software Engineering

Supervisor: RNDr. David Bednárek, Ph.D., Department of Software Engineering

Abstract: Introspection and reflection are powerful programming tools that allow generic solutions for difficult problems such as multiple dispatch or serialization. In C++, the support for introspection is seriously limited, while reflection is virtually non-existent. The language standard has been developing significantly in recent past and thus it is now possible to extend the language with those features using a custom tool. The result of this thesis is an introspection library working with metadata generated from source code and a compiler-based metadata generator. The functionality of the whole system shall be demonstrated by implementing multiple dispatch, for which there is no simple implementation in standard C++.

Keywords: introspection C++ source-code analysis

Obsah

1. Úvod

Hlavnými zameraniami tejto práce sú *reflexia* a *introšpekcia*. Reflexia je schopnosť programu manipulovať s reprezentáciami stavu daného programu ako s dátami. Jeden z aspektov reflexie je introšpekcia, čo je schopnosť programu pozorovať svoj vlastný stav (Demers a Malenfant, 1995). Introšpekcia je v moderných programovacích jazykoch bežnou schopnosťou: jej podpora sa nachádza napríklad v C#, Java, JavaScript a Python, čo sú niektoré z najpopulárnejších jazykov dnešnej doby (JetBrains, 2020).

Podobne populárnym jazykom je C++. V ňom je na rozdiel od zvyšných jazykov podpora pre introšpekciu minimálna. Dokážeme získať veľkosť a zarovnanie typu a z objektov získať za behu určité veľmi obmedzené informácie o ich type. Táto minimálna podpora dynamickej introšpekcie je spôsobená tým, že jazyk C++ sa vo všeobecnosti snaží o čo najmenšie behové prostredie. Obmedzená statická introšpekcia je zrejme dôsledkom náročnosti štandardizačného procesu, keďže návrh na jej podporu existuje už nejaký čas, no zatiaľ nebol do štandardu začlenený (Loikkanen, 2019). Je možné, že sa stane súčasťou C++23.

V iných sférach pokročil jazyk C++ výrazne. Hlavným príkladom je podpora vykonávania stále viac úkonov, ktoré sú typicky možné iba za behu programu, aj počas prekladu. Menšími novinkami sú napríklad koncepty a moduly. Tento nedávny posun umožňuje dotvoriť do jazyka podporu pre introšpekciu vo forme knižnice, teda bez potreby obmeny prekladača.

1.1 Využitie introšpekcie

Pred pokusmi implementovať do existujúceho jazyka tak rozsiahle rozšírenie, ako je introšpekcia, je vhodné spomenúť, aký zmysel má takú schopnosť v jazyku podporovať.

1.1.1 Serializácia

Jedným využitím introšpekcie môže byť automatická serializácia. Užitočné by bolo napísať jeden všeobecný serializačný algoritmus a každú triedu spracovať na základe jej štruktúry. Informácie o štruktúre by pochádzali práve z introšpekcie. Pre jednoduchú *triviálnu* triedu bez odkazov by mohol algoritmus postupne prechádzať dátové členy triedy daného objektu a rekurzívne ich serializovať. Triviálne triedy sú napríklad C structy.

1.1.2 Registrácia do knižnice

Všeobecnejšie využitie introšpekcie by bolo možné pri knižnici, ktorá potrebuje registrovať do nejakej štruktúry. Môže ísť napríklad o parser príkazového riadku, do ktorého sa zaznamenávajú akceptované príkazy. Uvažujme nasledujúce príkazy:

```
namespace commands
{   struct add
    {   static void execute(int a, int b)
        { out << a + b << '\n'; }
    };

    struct say
    {   static void execute(string message)
        { out << message << '\n'; }
    };
}
```

Typicky by sa tieto príkazy registrovali do parseru nejako takto:

```
parser.register("add", &commands::add::execute);
parser.register("say", &commands::say::execute);
```

Táto explicitná registrácia by pri introšpekcii nebola nutná. Knižnica by mohla prehľadať namespace `commands` a pre každú triedu v ňom registrovať jeden príkaz s menom danej triedy a handler funkciou `execute`. Predpokladáme, že knižnica by bola schopná zistiť, ako parsovať argumenty z typov parametrov funkcie, bez introšpekcie, hoci aj s takým úkonom by mohla introšpekcia pomôcť.

1.1.3 Multiple dispatch

Multiple dispatch je mechanizmus programovacích jazykov uplatňujúci sa pri volaní funkcií, kde dochádza k výberu implementácie funkcie na základe *dynamických typov* argumentov (Muschevici, Potanin, Tempero a Noble, 2008). Dynamický typ výrazu je v C++ typ najviac odvodeného objektu (v zmysle dedičnosti), na ktorý výraz odkazuje.

Ak by daná forma introšpekcie podporovala volanie funkcií s *overload resolution* (OR) — stačilo by aj v obmedzenej podobe — tieto volania by v podstate podporovali multiple dispatch.

Zjednodušene povedané — overload resolution v C++ funguje tak, že pri volaní vyberá tú najlepšiu funkciu na základe typov parametrov a argumentov. Príklad:

```
struct B {};

struct D1 : B {};
struct D2 : B {};

void f(const D1& a, const D1& b) { out << "11"; }
void f(const D1& a, const D2& b) { out << "12"; }
void f(const D2& a, const D1& b) { out << "21"; }
void f(const D2& a, const D2& b) { out << "22"; }

f(D1(), D2()); // vypíše 12
f(D2(), D2()); // vypíše 22
```

Overload resolution vykonáva prekladač pri preklade; v preloženom programe sa nachádza na mieste volania už len jedna selektovaná funkcia. Introšpekcia by mohla byť schopná robiť OR aj za behu:

```
reflection::invoke("f", D1(), D2()); // vypíše 12
reflection::invoke("f", D2(), D2()); // vypíše 22
```

Tento príklad nie je úplný; na ozajstný multiple dispatch by ešte bolo potrebné, aby introšpekcia podporovala *dynamickú referenciu*. To by bola pomocná trieda, ktorej objektami by sa dalo odkazovať na objekt ľubovoľného typu. Rozbor tejto schopnosti introšpekcie urobíme až pri návrhu programu.

1.2 Introšpekcia v C++

C++ ponúka introšpekciu v dvoch podobách.

Do prvej spadajú operátory `sizeof`, `sizeof...` a `alignof`. Napríklad, `sizeof` akceptuje ako svoj argument typ alebo výraz a vracia veľkosť daného typu, respektíve veľkosť typu daného výrazu. Ak je argument výraz, zvažuje sa jeho *statický typ*, nie dynamický typ. Statický typ je typ plynúci z deklarácie v zdrojovom texte, napríklad pri výraze volania funkcie je to návratový typ funkcie.

```
struct Base {};
struct Derived : Base {};

Base b;
Derived d;

Base& x = b;
Base& y = d;

x; // static type: Base, dynamic type: Base
y; // static type: Base, dynamic type: Derived
```

`sizeof` svoj argument nevyhodnocuje; predmet jeho skúmania je entita zdrojového textu, nie hodnota v pamäti. Ostatné dva operátory fungujú obdobne: nezvažujú stav programu počas behu, ale informácia, ktorú ponúkajú, plynie iba z formy zdrojového textu. Táto podoba introšpekcie pozoruje vlastnosti programu v kompilačnom čase - nazveme ju teda *kompilačná introšpekcia*.

Druhá podoba introšpekcie v C++ je založená na operátore `typeid`. Ten akceptuje typ alebo výraz a vracia odkaz na `std::type_info`, ktorý reprezentuje daný typ, respektíve typ výrazu. `std::type_info` má definované úplné usporiadanie a tiež dokáže získať meno reprezentovaného typu. Rozdiel oproti kompilačnej introšpekcii je tu v tom, že `typeid` svoj argument vyhodnocuje, pretože ako typ výrazu rozumie jeho dynamický typ. Ten je známy až za behu programu z hodnoty výrazu. Táto introšpekcia pozoruje program za behu - nazveme ju *behová introšpekcia*.

Tento pohľad na druhy introšpekcie v C++ ukazuje istú medzeru v ponúkaných nástrojoch štandardným C++. Ak jazyk ponúka spôsob, ako z typu určiť jeho veľkosť, a tiež poskytuje spôsob, ako z hodnoty premennej určiť jej „behový“ typ,

prečo nemožno jednoducho určiť z hodnoty premennej veľkosť jej „behového“ typu? To by vyžadovalo mapovanie medzi reprezentantmi typov a informáciami o typoch z kompilačnej introšpekcie. Toto prepojenie medzi kompilačnou a behovou introšpekciou bude jedným zo zámerov práce.

V tejto kapitole sú zámerne vynechané detaily, ktoré nie sú podstatné pre uvedenie do problematiky introšpekcie v C++. Špeciálne: dynamický typ má zmysel iba pre výrazy kategórie *glvalue*, *prvalue* výrazy sa v `typeid` nevyhodnocujú a `typeid` rozlišuje dynamický typ iba výrazov *polymorfných typov*. Všetky tieto nepresnosti a pojmy budú v práci vyjasnené neskôr, ak budú mať nejaký dopad na fungovanie programu.

1.3 Cieľ práce

Cieľom práce je rozšíriť schopnosti jazyka C++ o introšpekciu. Vo všeobecnosti sa programovací jazyk dá rozšíriť úpravou prekladača. To má žiaľ mnoho nevýhod:

- Dnešné prekladače sú veľmi veľké programy a vyznať sa v nich dostatočne na nejaký zmysluplný zásah je náročné.
- Aktuálne neexistuje jeden dominantný prekladač C++. V roku 2020 boli tri prekladače, ktoré používalo pravidelne aspoň 30 % programátorov C++ (JetBrains, 2020). Z toho by vyplývala potreba implementovať na viaceré platformy, ak by mal byť projekt využiteľný pre podstatné množstvo programátorov.
- Prekladače sa neustále vyvíjajú, čiže zachovanie kompatibility by vyžadovalo pravidelnú údržbu.
- Je nepravdepodobné, že by malý projekt jednej osoby bol začlenený do riadneho prekladača, bez ohľadu na kvalitu práce.

Okrem úpravy prekladača je možné rozšíriť C++ o introšpekciu použitím symbolov vygenerovaných pre ladenie programu alebo prenesením zodpovednosti na užívateľa, ktorý manuálne poskytne potrebné meta-informácie (Bräutigam, 2015). Nepoužijeme ani jeden z troch zatiaľ uvedených spôsobov.

Cieľ bude nájsť také riešenie, ktoré nevyžaduje upravenú verziu prekladača. Navyše aj také, ktoré nebude vyžadovať explicitnú registráciu entít jazyka do knižnice. Celkovo je cieľom, aby reflektovaný kód mohol ostať nezmenený. Hoci ide o knižnicu, snaha je, aby mal užívateľ pocit, že používa novú verziu jazyka.

Introšpekciou sa — tak, ako ju implementuje tento projekt — rozumie:

- Užívateľ môže reflektovať entity na *deskriptory*, ktoré uchovávajú informáciu o danej entite jazyka. Deskriptor je obyčajný C++ objekt. Napríklad, entite jazyka namespace zodpovedá objekt typu `namespace_t`. Deskriptory sú tiež schopné enumerovať svoje časti, členov, predkov a podobne.
- Je možné *dynamické volanie funkcií*, teda funkciu je možné zavolať cez jej deskriptor. Toto platí vrátane všetkých mechanizmov jazyka aplikujúcich sa počas volania; menovito: overload resolution a *implicitné konverzie*. Implicitné konverzie sú automatické premeny hodnôt a typov na argumentoch

volania funkcie uskutočnené pred volaním, aby sa zabezpečila typová zhoda s parametrami.

- Užívateľ má možnosť vytvoriť objekt dynamického typu (podobné `dynamic` zo C#). Tieto dynamické objekty budú pri volaní vracať deskriptory funkcií. Tiež bude k dispozícii dynamická referencia.

Ďalší cieľ je použiť `constexpr` tam, kde je to možné a stojí to za to. Nie je dôvod, aby užívateľ nemohol v kompilačnom čase získať z deskriptoru typu jeho veľkosť alebo napríklad aj jeho meno; všetky tieto informácie sú plne dostupné už pri preklade. Naopak, volanie funkcií je omnoho zložitejší úkon, ten `constexpr` byť nemusí a ani by to nemalo veľké využitie.

Ako demonštrácia fungovania projektu bude slúžiť implementácia *visitor* návrhového vzoru tak, že navštevovaná trieda nebude potrebovať implementovať virtuálnu funkciu prijímajúcu visitor. Ide o špeciálny prípad multiple dispatch. Zmysel takéhoto zadania je vynútiť mechanizmus schopný multiple dispatch a zároveň zachovať možnosť vytvoriť jednoduchý ekvivalent pomocou štandardného jazyka.

Tiež poskytneme porovnanie výkonu medzi virtuálnym volaním v C++, virtuálnym volaním v C#, volaním s použitím `dynamic` v C# a dynamickým volaním funkcií pomocou introšpekcie v C++.

Názov celého projektu je *PP*reflection.

1.4 Štruktúra práce

V kapitole 2 preskúmame už existujúce implementácie introšpekcie a porovnáme ich schopnosti.

Ako vedľajší produkt tohoto projektu vznikla knižnica *PP*, jej návrh a základné idey vysvetlíme v kapitole 3.

Návrh a implementáciu introšpekcie vytvoríme v kapitole 4. V tejto kapitole vyriešime úlohy zadané cieľom práce.

V kapitole 5 vysvetlíme použitie knižnice s konkrétnymi príkladmi.

Výsledky práce, teda porovnanie výkonu oproti programom s ekvivalentným chovaním bez použitia introšpekcie, prezentujeme v kapitole 6.

Na záver práce zhrnieme nedostatky a možné vylepšenia práce.

2. Súvisiace práce

V tejto kapitole preskúmame existujúce implementácie introšpekcie v C++. Cieľ je tiež ukázať „medzeru“ v ich schopnostiach, ktorú *PP*reflection zapĺňa. Časťou analýzy každej knižnice bude príklad jej použitia. Pri každom príklade predpokladáme existenciu nasledovnej triedy:

```
struct Struct {
    int field;
    S() : field(7) {};
    void method(double) {};
};
```

2.1 RTTR

Knižnica RTTR ponúka introšpekciu s manuálnou registráciou. Pri tejto manuálnej registrácii musí užívateľ explicitne vymenovať všetky entity, ktoré chce do introšpekcie registrovať. Dokáže napríklad enumerovať členov, vytvárať dynamické objekty, dynamicky volať funkcie a pristupovať ku dátovým členom skrz reflektované entity.

Dynamické volania vracajú typ `variant`, ktorý môže obsahovať objekt ľubovoľného typu. Nie sú pri nich povolené žiadne konverzie — typy parametrov a argumentov sa musia zhodovať. Tiež pri nich nie je možné použiť `variant` ako argument.

Autor: Axel Menzel, `https://github.com/rttrorg/rttr`.

Príklad

```
#include <rttr/registration>

RTTR_REGISTRATION
{   rttr::registration::class_<Struct>("Struct")
          .property("field", &Struct::field)
          .constructor<>()
          .method("f", &Struct::method);
}
// ...
Struct s;
auto t = type::get<Struct>();
auto field = t.get_property("field");
std::cout << field.get_value(s).get_value<int>(); // vypíše 7
```

2.2 meta

Táto knižnica nepoužíva makrá. Namiesto toho vytvára objekty `factory`, do ktorých sa manuálne registrujú informácie o entite. Má podobné schopnosti ako RTTR. Dynamické volania vracajú typ `any`, ktorý — na rozdiel od RTTR — je

možné použiť ako argument. Takisto ako predošlá knižnica nepodporuje konverzie, typy sa pri volaní musia zhodovať.

Autor: Michele Caini, `https://github.com/skypjack/meta`.

Príklad

```
#include <meta/factory.hpp>
#include <meta/meta.hpp>

std::hash<std::string_view> hash{};

meta::reflect<Struct>(hash("Struct"))
    .data<&Struct::field>(hash("field"));
    .ctor<>()
    .func<&Struct::method>(hash("method"));

Struct s;
auto t = meta::resolve<Struct>();
auto field = t.data(hash("field"));
std::cout << field.get(s).cast<int>(); // vypíše 7
```

2.3 SelfPortrait

Táto knižnica ponúka generátor metadát, takže nie je potrebná explicitná registrácia entít. Inak je schopnosťami porovnateľná s meta. Tiež neimplementuje konverzie pri dynamickom volaní.

Autori: de Bayser a Cerqueira (2012), `https://github.com/maxdebayser/SelfPortrait`.

Príklad

```
VariantValue s = Struct();
auto t = Class::lookup("Struct");
auto field = t.getAttribute("field");
std::cout << field.get(s).value<int&>(); // vypíše 7
```

2.4 Ostatné

- Ponder - používa makrá, nutná explicitná registrácia
  `https://github.com/billyquith/ponder`
- Refureku - nutné použitie makier na registráciu na mieste deklarácie entít
  `https://github.com/jsoysouvanh/Refureku`
- CPP-Reflection - zdanlivo populárna knižnica, no zdá sa byť nedokončená
  `https://github.com/AustinBrunkhorst/CPP-Reflection`

2.5 Zhrnutie

Použitie makier je medzi knižnicami pre introšpekciu populárny spôsob ako generovať metadáta. Žiaľ, v istom zmysle popiera princíp introšpekcie ako schopnosti jazyka. Ak musí užívateľ explicitne registrovať entity jazyka do knižnice, nejde potom o schopnosť *jazyka* reflektovať na svoju štruktúru. Užitočnosť týchto knižníc to samozrejme nepopiera — nutnosť registrovať entity sa môže aj tak vyplatiť — no poukazuje to na silný nedostatok. Tiež sme neboli schopní nájsť ani jednu knižnicu, ktorá by sa aspoň snažila mechanizmus volania reflektovaných funkcií priblížiť tomu zo štandardného C++, teda implementovala by implicitné konverzie.

Názov projektu	**Bez makier**	**Automatická registrácia**	**Konverzie**
Ponder	-	-	-
Refureku	-	-	-
RTTR	-	-	-
meta	✓	-	-
SelfPortrait	✓	✓	-
*PP*reflection	✓	✓	✓

Tabuľka 2.1: Porovnanie knižníc na introšpekciu v C++

3. Knižnica *PP*

Pri vývoji projektu vznikalo veľké množstvo kódu, ktorý nijako priamo nesúvisí s introšpekciou, preto sme z neho vytvorili samostatne stojacu knižnicu *PP*. Knižnica ponúka nástroje na metaprogramovanie, prácu s typmi, príjemnejšiu prácu s n-ticami, funkcie vyššieho rádu a `constexpr` implementácie niektorých štandardných tried.

Knižnica je výrazne inšpirovaná funkcionálnym programovaním, no zámerne to počas návrhu nespomíname, aby bolo zrejmé, že dané princípy možno objaviť čisto z potreby napísať čitateľnejší a kratší kód.

V ukážkach zdrojových textov sú vynechané deklarácie namespace, ktoré si je možné ľahko domyslieť.

3.1 Forwarding

Hoci v celom projekte sa držíme zásady použiť čo najmenej makier, jedna operácia má v C++ také chovanie, že makrá sú jediný spôsob, ako zvýšiť čitateľnosť kódu. Ide o *forwarding referencie*. Predpokladáme, že čitateľ má dostatočné znalosti C++, aby rozumel tomuto chovaniu.

Na lepšiu prácu s forwarding referenciami si vytvoríme tieto pomocné konštrukty:

```
#define PP_F(x) static_cast<decltype(x)&&>(x)

template <typename T> class forward_wrap
{   T&& ref;
public:
    constexpr forward_wrap(T&& ref) noexcept
        : ref(PP_F(ref)) {}
    constexpr forward_wrap(const forward_wrap& other) noexcept
        : ref(PP_F(other.ref)) {}

    constexpr decltype(auto) operator--(int) const noexcept
    { return PP_F(ref); }

    constexpr decltype(auto) operator()(auto&&... args) const
    { return PP_F(ref)(F(args)...); }
};
template <typename T> forward_wrap(T&&) -> forward_wrap<T>;

constexpr decltype(auto) unwrap_forward(auto&& x) { return PP_F(x); }

template <typename T>
constexpr decltype(auto) unwrap_forward(const forward_wrap<T>& x)
{ return x--; }

#define PP_FW(x) ::PP::forward_wrap{ PP_F(x) }
```

`PP_F(x)` je ekvivalentný `std::forward<T>(x)` pre premennú deklarovanú ako `T&& x`.

`forward_wrap` má dve využitia. Prvým je jednoduchšie forwardovanie: stačí nám raz zabaliť referenciu do objektu `forward_wrap` a potom pri každom volaní

`operator--` interne prebehne forward. Toto využijeme hlavne pri lambda capture, pretože to sprehľadní jej telo. Z toho istého dôvodu sme pridali aj operátor volania. Druhé využitie má trieda pri volaniach funkcií vyššieho rádu, ktoré vracajú novú funkciu a potrebujú do nej uložiť niektorý zo svojich parametrov. Keďže parametre sú referenčné, funkcia by potrebovala vedieť, či parameter skopírovať do hodnotovej premennej, alebo ho odovzdať ako referenciu. Toto vyriešime tak, že funkcie vyššieho rádu budú parameter vždy kopírovať, a ak užívateľ chce predať argument ako referenciu, zabalí ju do `forward_wrap`.

```
T object = /*...*/;
auto f = hof(functor, object);
auto g = hof(functor, std::move(object));
auto h = hof(functor, PP_FW(object);
auto i = hof(functor, PP_FW(std::move(object));
```

Funktor	**Typ vnútornej premennej**	**Inicializátor**
f	T	object
g	T	std::move(object)
h	T&	object
i	T&&	std::move(object)

Funkciám vyššieho rádu sa bližšie venuje kapitola 3.3.

3.2 Transformácie typov

Prvou a hlavnou úlohou, ktorú má knižnica *PP*, je zabezpečiť jednoduchý spôsob ako transformovať typy. Transformáciou typu rozumieme zobrazenie z typov na typy alebo hodnoty.

Ak by sme transformovali hodnoty, C++ ponúka jednoduchý nástroj: funkcie. Táto transformácia prijíma hodnotu a vracia hodnotu:

```
size_t times2(size_t a) { return 2 * a; }
```

Ak potrebujeme transformovať typy na typy, C++ ponúka šablónový alias.

```
template <typename T> using add_const_t = const T;
```

Ďalší silný nástroj, ktorý nám jazyk ponúka, je možnosť použiť tieto entity ako argumenty iných transformácií.

```
constexpr decltype(auto) applier(auto&& f, auto&&... a)
{ return PP_F(f)(PP_F(a)...); }
// times2(x) ~ applier(times2, x)

template <template <typename...> typename TT, typename... T>
using applier_t = TT<T...>;
// add_const_t<U> ~ applier_t<add_const_t, U>
```

Parameter `f` funkcie `applier` sme deklarovali ako forwarding referenciu, teda do argumentu môžeme vložiť ľubovoľný objekt, na ktorom sa dá zavolať operátor volania s danými argumentami.

Ďalšia možnosť je, že chceme transformovať typy a dostať hodnotu. Na to jazyk ponúka šablónové premenné.

```
template <typename T> constexpr inline auto size_of = sizeof(T);
```

Problém tohoto konštruktu je, že ho nemôžeme použiť ako argument žiadnej transformácie. Štandardná knižnica rieši tento problém tak, že entita pre danú transformáciu je vždy šablónová trieda, ktorá má v sebe vnorený výsledok ako alias, resp. statickú premennú.

```
template <typename T> struct add_const { using type = const T; };
template <typename T>
struct size_of { static constexpr auto value = sizeof(T); };
```

Pri implementácii `applier` môžeme využiť dedičnosť. Tak dostaneme do triedy potomka aliasy aj statické premenné.

```
template <template <typename...> typename TT, typename... T>
struct applier : TT<T...> {};

// size_of<U>::value ~ applier<size_of, U>::value
```

3.2.1 Hodnota ako typ

Problém by sa mohol zdať vyriešený, ale nie je to tak. Transformáciu `applier` už ďalej použiť ako argument nemôžeme. To je hlavný problém riešenia cez šablónové triedy. Menším problémom je, že sme rozdelili svet transformácií na dva. Jeden používa funkcie, ktoré transformujú hodnoty, a druhý používa šablónové triedy, ktoré transformujú typy. Pokiaľ ostávame vo svete šablónových tried, všetko je v poriadku. Akonáhle potrebujeme tieto dva svety spojiť, čaká nás nepríjemné „vybaľovanie“ výsledkov. Alternatívou je vytvoriť pomocný konštrukt, ktorý zoberie transformáciu ako funkciu a vytvorí z nej transformáciu ako triedu.

```
template <auto f> struct to_class
{   template <typename V> struct get
    { static constexpr auto value = f(V::value); };
};
// times2(size_of<U>::value) ~ to_class<times2>::get<size_of<U>>::value
```

Tak sme oddialili vybalenie zase až na koniec výrazu. Tiež sme vytvorili transformáciu `to_class<f>::template get`, ktorá sa dá použiť ako argument. Môžeme si tu všimnúť ešte jednu zaujímavosť: táto transformácia prijíma myšlienkovo hodnotu, ale v kóde akceptuje typ. Skúsime tento trik otočiť, a nepoužijeme myšlienku „hodnota ako typ“, ale „typ ako hodnota“.

3.2.2 Typ ako hodnota

Najprv vyriešime problém vybaľovania. Už vieme, že ak máme operáciu, ktorá prijíma typ a vracia hodnotu, môžeme použiť šablónovú premennú alebo šablónovú triedu; ich vlastnosti sme si už priblížili. Je ale aj iná možnosť: šablónové funkcie.

```
template <typename T> constexpr auto size_of() { return sizeof(T); }
```

Funkcia vracia priamo hodnotu, takže žiadne vybaľovanie nie je potrebné. Chyba ale je, že šablónová funkcia sa nedá použiť ako argument. To môžeme vyriešiť tak, že z nej vytvoríme *funktor*, teda objekt, ktorý sa dá zavolať ako funkcia. V C++ to najjednoduchšie docielime lambdou.

```
constexpr inline auto size_of = []<typename T>() { return sizeof(T); }
```

Problém je, že takýto funktor sa nedá zavolať. Typ kam vložiť nemáme a parametre nie sú žiadne. Tu nastupuje princíp typ ako hodnota. Využijeme mechanizmus jazyka *template argument deduction* (TAD). Je to mechanizmus, ktorý z typov argumentov šablónovej funkcie vydedukuje jej šablónové argumenty.

```
template <typename T> struct type_t {};
template <typename T> constexpr inline type_t<T> type = {};

constexpr inline auto size_of = []<typename T>(type_t<T>)
                                { return sizeof(T); }

// sizeof(U) ~ size_of(type<U>) ~ applier(size_of, type<U>)
```

Chovať sa k typom ako k hodnotám, je základná myšlienka knižnice *PP*. Tak môžeme všetky pomocné konštrukty v knižnici implementovať iba raz tak, aby pracovali s hodnotami, a chovať sa k typom aj hodnotám jednotne. Veľkou výhodou tohoto princípu je, že cez hodnoty môžeme vložiť typy do `tuple` a používať na nich všetky už existujúce konštrukty.

```
constexpr inline auto sizeofs = []<typename... T>(type_t<T>...)
                                { return (0 + ... + sizeof(T)); }

// apply(sizeofs, make_tuple(type<U>, type<V>)) ~ sizeof(U) + sizeof(V)
```

3.2.3 Typ ako koncept

Zatiaľ sme vo funkciách získavali z `type_t` typ pomocou TAD. Mohli by sme vytvoriť všeobecnejší spôsob, ktorý by sa dal použiť vo všetkých kontextoch. Môžeme do typu `type_t` pridať alias.

```
template <typename T> struct type_t { using type = T; };

constexpr inline auto sizeofs =
    [](auto... types)
    { return (0 + ... + sizeof(typename decltype(types)::type)); }
```

V tomto kóde sa nikde v `sizeofs` nespomína `type_t`. Jediná podmienka je, že dostaneme objekt nejakého typu, ktorý má alias `type`. Vytvorme teda taký koncept a pomocné konštrukty na získanie typu.

```
template <typename T>
concept type = requires { typename std::remove_reference_t<T>::type; };
template <typename T>
using get_type_t = std::remove_reference_t<T>::type;
#define PP_GT(x) get_type_t<decltype(x)>
```

Použitie `std::remove_reference_t` je dôležité, pretože v poslednej implementácii `sizeofs` sme potrebovali mať parametre *hodnotové*. To kladie ďalšiu podmienku na koncept typu: musí byť kopírovateľný. Ak sa chceme tejto podmienky zbaviť, zmeníme parametre na `auto&&`, lenže tak nám `decltype` vráti referenčný typ. Tejto referencie navyše sa zbavíme pomocou `std::remove_reference_t`.

```
constexpr inline auto sizeofs =
    [](concepts::type auto&&... t)
    { return (0 + ... + sizeof(PP_GT(t))); };
```

3.3 Funkcie vyššieho rádu

Pri návrhu transformácii typov sme dbali na jednu vec: aby transformácie boli entity, ktoré sa dajú predávať ako argument iným transformáciám. Teraz túto vlastnosť využijeme na vytvorenie *funkcií vyššieho rádu*, teda funkcií ktoré prijímajú alebo vracajú iné funkcie.

3.3.1 Funktory operátorov

Častým argumentom funkcií vyššieho rádu budú operátory. Tie v C++ nemajú podobu funktoru, takže si ich musíme vytvoriť. Pre každý binárny operátor `OP` vytvoríme funktor tohoto tvaru:

```
constexpr inline auto opr = [](auto&& x, auto&& y)
                            { return PP_F(x) OP PP_F(y); };
```

Analogicky vytvoríme funktory aj pre unárne operátory. Názvy týchto funktorov sa budú vždy skladať z troch písmen, napríklad `neg` pre `operator!`.

3.3.2 Vlastné operátory

Pre niektoré unárne a binárne funkcie bude vhodné vytvoriť operátor s rovnakým chovaním. To pre nového používateľa tejto knižnice sprvu zhorší čitateľnosť, ale veríme, že pri tak rozsiahlom používaní funkcií vyššieho rádu ako je v tejto knižnici to časom spôsobí opačný efekt a čitateľnosť to zvýši. Príkladom je tento riadok kódu z *PP*reflection, ktorý z n-tice objektov rôznych tried so spoločným predkom vytvorí pole referencií na daného predka.

```
auto arr = PP::static__cast * PP::type<const enum_value&> << tup;
```

Bez použitia vlastných operátorov by tento kód vyzeral takto:

```
auto arr = PP::tuple_map_forward_array(
    PP::apply_partially_first(
        PP::static__cast, PP::type<const enum_value&>),
    tup);
```

3.3.3 `functor`

Na vytvorenie vlastných operátorov budeme potrebovať ešte pár konštruktov navyše. Vezmime si ako príklad vymyslenú binárnu funkciu vyššieho rádu `f`, o ktorej sme sa rozhodli, že ju zastupuje operátor `+`.

```
constexpr decltype(auto) operator+(auto&& x, auto&& y)
{ return f(PP_F(x), PP_F(y)); }
```

Problém takto deklarovaného operátoru je, že jeho parametre sa viažu na ľubovoľný argument. Čiastočne to rieši mechanizmus jazyka ADL, ktorý by sme využili, ak by tento operátor bol deklarovaný vnútri nejakého namespace a použitý mimo neho. To ale stále nerieši problém použitia zvnútra namespace. Nechceme, aby sme omylom volali funkciu `f`, keď potrebujeme napríklad sčítať dve matice. Potrebovali by sme nejaký koncept, ktorým by sme obmedzili typy parametrov.

Pri funkciách vyššieho rádu je takmer vždy niektorým parametrom funktor. Bohužiaľ, v C++ nedokážeme vyrobiť všeobecný koncept, ktorý by hovoril, že daný objekt sa dá zavolať s *nejakými* argumentami. Najbližšie k nemu je koncept „zavolateľného“ typu, ale ten potrebuje typy argumentov daného volania. Túto situáciu vyriešime vytvorením konceptu, ktorý bude vyžadovať iba možnosť zavolať určitú funkciu na objekte danej triedy. Táto funkcia má vrátiť objekt, na ktorom je možné urobiť volanie.

```
template <typename T>
concept functor = requires { std::declval<T>().unwrap_functor(); } ||
                  requires { unwrap_functor_impl(std::declval<T>()); };
```

Tento koncept samozrejme nijako nezaručuje schopnosť správneho volania cez mechanizmy jazyka, no dáva sémantické obmedzenie. Ak užívateľ poskytne pre určitý typ dané funkcie, „registruje“ typ ako funktor. Predpokladáme, že to neurobí omylom.

K tomuto konceptu navyše vytvoríme funkciu `unwrap_functor`, ktorá z objektu spĺňajúceho `functor` „vybalí“ vnútorný objekt a pre ostatné sa chová ako identita.

Druhou možnosťou pre typ argumentu vlastného operátoru funkcie vyššieho rádu je `forward_wrap`. Pre prípad, že užívateľ zabalí do `forward_wrap` nejaký `functor`, vytvoríme aj funkciu `unwrap`, ktorá vybaľuje `forward_wrap` aj `functor` a volá sa rekurzívne. Vlastné operátory budú používať koncept `wrap`, ktorý je disjunkciou `functor` a konceptu, ktorý spĺňa iba trieda `forward_wrap`.

```
constexpr decltype(auto) operator+(
    concepts::wrap auto&& x, concepts::wrap auto&& y)
{ return f(unwrap_functor(PP_F(x)), unwrap_functor(PP_F(y))); }
```

Tiež vytvoríme implementáciu konceptu `concepts::functor`: šablónovú triedu `functor`. Táto trieda bude iba obalom, ktorý spĺňa daný koncept. Všetky funktory vytvorené knižnicou budú objekty tejto šablóny. Vďaka tomu budeme môcť volať vlastné operátory na hodnotách vrátených funkciami knižnice.

Ďalej budeme namiesto zápisu:

```
constexpr inline auto hof = functor{lambda};
```

používať pseudo kód:

```
hof = lambda;
```

3.3.4 Skladanie

Často využívanou operáciou na funkciách bude skladanie funkcií. Pri tejto funkcii ešte kvôli názornosti uvedieme implementáciu.

```
compose =
    [](auto&& ff, auto&& gg)
    { return functor{[d = unwrap_functor(PP_F(ff)),
                      g = unwrap_functor(PP_F(gg))]
                     (auto&&... args) requires /**/
                     { return unwrap(f)(unwrap(g)(PP_F(args)...)); }}; };

constexpr auto operator|(concepts::wrap auto&& f, concepts::wrap auto&& g)
{ return compose(unwrap_functor(PP_F(f)), unwrap_functor(PP_F(g))); }

// compose(f, g)(...) === f(g(...))
// requires { compose(f, g)(...); } <=> requires { f(g(...); }
```

`requires` je potrebné kvôli tomu, aby mohol užívateľ testovať platnosť výrazu, ktorý používa objekt vrátený funkciou `compose`. Konkrétny výraz je vynechaný kvôli stručnosti.

3.3.5 Parciálna aplikácia

Parciálna aplikácia je operácia na funkcii, ktorá fixuje argumenty funkcie. Napríklad, pre binárnu funkciu, ktorá násobí dve čísla, by sa mohla parciálnou aplikáciou vytvoriť unárna funkcia, ktorá násobí dvojkou. Vo všeobecnosti táto operácia môže fixovať ľubovoľný počet argumentov na ľubovoľných pozíciách. Ukázalo sa, že v celom projekte túto operáciu vyžadujeme 68-krát, no vždy fixujeme práve jeden argument a 62 prípadov z toho fixujeme iba prvý. Implementácia špeciálnejšej formy tejto operácie, ktorá fixuje iba prvý argument, je omnoho jednoduchšia a má príjemnú vlastnosť binárnej operácie. Vytvoríme teda dve verzie, `apply_partially_first` a všeobecné `apply_partially`. Všeobecná verzia bude akceptovať okrem funkcie, ktorú aplikuje, a argumentu, ktorý fixuje, aj index, na ktorom fixovaný argument leží. `apply_partially_first` dostane operátor `*`.

3.3.6 Negácia

Príjemnou operáciou by bolo vytvorenie negácie z funkcie, teda pre funkciu `f` vytvoriť `nf`, aby `nf(...)` `!f(...)`. Vďaka dvom už vytvoreným funkciám bude táto implementácia veľmi jednoduchá, čo demonštruje zmysel podpory funkcií vyššieho rádu.

```
constexpr inline auto negate = compose * neg;
```

3.4 Hodnota ako hodnota

Predpokladajme nasledovnú situáciu: máme funkciu, ktorá potrebuje dostať ako parameter hodnotu známu už v kompilačnom čase. Jeden príklad by bola naša funkcia `apply_partially`, ale názornejší príklad je funkcia, ktorá z n-tice získa prvok na danom indexe. Štandardná knižnica takúto funkciu ponúka a jej forma je `std::get<I>(t)`, kde `I` je index a `t` je n-tica (`std::tuple`). Problém predávania argumentu indexu pomocou šablónového argumentu je, že pri našom návrhu funkcií vyššieho rádu nemôžeme túto funkciu napríklad parciálne aplikovať a vytvoriť unárnu operáciu, ktorá získava prvok n-tice vždy na rovnakom indexe. Potrebovali by sme, aby sa informácia o indexe predávala cez obyčajný argument. Podobný problém sme už vyriešili pri typoch. Vytvorili sme prázdnu šablónovú triedu, ktorú sme použili ako parameter šablónovej funkcie a pomocou TAD sa vydedukoval typ. To isté môžeme urobiť pre hodnoty.

```
template <auto v>
struct value_t { static constexpr auto value_f() { return v; } };
```

Tak ako pri typoch, aj pri hodnotách bude užitočné vytvoriť z tohoto princípu koncept. Postup je analogický k tomu pri typoch, takže ho preskočíme. Takto by potom vyzerala implementácia `get` ako `functor`:

```
get_ = [](concepts::value auto&& i, auto&& tuple)
       { return get<PP_GV(i)>(PP_F(tuple)); };
```

3.5 n-tice

Veľmi užitočnou triedou ponúkanou štandardnou knižnicou je `std::tuple`. Je často čitateľnejším nástrojom na manipulovanie s n-ticami hodnôt ako je *parameter pack*. Parameter pack je parameter, ktorý akceptuje ľubovoľný počet argumentov; píše sa s troma bodkami, napríklad `auto&&...` `std::tuple` tiež ponúka výhodu, že z hľadiska jazyka ide o jeden objekt. Bohužiaľ, jediný nástroj na manipulovanie s `std::tuple`, ktorý štandardná knižnica ponúka, je `std::apply`, ktorý vo všeobecnosti aj tak vyžaduje ako argument funkciu, ktorá akceptuje parameter pack.

3.5.1 Koncept

Aby sme neboli obmedzení len na jednu triedu, vytvoríme nový koncept: `concepts::tuple`. Potrebovali by sme, aby sme mohli pre každý index do veľkosti n-tice zavolať nejakú prístupovú operáciu ku prvku. Zatiaľ vytvorme pomocný koncept pre prístup na *jeden* index:

```
template <typename T, auto I>
concept tuple_access = requires { std::declval<T>()[value<I>]; };
```

Takto definovaný koncept n-tice má veľkú chybu: `std::tuple` ho nespĺňa a ani ho splniť nijako nemôže. `operator[]` totiž musí byť vždy členskou funkciou. Toto je vedomé rozhodnutie, keďže si v tomto projekte viac ceníme možnosť prehľadne sa odkazovať na prvky n-tice ako zjednodušenie, ktoré plynie z už existujúcej implementácie konceptu. Minimálna implementácia konceptu by bol obal `std::tuple`, ktorého `operator[]` volá `std::get`. Aby sme mohli používať *structured binding* (`auto [x, y] = tuple`) pre vlastnú implementáciu n-tice, budeme potrebovať špecializovať `std::tuple_size` a `std::tuple_element`. Tiež je nutné implementovať funkciu get. To môžeme vďaka konceptu vyriešiť jednou implementáciou:

```
template <size_t I>
constexpr decltype(auto) get(concepts::tuple auto&& t)
{ return PP_F(t)[value<I>]; }
```

Z potreby kontrolovať indexy do veľkosti n-tice vyplýva, že potrebujeme aj operáciu, ktorá nám vráti veľkosť n-tice. Keďže veľkosť n-tice potrebujeme poznať už pri preklade, návratový typ operácie, ktorá vráti veľkosť, potrebuje spĺňať koncept `value`. Podobne ako pri koncepte `functor`, dáme implementácii možnosť poskytnúť členskú aj namespace funkciu.

Ako z informácie o veľkosti n-tice zistíme, či je možné získať prvky na *každom* indexe? Pridáme parameter. Ak by sme ako šablónové parametre mali všetky indexy, na ktorých chceme skontrolovať prístup, koncept by sa implementoval jednoducho:

```
template <typename T, auto... I>
concept tuple_accesses = (tuple_access<T, I> && ...);
```

Problém je len, ako dostať do šablónového parametru hodnoty. Zase použijeme TAD. Na to potrebujeme funkciu, ktorá akceptuje nejaký šablónový typ s indexami, nazvime ho `value_sequence`. Bude to prázdna trieda, jediná jej informácia leží v šablónových parametroch.

```
template <auto... I>
constexpr auto helper(concepts::type auto&& t, value_sequence<I...>)
{ return tuple_accesses<PP_GT(t), I...>; }
```

S takouto pomocnou funkciou už ľahko implementujeme celý koncept.

value_sequence

Ostáva vyriešiť, ako vytvoriť objekt šablóny `value_sequence` z veľkosti n-tice. Rekurzívne riešenie s lineárnou zložitosťou je celkom triviálne. My vytvoríme efektívnejšiu implementáciu s iba logaritmom volaní. Vytvoríme pomocnú funkciu, ktorá zdvojnásobí dĺžku sekvencie jednoducho takto: zapíšeme ju dvakrát za sebou a v druhej kópii pričítame ku každému číslu dĺžku vstupnej sekvencie.

```
template <auto... I>
constexpr auto double_value_sequence(value_sequence<I...>)
{ return value_sequence<I..., (sizeof...(I) + I)...>{}; }
```

S touto pomôckou je vytvorenie sekvencie s dĺžkou $n \neq 0$ iba zdvojenie sekvencie s dĺžkou $\lfloor \frac{n}{2} \rfloor$ a pridanie čísla n na koniec, ak je nepárne.

3.5.2 Operácie

Pri skoro každej operácii na n-ticiach budeme potrebovať indexovať všetky prvky. Ako sme už videli, to vieme docieliť pomocnou funkciou, ktorá cez TAD vytvorí zoznam indexov ako parameter pack. Aby sme nemuseli zakaždým písať tieto pomocné funkcie, vytvoríme si abstrakciu. `std::apply` je jedným príkladom takej abstrakcie, nám sa avšak bude hodiť ísť ešte o stupeň vyššie. Vytvoríme funktor `apply_pack`:

```
apply_pack = []<auto... I>
             (auto&& packer, auto&& selector, value_sequence<I...>)
                 -> decltype(auto)
             { return unwrap_functor(PP_F(packer))
                      (unwrap_functor(PP_F(selector))(value<I>)...); };
```

`selector` určuje, aký význam má daný index, `packer` je operácia, ktorú chceme nakoniec vykonať. Ekvivalent `std::apply` by potom bol:

```
tuple_apply =
    [](auto&& f, concepts::tuple auto&& t) -> decltype(auto)
    { return apply_pack(unwrap_functor(PP_F(f)),
                        tuple_get(partial_tag, value<1>, PP_FW(t)),
                        tuple_value_sequence_for(PP_F(t))); };
```

`packer` je tu f a `selector` je operácia, ktorá z t vyberie prvok na danom indexe. `tuple_value_sequence_for` vytvorí sekvenciu $(0, \ldots, n-1)$ pre n-ticu rozmeru n.

Mapa

Druhou operáciou, ktorú implementujeme, bude *mapa*, teda funkcia vyššieho rádu, ktorá aplikuje unárnu funkciu na každý prvok n-tice. Vďaka `tuple_apply` je implementácia vcelku triviálna. Priradíme jej operátor `+`.

```
tuple_map = [](auto&& map, concepts::tuple auto&& t)
            { return tuple_apply([m = PP_FW(map)](auto&&... elements)
                                 { return tuple{m(PP_F(elements))}...); },
                                 PP_F(t)); };
```

Fold

Druhá veľmi užitočná funkcia vyššieho rádu je *fold*. Fold dostane binárnu operáciu, počiatočnú hodnotu a n-ticu, a aplikuje operáciu na prvky n-tice tak, že výsledky operácie používa zase ako vstup ďalšieho volania. Budeme uvažovať iba ľavý fold; vytvoriť z neho abstraktnejšiu funkciu, ktorá akceptuje parameter o smere, je triviálne, pôjde iba o pridanie pár podmienok a vytvorenie dvoch zrkadlových verzií rovnakých konštruktov.

Vyžadujeme, aby tieto dva zápisy boli ekvivalentné:

```
int f(int a, int b);

auto result1 = f(f(f(0, 1), 2), 3);
// ~
auto result2 = tuple_foldl(f, 0, tuple(1, 2, 3));
```

Tento príklad by sa mohol zdať ako dostatočná podmienka na vytvorenie korektnej funkcie. Vytvorme ale trochu zvláštnejšie `f`.

```
struct S
{   int x;
    S(int x) : x(x) { std::cout << x; }
    S(const& S) = delete;
    S(S&&) = delete;
};
S f(S, int x);

auto result1 = f(f(f(S(0), 1), 2), 3);
// ~ ?
auto result2 = tuple_foldl(f, S(0), tuple(1, 2, 3));
```

Vytvorili sme triedu, ktorá sa nedá kopírovať, no aj tak je vďaka mechanizmu *copy elision* použiteľná. Copy elision je vynechanie volania konštruktoru v prípade, že na miesto, ktoré očakáva hodnotu, vložíme výraz kategórie *prvalue* rovnakého typu. Výraz s kategóriou prvalue je napríklad volanie konštruktoru alebo volanie funkcie s hodnotovým návratovým typom. Vynechanie volania znamená, že konštruktor vôbec nemusí byť dostupný. Ak si zadáme úlohu takto, je neriešiteľná. Ak `tuple_fold` nie je makro preprocesora, musí akceptovať počiatočnú premennú ako parameter a potom ju nejakým spôsobom forwardovať funkcii `f`. V tomto procese sa nutne stratí prvalue vlastnosť tejto hodnoty. S malou obmenou by sa zadanie splniť už dalo.

```
auto result1 = f(f(f(S(0), 1), 2), 3);
// ~
auto result2 = tuple_foldl(f, []){ return S(0); }, tuple(1, 2, 3));
```

Použijeme mechanizmus jazyka, ktorý dokáže urobiť fold na parameter packu. Bohužiaľ, C++ je schopné robiť fold iba cez operátor. Definujeme si teda vlastný, do ktorého vložíme aj trik s bezparametrickým funktorom.

```
template <typename F, typename I> struct fold_wrap { F&& f; I i; };
template <typename F, typename I> fold_wrap(F&&, I) -> fold_wrap<F, I>;

template <typename F, typename T>
constexpr auto operator||(fold_wrap<F, T> w, auto&& ee)
{ return fold_wrap{PP_F(w.f),
                   [f = PP_FW(w.f), i = w.i, e = PP_FW(ee)]()
                       -> decltype(auto)
                   { return f--(i(), e--); }}; }
```

V capture nemôžeme ukladať referenciu na celý objekt `w`, keďže je parametrom funkcie, teda je dočasný. Na `w.f` a `ee` referencie držať môžeme, tieto objekty existujú pred aj po volaní operátoru `||`. `i` musíme kopírovať, pretože ide o subobjekt dočasného `w`. Kopírovať ho je bezpečné, pretože jeho obsah sú iba referencie a objekt funktoru z predošlej iterácie. Zaujímavé je tu ešte, že pri volaní `f` je potrebné použiť `--` a pri volaní `i` používame operátor volania definovaný vo `forward_wrap`. Zase ide o problém s copy elision: ak by sme volali funkciu nepriamo cez operátor volania, stratili by sme prvalue kategóriu výrazu.

S takto definovaným operátorom je potom implementácia nášho fold veľmi krátka:

```
tuple_foldl =
    [](auto&& ff, auto&& ii, concepts::tuple auto&& t)
    { return tuple_apply(
          [f = PP_FW(ff), i = PP_FW(ii)]
          (auto&&... e) -> decltype(auto)
          { return (fold_wrap{f--, i} || ... || PP_F(e)).i(); },
          PP_F(t)); };
```

Ostatné

Prešli sme dve najzákladnejšie operácie spájajúce funktory a n-tice. S týmito nástrojmi môžeme jednoducho vytvoriť ďalšie operácie na n-ticiach, ako napríklad *zip*, kartézsky súčin, mapovanie n-tice do poľa, zreťazenie a *find*. Implementáciu týchto operácií uvádzať už nebudeme, pretože sa pri nich už nevyskytujú tak zaujímavé nápady.

3.6 `constexpr`

Ďalšia „diera“ v štandardnej knižnici, ktorú *PP* zapĺňa, je chýbajúca podpora `constexpr` v implementácii niektorých konštruktov. Pri ukážkach kódu v tejto časti pôjde iba o výber najzaujímavejších častí, nie o úplnú implementáciu.

3.6.1 Výnimky

V implementáciách nebudeme zotavovať stav objektov pri výnimkách. Označenia `noexcept` sú správne, ale pri volaní funkcií, ktoré môžu vyhodiť výnimku,

nie je zaručený stav objektu, ak sa výnimka vyhodí. Toto rozhodnutie sme urobili s cieľom ušetriť čas na triviálnych častiach projektu. Dbali sme na to, aby sme pri návrhu vytvorili priestor pre správnu implementáciu, ktorá by zvládala aj výnimky.

3.6.2 `static_block`

Kontajnery sú štruktúry, ktoré reprezentujú množinu prvkov jedného typu. Budeme sa zaoberať tými, ktoré držia prvky v súvislej pamäti. Takéto kontajnery si vždy pamäť alokujú vopred, aby sa vyhli zbytočne častým alokáciám. Počtu prvkov, ktoré by sa do danej štruktúry zmestili bez novej alokácie sa hovorí *kapacita*. Ich pamäť môže byť *dynamická* (heap) alebo *statická* (stack).

Najprv sa budeme zaoberať statickou pamäťou. Vytvoríme pomocnú triedu `static_block`, ktorá bude reprezentovať súvislú silno typovanú neinicializovanú statickú pamäť. Úloha konštruovať objekty v tejto pamäti pripadá na užívateľa tejto triedy.

Problém statickej alokácie súvislej pamäti a `constexpr` je, že ak chceme alokovať bez inicializácie, musíme alokovať iba pole bajtov. Ak by sme alokovali ako pole typu, ktorý ukladáme, prvky poľa by sa všetky skonštruovali. Pre prístup do pamäte vytvorenej ako pole bajtov kontajneru potrebujeme `reinterpret_cast`, ktorý v `constexpr` kontexte nie je povolený.

Keďže alokácia cez štandardný alokátor povolená v `constexpr` je, toto obmedzenie obídeme tak, že ak je `static_block` vytvorený v `constexpr` kontexte, alokujeme „dynamicky“ (táto alokácia nebude skutočne dynamická, keďže musí nastať už pri preklade).

Problém je, že na objekte vytvorenom v `constexpr` kontexte sa stále dajú volať funkcie mimo tohto kontextu. Preto si musíme informáciu o tom, ako bol objekt vytvorený, uložiť. To síce narúša bajtovú štruktúru, no to sme nepovažovali v tejto knižnici za prioritu.

```
template <typename T, size_t Count>
class static_block
{   union { alignas(T) char buffer[Count * sizeof(T)];
            T* constexpr_ptr; };
    bool constant_created;
public:
    constexpr static_block() noexcept : buffer(), constant_created(false)
    {   if (std::is_constant_evaluated())
        {   constexpr_ptr = allocator<T>().allocate(Count);
            constant_created = true; } }

    constexpr ~static_block() noexcept
    {   if (constant_created)
            allocator<T>().deallocate(constexpr_ptr, Count); }

    constexpr T* begin() noexcept
    {   if (constant_created) return constexpr_ptr;
        else                  return reinterpret_cast<T*>(buffer); }

    // begin() const, end(), end() const

    constexpr auto count() const noexcept { return Count; } };
```

3.6.3 optional

Prvou štandardnou štruktúrou podporujúcou constexpr, ktorú implementujeme, je optional. Na optional sa dá pozerať ako na kontajner s kapacitou 1, ktorý alokuje statickú pamäť. Použijeme v ňom teda static_block.

Jeden constexpr problém, ktorý potrebujeme pri optional vyriešiť, je konštruovanie. Ak máme alokovanú pamäť, v ktorej nie je skonštruovaný objekt, môžeme ho na mieste vytvoriť pomocou *placement new*. Tento výraz akceptuje adresu, nič nealokuje a skonštruuje objekt na zadanom mieste. Bohužiaľ, funkcia, ktorá sa pri takom výraze volá, nie je constexpr. Štandard ale ponúka od verzie C++20 funkciu s ekvivalentným chovaním, ktorá constexpr je, std::construct_at.

Rozhranie je podmnožinou toho štandardného, pretože nie všetky metódy boli pri tomto projekte potrebné. Táto ukážka obsahuje iba najpodstatnejšie časti implementácie:

```
template <typename T>
class optional
{   static_block<T, 1> block;
    bool valid;

public:
    constexpr optional() : block(), valid(false) {}
    constexpr optional(TAG, auto&&... args) : block(), valid(true)
    { construct(PP_FORWARD(args)...); }

private:
    constexpr void destroy() { if (valid) get_ptr()->~T(); }

    constexpr void construct(auto&&... args)
    { std::construct_at(get_ptr(), F(args)...); } };
```

Zaujímavý je konštruktor, ktorý prijíma ako prvý parameter typ TAG. Tento typ je prázdny a slúži iba na odlíšenie od copy a move konštruktorov pri overload resolution. To dovoľuje užívateľovi vynútiť konštruovanie vnútorného objektu, hoci typ argumentu lepšie sedí na iný konštruktor. Je to princíp, ktorý tu nemá veľké využitie, ale využívame ho v celej knižnici pri problémoch s nesprávnym výberom v overload resolution.

3.6.4 unique_pointer

Ďalej vytvoríme pomocnú triedu pre dynamickú pamäť. Najprimitívnejší konštrukt, ktorý na prácu s dynamickou pamäťou štandardná knižnica ponúka je std::unique_ptr. Jeho zmyslom je, že zabezpečuje volanie delete pomocou princípu *RAII*. Tento princíp znamená prakticky to, že zdroje — v tomto prípade pamäť — získavame v konštruktore a uvoľňujeme ich v deštruktore. Tým, že lokálne premenné sa deštruujú na konci ich bloku, dostaneme automatické uvoľňovanie zdrojov. Chybou std::unique_ptr je, že nemá constexpr konštruktory, teda vytvoríme vlastný ekvivalent.

std::unique_ptr má ešte jednu vlastnosť. Tou je unikátnosť v tom zmysle, že každý objekt std::unique_ptr, ktorý spravuje nejakú pamäť, je jej jediným

správcom. Pod pojmom správca rozumieme, že je sémanticky zodpovedný za uvoľnenie danej pamäti.

Tieto dva princípy, RAII a unikátnosť, možno implementovať abstraktne v pomocných triedach.

scoped

Všetky triedy, ktoré využívajú princíp RAII, potrebujú vykonať operáciu, ktorá uvoľňuje zdroje, pri priradení a v deštruktore. Túto operáciu uložíme ako funktor.

```
template <typename T, typename Destructor>
class scoped
{   compressed_pair<T, Destructor> pair;

public:
    constexpr scoped& operator=(const scoped& other)
    {   if (this != &other) { destroy(); //... }
        return *this; }

    constexpr ~scoped() { destroy(); }

private:
    constexpr void destroy() { pair.second(pair.first); } };
```

`compressed_pair` označuje triedu, ktorá funguje ako obyčajná dvojica, až na prípad, kedy je niektorá z obsiahnutých tried prázdna. Vtedy optimalizuje, a táto prázdna trieda v dvojici nezaberá žiadne miesto. Toto je veľmi užitočná optimalizácia, pretože `Destructor` bude v absolútnej väčšine použití tejto triedy prázdna trieda. Vďaka tomu je použitie `scoped` oproti vlastnej implementácii RAII deštruktoru a operátoru priradenia „zadarmo“.

movable

Zmyslom využitia move sémantiky je väčšinou „vykradnutie“ objektu, ktorý je argumentom. `movable` objekt bude teda držať funktor, ktorý pri volaní akceptuje objekt na vykradnutie, vykradne ho a vráti nový objekt rovnakého typu, ktorý drží zdroje pôvodného objektu.

```
template <typename T, typename Releaser>
class movable
{   compressed_pair<T, Releaser> pair;

public:
    movable() = default;
    constexpr movable(const movable& other)
        : pair(other.pair.first, other.pair.second) {}
    constexpr movable(movable&& other)
        : pair(other.release(), move(other).pair.second) {}

    constexpr T release() noexcept { return pair.second(pair.first); }
};
```

Je zopár typov pre `Releaser`, ktoré pokrývajú väčšinu použití `movable`, preto ich implementujeme dopredu.

```
struct default_releaser
{   template <typename T>
    constexpr auto operator()(T& x) { return exchange(x, T()); } };

struct zero_releaser
{   constexpr auto operator()(auto& x) { return exchange(x, 0); } };

struct nullptr_releaser
{   constexpr auto operator()(auto& x) { return exchange(x, nullptr); } };

struct move_releaser
{   constexpr auto operator()(auto& x) { return move(x); } };
```

pointer

Pripomeňme si cieľ: implementovať `constexpr std::unique_ptr`. Štandardne má táto trieda dve verzie: jedna alokuje jeden objekt, druhá alokuje pole objektov. Obe alokácie sa vykonávajú pomocou výrazu `new`. Pre potreby v našej knižnici by sa nám hodila ešte tretia verzia, ktorá by alokovala pomocou alokátoru. Týmto trom verziám alokácie budú zodpovedať tri triedy; spoločne ich budeme nazývať *pointery*. Tieto pointery budú v `unique_pointer` zabalené v `movable` a `scoped` a spolu budú tvoriť v podstate celú jeho implementáciu.

Od pointeru teda chceme, aby: mal na starosti alokáciu, držal ukazovateľ a bol schopný nechať sa vykradnúť. Vykradnutie bude fungovať tak, že žiadame aby sa dal objekt pointeru skonštruovať z `nullptr`, potom vykradneme jednoduchým `std::exchange` za `nullptr`. Alokáciu budú pointery vykonávať v konštruktore a na dealokáciu vyžadujeme členskú funkciu. Tento popis zjavne smeruje ku zavedeniu konceptu, tým sa ale nebudeme zdržiavať.

Vytvorme tri triedy pre tri typy alokácie:

```
template <typename T>
class pointer_base
{   T* ptr;

public:
    explicit constexpr pointer_base(T* ptr) noexcept : ptr(ptr) {}
    constexpr pointer_base(nullptr_t) noexcept : ptr(nullptr) {}
};
template <typename T>
struct pointer_new : public pointer_base<T>
{   constexpr pointer_new(TAG, auto&&... args)
        : pointer_base<T>(new T(F(args)...)) {}

    constexpr void deallocate() { delete this->ptr; }
};
template <typename T>
struct pointer_new_array : public pointer_base<T>
{   explicit constexpr pointer_new_array(size_t count)
        : pointer_base<T>(new T[count]) {}

    constexpr void deallocate() { delete[] this->ptr; }
```

```
};
template <typename T, typename A>
class pointer_allocate : public pointer_base<T>
{   compressed_pair<size_t, A> pair;

public:
    constexpr pointer_allocate(nullptr_t)
        : pointer_base<T>()
        , pair(0, A()) {}

    constexpr pointer_allocate(auto&& allocator, size_t count)
        : pointer_base<T>(PP_F(allocator).allocate(count))
        , pair(count, F(allocator)) {}

    constexpr size_t count() const noexcept { return pair.first; }

    constexpr void deallocate()
    { if (this->ptr) pair.second.deallocate(this->ptr, count()); }
};
```

V tejto ukážke vynechávame schopnosť pointerov vytvoriť sa z pointeru ukazujúceho na objekt potomka, keďže ide o zbytočne dlhú a nezaujímavú časť implementácie.

`unique_pointer`

Implementácia je vďaka vytvoreným konštruktom už triviálna.

```
struct deleter
{   constexpr void operator()(auto& wrapped_ptr) const
    {   auto& ptr = wrapped_ptr.get_object();
        ptr.deallocate();
        ptr = nullptr; }
};
template <typename Pointer>
class unique_pointer
{   scoped<movable<Pointer, nullptr_releaser>, unique_pointer_deleter> p;

public:
    unique_pointer() = default;
    unique_pointer(unique_pointer&& other) = default;
    unique_pointer& operator=(unique_pointer&& other) = default;

    unique_pointer(const unique_pointer&) = delete;
    unique_pointer& operator=(const unique_pointer&) = delete;
};
```

3.6.5 `dynamic_block`

Vytvoríme `dynamic_block`, ekvivalent `static_block`-u pre dynamickú pamäť. Pôjde o neinicializovaný súvislý blok bajtov. Chceme, aby nebol závislý od konkrétneho alokátoru, použijeme teda `pointer_allocate`. Keď ten zabalíme do `unique_pointer`, máme v podstate `dynamic_block` hotový. Pridáme iba metódu na vytvorenie nového bloku s vykradnutím alokátora. Tú využijeme vo `vector`.

```
template <typename T, typename Allocator> class dynamic_block
{   unique_pointer<pointer_allocate<T, Allocator>> ptr;
public:
    constexpr auto spawn_new(size_t count)
    { return dynamic_block(move(ptr.get_object().get_allocator()), count); }
};
```

3.6.6 `vector`

Všetky doteraz vytvorené nástroje spojíme do jednej triedy, constexpr implementácie pre `vector`.

```
template <typename T, typename Allocator = std::allocator<T>>
class vector
{   static constexpr size_t default_capacity = 16;

    dynamic_block<T, Allocator> block;
    movable<size_t, zero_releaser> count_;

    constexpr void destroy_all() noexcept;
public:
    vector(vector&&) = default;
    vector& operator=(vector&&) = default;

    constexpr ~vector() { destroy_all(); }

    constexpr void push_back(auto&&... args);
    constexpr T pop_back();
    constexpr void clear() noexcept;
    constexpr void erase_until_end(const T* i) noexcept;
    constexpr void remove(auto&& predicate); };
```

3.6.7 `small_optimized_vector`

Ako drobné vylepšenie, ktoré ďalej demonštruje užitočnosť vytvorených pomocných konštruktov, môžeme vytvoriť vector, ktorý optimalizuje na malý počet prvkov tak, že ich ukladá do statickej pamäti: `small_optimized_vector`. Veľkosť statickej pamäti dostane trieda ako šablónový parameter. Túto triedu budeme často využívať v mechanizmoch *PP*reflection, keďže množiny, ako je napríklad množina funkcií zvažovaných overload resolution, majú síce teoreticky neobmedzenú veľkosť, no väčšina je veľmi malá, typicky dokonca o veľkosti 1.

Hlavná myšlienka je, že `small_optimized_vector` bude spravovať statickú aj dynamickú pamäť. Do veľkosti, ktorá sa zmestí do statickej pamäti, bude využívať tú; akonáhle je potrebné viac pamäti, prejde do dynamickej. Otázka je, či sa vrátiť do statického bloku, keď veľkosť zase klesne. Veľkosť vectoru pri našom použití nebude klesať nikdy, takže sme sa rozhodli, že do statickej pamäti sa tento vector už nevracia.

Ukážku kódu kvôli stručnosti vynecháme.

3.7 Pohľady

Poslednou veľkou oblasťou, ktorú *PP* pokrýva, sú *pohľady*. Pohľad je abstraktný pojem, ktorý označuje postupnosť prvkov. Oproti kontajnerom je rozdiel ten, že pohľad nemusí *vlastniť* svoje prvky. Väčšinou sú kontajnery zároveň aj pohľadmi.

3.7.1 Koncept

Súvisiaci pojem s pohľadmi sú *iterátory*. Iterátor je objekt, ktorý ukazuje na prvok v postupnosti. V C++ sa dajú definovať tri základné kategórie iterátorov, pričom každá rozširuje predošlú. Najslabšia kategória, *forward*, vyžaduje základné vlastnosti spoločné pre všetky iterátory: „posuň sa na ďalší prvok“ a „vráť prvok“. *Bidirectional* navyše vyžaduje operáciu, ktorá posunie iterátor na predošlý prvok. *Random access* je najsilnejšou kategóriou, tá vyžaduje aj schopnosť posunúť sa o ľubovoľný počet prvkov a možnosť určiť vzdialenosť dvoch iterátorov. Všetky tieto operácie je zvykom volať ako operátory. Definujme si koncepty:

```
template <typename T> concept iterator = requires(T i)
{   ++i;
   { *i } -> non_void;
};
template <typename T> concept iterator_bi = iterator<T> && requires(T i)
{   --i;
};
template <typename T> concept iterator_ra = iterator_bi<T> && requires(T i)
{   i += ptrdiff_t(0);
    { i[ptrdiff_t(0)] } -> non_void;
    i - i;
};
```

Ďalej vytvorme koncept `view`. Od pohľadu žiadame, aby sa na ňom dali zavolať funkcie `begin` a `end` (členské alebo namespace), ktoré nám definujú hranice pohľadu. Ďalej potrebujeme, aby `begin` vracala iterátor. Od `end` by sme mohli tiež vyžadovať, aby vracal iterátor. To je ale zbytočne silná podmienka, ktorú nebudú schopné všetky inak zmysluplné pohľady splniť. Uvažujme najbežnejšie použitie pohľadov: iterácia cez všetky prvky. To, či sme na konci pohľadu, určíme rovnosťou medzi iterátorom začiatku, ktorým pohybujeme dopredu, a „iterátorom“ konca. Na koniec v tomto použití teda vôbec nemusí ukazovať iterátor. Vytvoríme koncept sentinel, ktorý toto chovanie zachytáva:

```
template <typename S, typename I>
concept sentinel = iterator<I> && equatable<I, S>;
```

`sentinel` je binárna relácia iterátoru a ľubovoľného typu, ktorá hovorí, že iterátor možno porovnať s daným typom na rovnosť. Pri iterácii cez pohľad teda stačí vyžadovať, aby koniec vyznačoval iba sentinel iterátoru začiatku.

```
template <typename T> concept view = requires
{   { BEGIN(declval<T>()) } -> iterator;
    { END(declval<T>()) } -> sentinel<decltype(BEGIN(std::declval<T>()))>;
};
```

`BEGIN` a `END` označujú volanie funkcie s menom begin, resp. end, ktorá je buď členská bez parametrov, alebo namespace s jedným parametrom.

Koncept s týmto významom sa nachádza aj v štandardnej knižnici, no v čase začiatku vývoja tohoto projektu ešte nebola implementácia v prekladačoch hotová. Koncept sme ale vytvorili takmer ekvivalentne, takže naše triedy, ktoré spĺňajú view, možno použiť v *range-based* for cykle a všetky štandardné kontajnery spĺňajú náš koncept.

`simple_view`

Najprimitívnejším spôsobom, ako vytvoriť pohľad, je použiť dvojicu iterátorov. Jeden určuje začiatok pohľadu, druhý jeho koniec. Pre takýto pohľad vytvoríme triedu `simple_view`. Niekedy je jednoduchšie vytvoriť namiesto triedy pohľadu iba triedu iterátoru s porovnaním, takže vtedy je možné zabaliť dva také iterátory do `simple_view`, a tým splniť koncept `view`.

3.7.2 Operácie

Je mnoho operácií, ktoré knižnica *PP* na pohľadoch implementuje. V tejto časti ukážeme iba tie tri, ktoré budeme neskôr potrebovať v *PP*reflection: transformácia, zreťazenie a zip.

Všetky tieto operácie budú mať spoločnú vlastnosť: ide o operácie na objektoch pohľadov, nie na prvkoch pohľadov samotných. Tomuto princípu sa hovorí *lazy evaluation*; operácia na prvkoch — ak sa nejaká má stať — sa stane až pri iterácii cez pohľad.

Transformácia

Transformácia pohľadu je lazy mapovanie prvkov pohľadu. Argumentom je pohľad a unárna operácia. Základnou myšlienkou `transform_view` je vytvoriť iterátor, ktorý — podobne ako `movable` a `scoped` — obsahuje `compressed_pair` iterátoru z pôvodného pohľadu a funktoru. Pri dereferencii iba zavoláme funktor. Pohľad z tohoto iterátoru vytvoríme pomocou `simple_view`.

```
template <typename I, typename T>
class transform_iterator
{   compressed_pair<I, T> pair;
public:
    constexpr decltype(auto) operator*() const
    { return pair.second(*pair.first); }

    constexpr void step() { ++pair.first; } };
```

Zreťazenie

Pri zreťazení nebude nastávať žiadna transformácia prvkov, potrebujeme sa len chovať k dvom pohľadom ako k jednému, pričom prvý prvok druhého pohľadu nasleduje za posledným prvkom pohľadu prvého. To docielime zase vytvorením iterátoru a následným zabalením do `simple_view`. Tento iterátor si bude pamätať

začiatok a koniec prvého pohľadu a začiatok druhého. Najprv posúva začiatok prvého, ak sa rovná začiatok a koniec prvého pohľadu, začne posúvať začiatok druhého.

```
template <typename I, typename E, typename J>
class view_chain_iterator
{   I i; E e; J j;
public:
    constexpr void step()
    {   if (i != e) ++i;
        else        ++j; }
};
```

Zip

Zip akceptuje n pohľadov a vytvorí jeden pohľad na n-ticu. Zase vytvoríme iba iterátor, ktorý zabalíme do `simple_view`. Iterátor si bude pamätať n-ticu začiatkov pohľadov. Krok dopredu posunie postupne každý iterátor v n-tici. Dereferencia mapuje n-ticu iterátorov na n-ticu dereferencovaných prvkov. Implementácia tejto triedy je veľmi jednoduchá vďaka vopred vytvoreným operáciám na n-ticiach.

```
template <typename... I>
struct zip_iterator : tuple<I...>
{   constexpr auto operator*() const { return der + *this; }
    constexpr void step() { tuple_for_each(ipr, *this); }
};
```

`der` a `ipr` označujú funktory pre `operator*` a `operator++`, respektíve.

3.7.3 `any_iterator`

Pomôckou, ktorú budeme často využívať v *PP*reflection, je `any_iterator`. Je to trieda, ktorá poskytuje jednotný objekt pre iterátory rovnakej kategórie vracajúce (takmer) rovnaký typ.

Napríklad, do `any_iterator<iterator_category::bi, int&>` chceme priradiť iterátory z `std::list<int>` aj `std::set<int>`.

Využijeme základný princíp objektovo orientovaného programovania: dedičnosť. Iterátory na rôzne kontajnery spoločného predka nemajú, ale môžeme si vytvoriť obalovú šablónovú triedu pre každý typ iterátoru, ktorej jedného predka určiť môžeme. Tento princíp sa v C++ nazýva *type erasure*. Spoločný predok bude predpisovať očakávané virtuálne funkcie, ako je napríklad „krok dopredu“ alebo dereferencia. Samozrejme, dereferencia je trochu problematická, pretože nevieme, aký typ vrátiť. Preto bude predok šablónový, podľa typu, ktorý vracia dereferencia. Navyše, existuje viacero kategórií iterátorov, pričom každá rozširuje rozhranie predošlej — preto bude druhý parameter kategória.

```
template <IC Category, typename T> class AI_BASE {};

template <typename T>
struct any_iterator_base<IC::fw, T> : AI_BASE<IC::fw, T>
```

```
{   constexpr virtual T dereference() const = 0;
    constexpr virtual void increment() = 0;
    constexpr virtual bool equal(const AI_BASE&) const = 0;
    constexpr virtual ~any_iterator_base() {}
};
template <typename T>
struct any_iterator_base<IC::bi, T> : any_iterator_base<IC::bi, T>
{   constexpr virtual void decrement() = 0;
};
template <typename T>
struct any_iterator_base<IC::ra, T> : any_iterator_base<IC::ra, T>
{   constexpr virtual void advance(ptrdiff_t) = 0;
    constexpr virtual T index(ptrdiff_t) const = 0;
    constexpr virtual ptrdiff_t diff(const any_iterator_base&) const = 0;
};
```

Vytvorenie obalu pre konkrétny typ iterátoru je už celkom priamočiare. Najprv treba určiť jeho kategóriu a typ dereferencie, tak určíme predka. Potom už len treba implementovať virtuálne funkcie. Malým problémom je implementácia binárnych funkcií „porovnanie“ a „rozdiel“. Implementujúca trieda by mohla použiť `dynamic_cast` na `other` aby zistila, či je rovnakého typu; ak je, tak môže porovnať. Tieto funkcie by normálne mali fungovať aj na rôzne typy. Toto umožníme tak, že do obalových tried pridáme šablónový parameter pack, ktorý bude značiť iné kompatibilné iterátorové typy. Obal potom prejde okrem svojho typu aj tieto typy a postupne vyskúša `dynamic_cast`.

`any_iterator` bude teda obsahovať `unique_pointer` na predka, ktorý bude v skutočnosti ukazovať na nejaký obal konkrétneho iterátoru. Operátory, ktoré sa od neho očakávajú, budú volať virtuálne funkcie cez `unique_pointer`.

`any_view`

Kvôli osobitostiam triedy `any_iterator`, špeciálne kvôli nutnosti menovať iné porovnateľné typy a dynamickej alokácii, nie je vhodné `any_iterator` vkladať do `simple_view`. Preto vytvoríme samostatnú triedu `any_view`, ktorá bude požiadavky pohľadu spĺňať trochu sofistikovanejšie.

3.8 `ostream`

V projekte budeme potrebovať vypisovať mená deskriptorov na výstup. Cieľ nám zadáva označiť čo najviac funkcií `constexpr`. `std::ostream` `constexpr` nie je, preto si vytvoríme vlastné rozhranie. Použitie `any_view` pri výstupe nie je optimálne, efektívnejšie by bolo predávať text napríklad ako ukazovateľ na znaky. Použitím `any_view` ale získavame možnosť predať funkcii ľubovoľný pohľad na znaky.

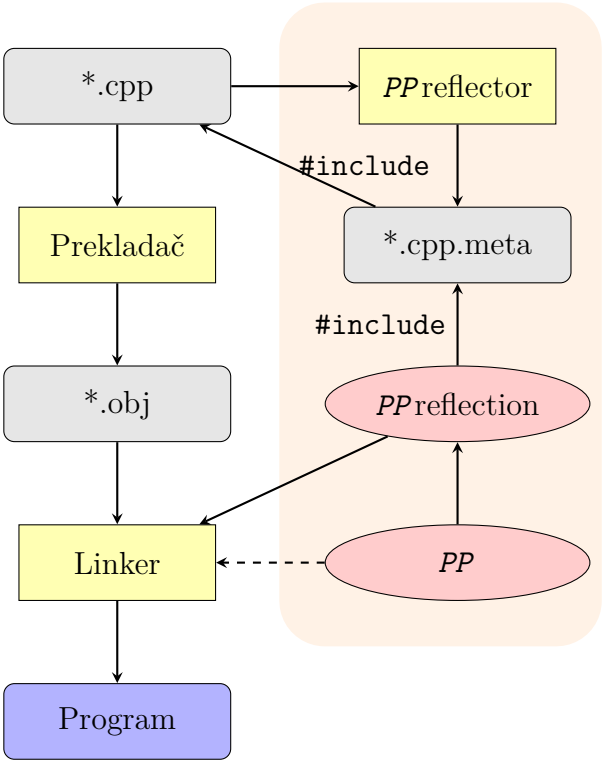
```
struct ostream
{ constexpr virtual void write(any_view<IC::fw, char>) noexcept = 0; };
```

4. Návrh a implementácia

V tejto kapitole urobíme návrh celého projektu. Pri netriviálnej implementácii uvedieme aj zdrojový text. Pôjde predovšetkým o vyriešenie problémov zadaných cieľom práce.

4.1 Architektúra

Aby sme obišli potrebu upravovať prekladač, upravíme jeho vstup. Projekt rozdelíme na dve časti: knižnica *PP*reflection a generátor *PP*reflector. *PP*reflection sa použije ako statická knižnica a *PP*reflector bude zo zdrojového kódu generovať metadáta, ktoré bude *PP*reflection konzumovať. Vstup do prekladača sa upraví tým spôsobom, že užívateľ do `.cpp` súboru pripíše príkaz `#include` s názvom vygenerovaného súboru.



Obr. 4.1: Proces prekladu s introšpekciou

Na obrázku môžeme v ľavom stĺpci vidieť tradičný postup pri preklade C++. Oranžovým rámčekom sú vyznačené pridané fázy týmto projektom.

Introšpekciu urobíme pre každú prekladovú jednotku zvlášť. Uvažujme nasledujúci príklad dvoch prekladových jednotiek:

```
// X.cpp
namespace N {}
// Y.cpp
namespace N {} namespace M {}
```

Každej entite zodpovedá jeden objekt deskriptoru. Otázkou je, či sa budeme chovať k `N` zo súboru `X.cpp` a `N` zo súboru `Y.cpp` ako ku rovnakej entite. Ak by sme ich chápali ako jednu entitu, mohli by sme spojiť všetky súbory vstupného programu a vygenerovať jedny spoločné metadáta. Tie by sme potom museli pomocou preprocesoru vložiť do každej prekladovej jednotky. Tieto metadáta by sa odkazovali na reflektované entity, takže každá prekladová jednotka by musela obsahovať všetky entity z celého programu. Rozhodli sme sa neísť týmto smerom a pri introšpekcii zvažovať každú prekladovú jednotku samostatne, akoby sama tvorila celý program. To, že medzi prekladovými jednotkami prebieha linkovanie, je pre našu knižnicu transparentné. V uvedenom príklade by sa teda vytvorili celkom tri objekty deskriptorov.

Z rozsahových dôvodov nebudeme reflektovať aliasy a premenné. Nie je žiadny technický dôvod, prečo by to nebolo možné, ale na demonštráciu, že introšpekcia je implementovateľná, bude stačiť reflektovať triedy, namespace-y a funkcie. Doplniť podporu pre chýbajúce entity neskôr by už nemalo byť po návrhovej ani implementačnej stránke náročné.

Reflektovať šablóny je takmer nemožné. C++ sa chová ku šablónam tak, že pre každý zoznam šablónových argumentov, s ktorým bola šablóna použitá, vytvorí prekladač samostatnú entitu. Preto nie sú šablóny entitami v tom istom zmysle, ako sú zvyšné nešablónové entity. Introšpekcia nemôže vopred vedieť, aké šablónové argumenty plánuje užívateľ pre danú šablónu použiť. To by sa mohlo vyriešiť tým, že užívateľ by musel vymenovať všetky používané kombinácie argumentov vopred. S takýmto obmedzením je už introšpekcia šablón veľmi slabá, takže ju nebudeme podporovať a šablónové entity budeme ignorovať.

Ignorácia šablón znamená, že nereflektujeme ani ich špecializácie. To má za následok, že kód, ktorý sám o sebe nie je šablónový, nemožno reflektovať, ak používa konkrétne špecializácie šablón. Tento nedostatok je možné vyriešiť samostatne od problému s introšpekciou šablón ako takých, no z dôvodu rozsahu práce sme ho zatiaľ hlbšie v projekte neadresovali.

4.2 Podoba metadát

Hlavným problémom, ktorý nám cieľ práce predkladá, je zariadiť, aby užívateľ nemusel explicitne registrovať entity jazyka do našej knižnice. To rieši automatická generácia metadát zo zdrojového textu. V tejto časti urobíme návrh podoby vygenerovaných metadát, návrh generátoru sa nachádza v podkapitole 4.9.

Z návrhu štruktúry projektu vyplýva, že metadáta musí tvoriť platný štandardný C++ kód, keďže prekladač nie je upravený. Pre zjednodušenie na chvíľu uvažujme, že jedinými dátami o entitách, ktoré chceme reflektovať, sú ich názvy. Tieto dáta môžeme uložiť do premennej. Keďže ide o informáciu známu za prekladu, premenná môže byť `constexpr` (a navyše aj `inline`).

```
// dummy.cpp
struct S {};
struct T {};

#include "dummy.cpp.meta"
```

```
// dummy.cpp.meta
constexpr inline auto S_name = "S"sv;
constexpr inline auto T_name = "T"sv;
```

Z tohoto príkladu môžeme vidieť, že by bolo veľmi prirodzené vedieť sa odkazovať na metadáta o type cez samotný typ, namiesto názvu premennej. To sa samozrejme dá docieliť špecializáciou šablónovej premennej.

```
template <typename T> constexpr inline auto name = /* ? */;

template <> constexpr inline auto name<S> = "S"sv;
template <> constexpr inline auto name<T> = "T"sv;
```

Ak o tomto reflektovaní uvažujeme ako o mapovaní, v príklade mapujeme typ na hodnotu „meno“. Ak by sme chceli mapovať namespace, aký kľúč použijeme v kóde? Mohli by sme zachovať princíp, že na metadáta mapujeme *typy* a vytvoriť každému namespace jeden reprezentujúci typ. Takému typu budeme hovoriť *tag*.

```
namespace N {}

// .meta
#define PPR_META template <> constexpr inline

namespace tags { struct N; }

PPR_META auto name<tags::N> = "N"sv;
```

Typy sú sami sebe tagmi, ale aký tag majú funkcie? Tu sa dostávame k jednému nedostatku návrhu projektu. Samozrejme, funkcie môžu dostať — takisto ako namespace — každá svoj typ. Jednoduchšie by bolo funkciám priradiť ako tag ich adresu. Tak ušetríme typ za každú funkciu a navyše nemusíme vyriešiť, ako z tagu získať adresu. Spôsobili sme tým dva problémy. Prvý: adresa nie je typ. To má celkom jednoduché riešenie, vložíme ju do `PP::value`. Druhým je, že nie je možné získať adresu neverejnej funkcie. Toto vyriešime veľmi „surovo“: budeme ignorovať všetky neverejné entity. Toto rozhodnutie nemá až také veľké nepriaznivé následky, ako by sa mohlo zdať, a navyše je jednoducho vratné. Bližšie vysvetlenie dôvodov a následkov ignorovania neprístupných entít sa nachádza v kapitole 7.1. Zatiaľ je iba dôležité vedieť, že naše riešenie nebude úplne korektné, ale je v známom zmysle predchodcom korektného riešenia. Predovšetkým v tom, že je jednoduchšie na implementáciu.

Teraz predpokladajme, že o entitách potrebujeme reflektovať ešte nejaké informácie navyše. Nazvime ich `info`.

```
struct S {};

// .meta
template <typename T> constexpr inline auto name = /* ? */;
template <typename T> constexpr inline auto info = /* ? */;

PPR_META auto name<S> = "S"sv;
PPR_META auto info<S> = "s"sv;
```

Rôznych druhov informácií o entitách bude ešte niekoľko. Bolo by jednoduchšie mať len jednu šablónovú premennú a typ metadát, čo o entite potrebujeme, kódovať transformáciou tagu entity. To sa dá docieliť pomocou šablónových tried, ktoré nazveme *tag šablóny*.

```
struct S {};

// .meta
template <typename> struct name;
template <typename> struct info;

template <typename T>
constexpr inline auto metadata = /* ? */;

PPR_META auto metadata<name<S>> = "S"sv;
PPR_META auto metadata<info<S>> = "s"sv;
```

Ak by sme chceli namiesto mena triedy zistiť jej `info`, stačí zmeniť šablónovú triedu, nie meno premennej. To má pozitívne implikácie pre implementáciu, keďže šablónová premenná nemôže byť na rozdiel od šablónovej triedy argumentom nejakej operácie.

`reflect`

Ako rozhranie pre použitie metadát vytvoríme funkciu `reflect`, ktorá akceptuje tag šablónu špecializovanú tagom a vráti referenciu na príslušné metadáta.

Keďže väčšina použití `reflect` potrebuje získať deskriptor, vytvoríme funkciu `reflect_descriptor`, ktorá akceptuje tag a vráti referenciu na príslušný deskriptor.

4.2.1 Inicializátor metadát

Nešpecializované `metadata` treba nejako inicializovať, no vložili sme tam iba komentár. Aby sme určili, čím by sme mali inicializovať, treba sa zamyslieť, kedy sa k tejto hodnote dostaneme. Pre všetky reflektované entity vytvoríme špecializáciu, takže ak program pristupuje ku hodnote z nešpecializovaného `metadata`, znamená to, že pristupuje k typu informácie, ktorá nie je o entite zaznamenaná, alebo k entite, ktorá nebola reflektovaná. V každom prípade ide o chybu, takže do inicializácie treba dať niečo, čo bude chybu pri prístupe ohlasovať.

Mohli by sme inicializovať objektom prázdneho typu s názvom, ktorý slúži ako správa o chybe. Tento názov sa objaví pri preklade niekde vo výstupe prekladača.

```
struct unreflected_entity_error {};
template <typename T>
constexpr inline auto metadata = unreflected_entity_error{};
```

Problém takého riešenia je, že zlyhanie závisí od použitia premennej `metadata`, tým pádom hlásenie o chybe môže byť vo výstupe prekladača ťažko viditeľné. Ak by sa `metadata` používalo napríklad takto: `metadata<foo<bar>>.f()`, dostali by sme chybu, že `f` nie je členom `unreflected_entity_error`. Tento trik dokonca

ani nezaručuje, že k nejakému zlyhaniu dôjde — hoci je pravda, že `metadata` bude používané tak, že prázdna trieda jeho použitiu určite vyhovovať nebude. Ak by sa náhodou premenná použila iba takto: `auto x = metadata<foo<bar>>`, k žiadnej chybe nedôjde.

Šikovnejšie by bolo, ak by *akékoľvek použitie* nešpecializovaného `metadata` vyvolalo chybu za prekladu. To sa dá docieliť — dokonca s ľubovoľnou správou vo výstupe — pomocou `static_assert`. Naivné riešenie by vyzeralo takto:

```
struct unreflected_entity_error
{ static_assert(false, "unreflected entity"); };

template <typename T>
constexpr inline auto metadata = unreflected_entity_error{};
```

Toto fungovať nebude. Zlyhanie pri preklade dostaneme vždy, pretože prekladač môže bezpečne povedať, že podmienka v `static_assert` bude vždy `false`. Musíme podmienku „skryť“, aby bola závislá od template parametru.

```
template <typename> constexpr inline auto always_false = false;

template <typename T> struct unreflected_entity_error
{ static_assert(always_false<T>, "unreflected entity"); };

template <typename T>
constexpr inline auto metadata = unreflected_entity_error<T>{};
```

Teraz dostaneme prekladovú chybu zo zlyhania `static_assert` iba vtedy, keď program pristúpi ku nešpecializovanému `metadata`. A navyše s jednoznačnou a zrozumiteľnou správou o chybe.

4.2.2 Linkovanie

Hoci sa pri generácii metadát obmedzujeme naraz iba na jednu prekladovú jednotku, na jednom mieste musíme zvažovať aj linkovanie medzi jednotkami. Vytvorili sme princíp tagov, ktorými sa odkazujeme na entity, ktoré nám inak jazyk nedovolí používať ako argument. Týmito tagmi eventuálne špecializujeme rôzne šablónové premenné a funkcie. Ak medzi dvoma prekladovými jednotkami linker nájde rovnakú šablónu špecializovanú rovnakým typom, dôjde k linkovaniu a informácie z jednej prekladovej jednotky nám „prepíšu“ tie z druhej. Aby sme tomu zabránili, potrebujeme, aby tagy boli unikátne typy v každej prekladovej jednotke. To zariadime jednoducho tak, že deklarácie všetkých tagov a tag šablón vložíme do anonymného namespace.

4.3 Rozhranie deskriptorov

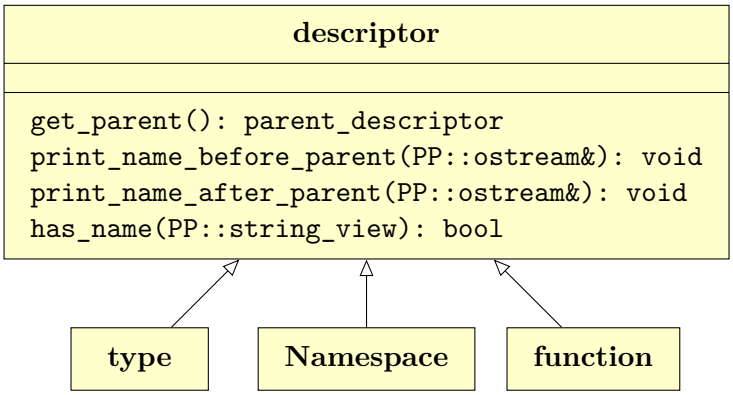
Z cieľa práce vieme, že každej entite zodpovedá objekt deskriptoru. Niektoré typy entít sme sa rozhodli nereflektovať, takže nám ostali tri základné kategórie: typ, namespace a funkcia. Tieto kategórie sú zjavne odlišné: typ je vlastnosť výrazu, funkcie sa dajú volať a namespace je iba „škatuľa“. Vytvorme si teda aj tri

triedy deskriptorov so spoločným predkom `descriptor`. Každá trieda reprezentuje jeden druh entity.

Všetky entity majú spoločné to, že majú nejaké meno. Jedinou výnimkou je globálny namespace, ktorý zatiaľ ignorujme; vrátime sa k nemu pri implementácii. Ak má entita meno `E`, ale nachádza sa v namespace `N`, jej celé meno je `N::E`. Ak ide o funkciu, jej návratový typ prechádza meno rodičov, napríklad `void N::f()`. Preto rozdelíme výpis mena na dve virtuálne funkcie, časť pred a po mene rodiča. Celé meno už zvládne trieda descriptor napísať pomocou týchto dvoch funkcií a mena rodiča.

Entita prislúcha typu iba vtedy, ak ide o užívateľsky definovaný typ. Rozšírime pojem deskriptoru tak, že okrem všetkých entít vytvoríme deskriptor aj všetkým typom. Užívateľsky definované typy sú zároveň typmi aj entitami; dostanú iba jeden deskriptor.

Pre entity dáva vždy zmysel pojem rodič; jedinou výnimkou je zase globálny namespace. Typy, ktoré nie sú užívateľsky definované, napríklad ukazovatele, rodiča nemajú. Potrebujeme vytvoriť jednu triedu, ktorá reprezentuje pojem rodiča deskriptoru. Na to použijeme *sum typ*. Sum typ je typ, ktorý je sémantickým zjednotením viacerých typov. Rodič deskriptoru môže vo všeobecnosti byť namespace, trieda alebo nič. Toto zjednotenie nazveme `parent_descriptor`. Zjednotenie namespace a triedy nazveme `class_or_namespace`. Sum typy budeme implementovať pomocou `std::variant`. V skutočnosti potrebujeme zjednotenia konštantných referencií na uvedené typy, ale `std::variant` referencie neakceptuje. To sa dá vyriešiť zabalením referencie do triedy podobnej `PP::forward_wrap`.



Obr. 4.2: Hierarchia deskriptorov

4.3.1 Rozhranie deskriptorov typov

Všetky typy budú vedieť vypísať svoj názov, ktorý má tiež podľa syntaxe dve polovice, ktoré obe patria za názov rodiča. Tieto dve polovice názvu využijeme pri vypisovaní názvov zložených typov, ako napríklad referencií, kde sa medzi dve polovice názvu odkazovaného typu vloží `(&)`, resp. `(&&)`.

Typy sa v C++ dajú rozdeliť do nasledujúcich disjunktných kategórií:

- referenčné typy - `cv T ref`
- `void`

- funkčné typy - `R(P...) cv ref+ n`
- polia bez rozmeru - `T[]`
- polia s rozmerom - `T[N]`
- ukazovatele - `cv T*`
- ukazovatele na člena - `cv T C::*`
- `nullptr_t`
- floating point typy
- celočíselné typy
- `enum`
- `union`
- triedy

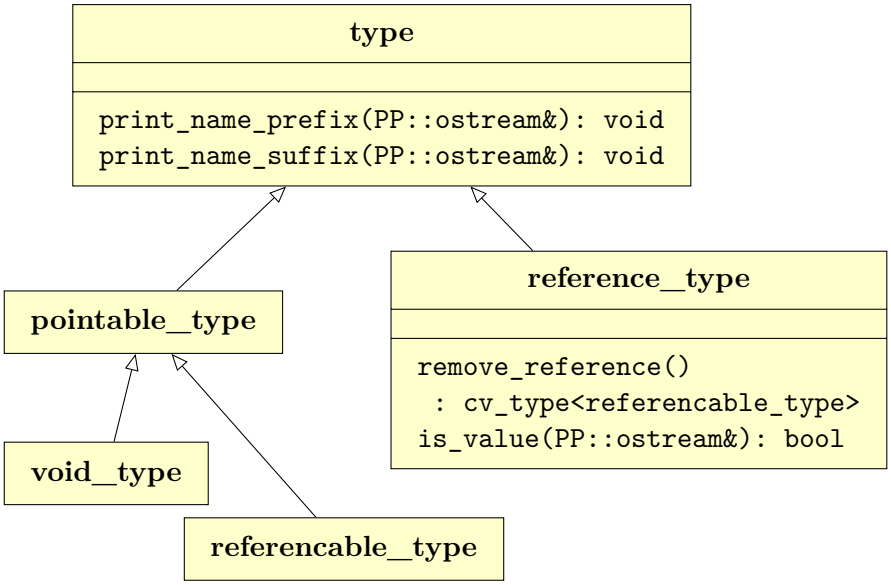
ref označuje takzvaný ref kvalifikátor, čo je `&` alebo `&&`. *ref+* je ref navyše s možnosťou prázdneho slova. *cv* označuje cv kvalifikátor, čo je prázdne slovo, `const`, `volatile` alebo `const volatile`.

Niektoré z týchto kategórií by sa dali rozdeliť na ešte menšie skupiny, ale takéto delenie bude nášmu projektu stačiť. Potrebovali by sme tieto kategórie nejako zmysluplne preložiť do stromu dedičnosti. Na to potrebujeme nájsť nejaké väčšie zoskupenia.

Prvú väčšiu kategóriu by mohli tvoriť typy, na ktoré sa dá vytvoriť ukazovateľ. To sú všetky typy okrem referenčných. Máme teda prvé delenie typov: referenčné vs. ukázateľné.

Všetky referenčné typy musia vedieť vrátiť, na aký typ odkazujú. Nazvime túto kategóriu typov referencovateľné. To sú všetky ukázateľné typy okrem `void`. Problém je, že referencie odkazujú na cv kvalifikovaný typ. Tento princíp sa bude vyskytovať často. Niektoré miesta v jazyku vyžadujú typ, iné dvojicu cv a typ. Pre túto dvojicu si vytvoríme triedu `cv_type` šablónovanú podľa typu. Referencie môžu byť navyše dvoch druhov, lvalue a rvalue.

Ukázateľným a referencovateľným typom zatiaľ nedáme žiadne špeciálne funkcie. Do kategórie `void` patrí iba jeden typ, takže ten tiež nepotrebuje väčšie rozhranie.



Obr. 4.3: Hierarchia deskriptorov typov

Rozhranie referencovateľných typov

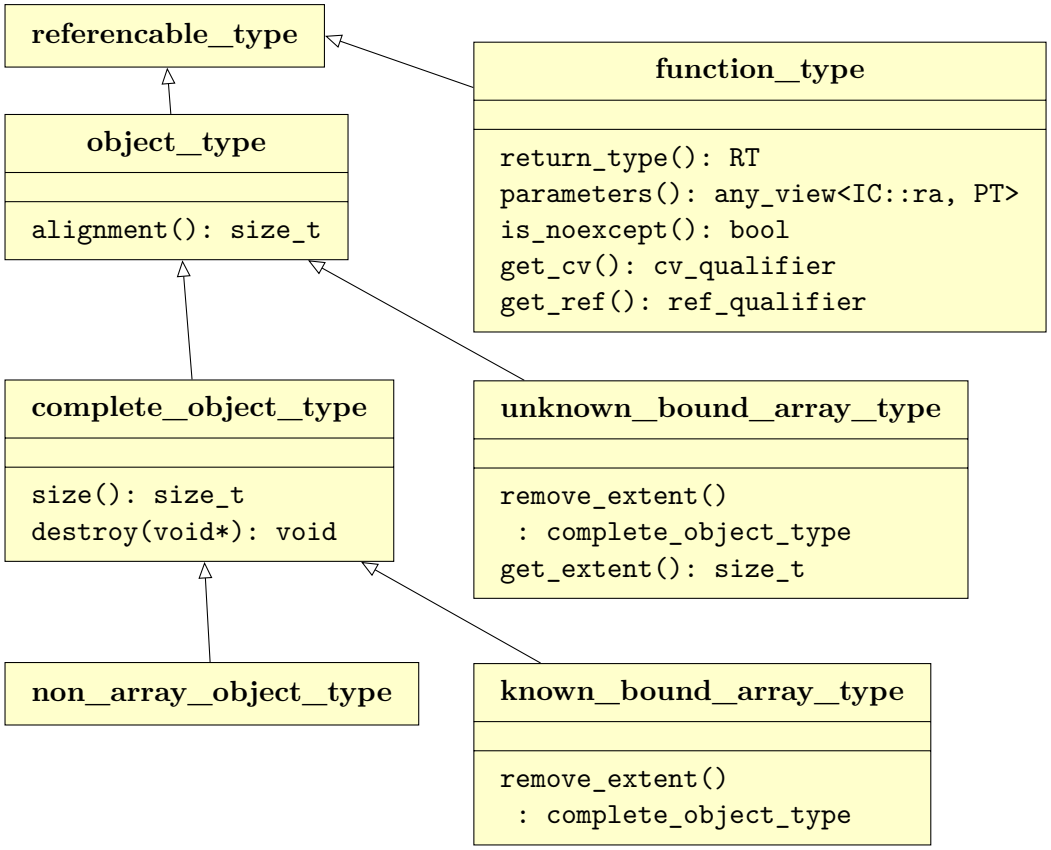
Typy, ktoré môžu vytvárať objekty, sú všetky referencovateľné typy okrem funkčných typov. Funkčné typy vracajú svoj návratový typ, zoznam typov parametrov, svoje cv a ref kvalifikátory a či sú `noexcept`.

Zvláštnou kategóriou sú polia bez rozmeru. Tie sú podľa štandardu *nekompletný typ*. Nekompletné sú tiež triedy medzi svojou deklaráciou a definíciou. Mechanizmus nekompletných typov sme nepreskúmali dostatočne, aby sme ho vedeli korektne implementovať, takže deklarované ale nedefinované triedy ignorujeme.

Všetky typy polí sú podľa štandardu objektovými typmi, ale iba kompletné typy z nich majú veľkosť. Preto vytvoríme kategóriu objektových typov a z nich dediace kompletné objektové typy. Keďže ignorujeme nekompletné triedy, všetky objektové typy v našej hierarchii poznajú svoje zarovnanie. Kompletné objektové typy vedia navyše deštruovať objekt svojho typu.

Keďže sme už vyčlenili polia bez rozmeru, ďalej môžeme vyčleniť polia s rozmerom. Obe kategórie polí budú vedieť vrátiť typ a cv svojho prvku. Prvkom poľa môže byť iba kompletný objektový typ. Polia s rozmerom navyše vracajú svoj rozmer.

Ešte musíme vyriešiť, akej kategórie je návratový typ funkcie. Tu narážame prvýkrát na jeden ťažko riešiteľný problém typového systému C++. Nedá sa z neho vytvoriť stromová štruktúra tak, aby sme zachytili všetky dôležité kategórie. Keďže `constexpr` nepodporuje virtuálnu dedičnosť, ako jediné riešenie nám ostáva zase použiť sum typy. Návratový typ je `void`, referencia alebo kompletný objektový typ okrem polí. Typom parametru môžu byť rovnaké kategórie ako pri návratovom type s výnimkou `void`.



Obr. 4.4: Hierarchia deskriptorov referencovateľných typov

Rozhranie objektových typov okrem polí

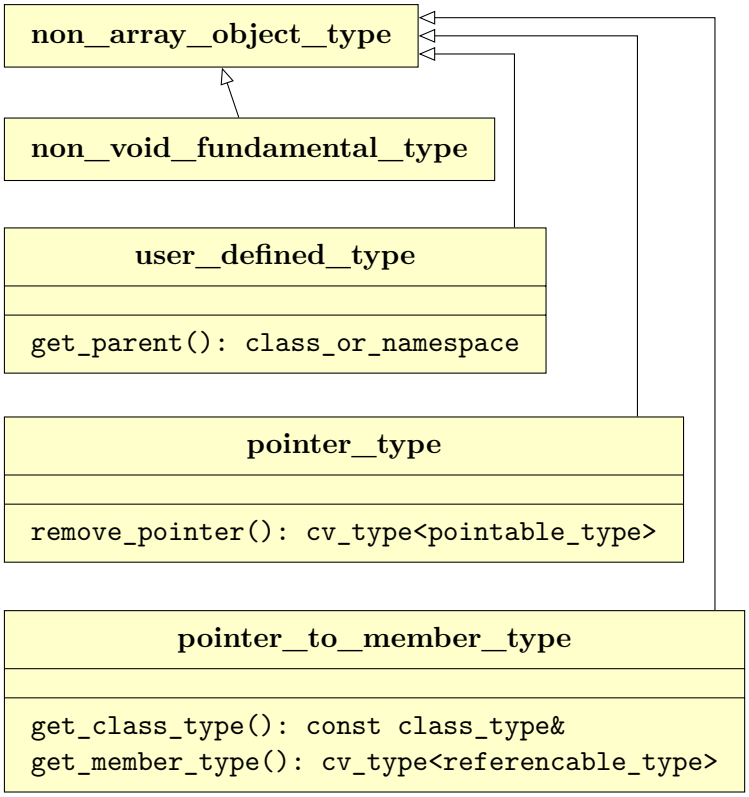
Ostali nám štyri kategórie: ukazovatele, ukazovatele na člena, užívateľsky definované typy a *fundamentálne typy* okrem `void`. Fundamentálne sú typy poskytnuté jazykom, napríklad `int`, `char` alebo `bool`.

Ukazovatele ukazujú na cv kvalifikovaný ukázateľný typ.

Ukazovatele na člena ukazujú na člena určitej triedy, pričom daný člen môže byť akéhokoľvek referencovateľného typu, keďže nemôžeme ukazovať na referenciu a člen triedy nemôže byť `void`.

Užívateľsky definované typy majú za rodiča triedu alebo namespace.

Fundamentálne typy nemajú zatiaľ nijakú pre nás užitočnú spoločnú vlastnosť.

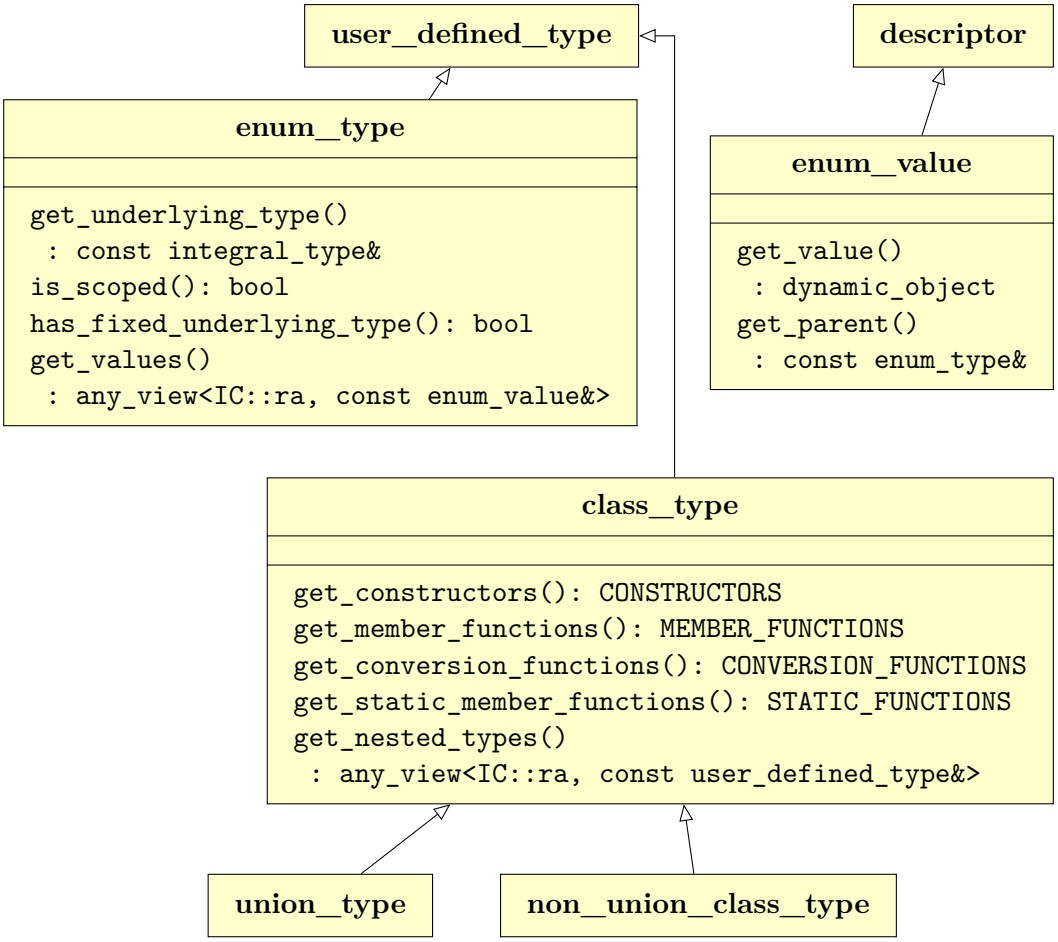


Obr. 4.5: Hierarchia deskriptorov objektových typov okrem polí

Rozhranie užívateľsky definovaných typov

Sú tri základné druhy užívateľsky definovaných typov: triedy, uniony a enumy. V štandarde sa uniony a triedy spolu nazývajú *classes* a triedy, tak ako ich myslíme tu, sú nazvané *non-union classes*. Toto rozdelenie dáva zmysel, pretože väčšina vlastností tried sa aplikuje aj na union. Vlastnosti špecifické pre triedy budeme dopĺňať neskôr, zatiaľ nám postačí do rozhrania vložiť spoločné vlastností pre *classes*. Tie všetky súvisia so získaním nejakého pohľadu na ich členov, ako napríklad získanie pohľadu na členské funkcie.

Zaujímavé sú enumy. Každému enumu zodpovedá zoznam jeho povolených hodnôt, ktorým je priradené meno. Tieto hodnoty budeme tiež reflektovať — priradíme každej vlastný deskriptor. Okrem hodnôt sa dá pri enume hovoriť o jeho *underlying* type, čo je celočíselný typ, v ktorom je uložená hodnota enumu. Enumy môžu byť tiež *scoped*, čo je označené v deklarácii ako `enum` vs. `enum class`, a môžu mať *fixed underlying* typ, čo znamená že underlying typ je explicitne stanovený.



Obr. 4.6: Hierarchia deskriptorov užívateľsky definovaných typov

`reflect`

Na príjemnejšiu prácu s reflektovaním typov, vytvoríme v `type` statickú funkciu `reflect`, ktorá určí z typu argumentu správny návratový typ podľa kategórie, do ktorej typ spadá. To znamená, že táto funkcia môže byť použitá pred vytvorením metadát, pretože jej návratový typ nezávisí od definície tejto funkcie.

4.4 Dynamické štruktúry

4.4.1 Referencia

S hotovým typovým systémom môžeme implementovať centrálnu triedu pre prístup k hodnotám skrz introšpekciu — `dynamic_reference`. Táto trieda bude symbolizovať slabo typovanú referenciu, s možnosťou priradenia, teda zmeny toho, na čo odkazuje. Tým sa skôr podobá na referencie napríklad v C#.

Referencie môžu odkazovať v C++ na objekty alebo funkcie. Potrebovali by sme jeden typ, ktorý by dokázal odkazovať na obe. Naivne by sme mohli použiť `void*`, keďže do takého typu môžeme priradiť adresy všetkých objektov (ignorujúc potrebný `const_cast`). Adresy funkcií by sme mohli pretypovať. To je ale nesprávne, pretože štandard nám takúto konverziu nedovoľuje. Dovoľuje nám

pretypovať ukazovatele na funkciu medzi sebou. Vytvoríme teda sum typ `void*` a `void(*)()`.

Ďalej si dynamická referencia potrebuje pamätať, akého typu je objekt alebo funkcia, na ktorý ukazuje. Keďže potrebujeme byť schopní meniť, kam ukazuje, nemôžeme túto informáciu ukladať referenciou. Urobíme to teda ukazovateľom. Keďže vytvárame dynamickú *referenciu*, potrebujeme ukazovať na referencovateľný alebo referenčný typ. Zvolíme ukazovateľ na referencovateľný typ, pretože tak získame schopnosť jednoducho meniť kvalifikátory dynamickej referencie. Pri tejto možnosti si ešte musíme uložiť cv kvalifikátor odkazovaného typu a ref kvalifikátor danej dynamickej referencie.

Chceme, aby bolo možné vytvoriť dynamickú referenciu pomocou *copy inicializácie* z ľubovoľného výrazu:

```
void f(dynamic_reference);
int x = 7;
dynamic_reference r = x;
f(x);
```

To dosiahneme vytvorením takéhoto konštruktoru:

```
constexpr dynamic_reference(auto&& r) noexcept
        : dynamic_reference(&r, type::reflect(PP_DECLTYPE(r)))
{}
```

`PP_DECLTYPE` nám určí `PP::type` z výrazu `r`. Dôležité je upozorniť, že týmto konštruktorom sa nepredlžuje životnosť, teda napríklad takéto použitie vedie na chybný program, keďže ukladáme odkaz na dočasnú premennú:

```
dynamic_reference r = 5;
```

Pre prístup k silno typovanej hodnote vytvoríme funkciu `cast_unsafe`. Časť „unsafe“ v mene signalizuje, že nebudeme nijako kontrolovať korektnosť takého prístupu, funkcia teda iba vykoná `reinterpret_cast`, kvôli čomu táto funkcia nemôže byť `constexpr`. To je hlavný dôvod, prečo nemôže byť `constexpr` ani dynamické volanie funkcií.

Keďže si dynamická referencia ukladá informáciu o cv a ref v premennej, je veľmi jednoduché jej tieto vlastnosti zmeniť. Vytvoríme si na to členskú funkciu `with_cv_ref`, ktorá vráti novú referenciu s inými cv a ref charakteristikami. Táto funkcia bude užitočná pri konverziách, keďže veľkú časť konverzií tvoria tie, kde sa nemení typ a menia sa iba cv alebo ref kvalifikátory.

Ako poslednú schopnosť pridáme triede `dynamic_reference` `visit`. Ako z názvu vyplýva, ide o analógiu `visit` z `std::variant`. Niekedy bude síce užívateľ poznať typ dynamickej referencie, no nie jej cv a ref vlastnosti. Na to vytvoríme funkciu, ktorá akceptuje silný typ a navštevujúci funktor, ktorého úlohou má byť akceptovať všetky možné druhy referencií na daný typ, vrátane všetkých cv kvalifikátorov daného typu. `dynamic_reference` podľa svojich kvalifikátorov vyberie správny `cast_unsafe` a zavolá daný funktor. S rovnakým princípom vytvoríme aj `visit_ptr`, ktorá predpokladá, že dynamická referencia ukazuje na ukazovateľ a urobí výber správneho volania podľa cv ukázaného typu.

```
int x = 4;
dynamic_reference rx = std::move(x);
rx.visit(type<int>, f); // ekvivalentné f(std::move(x))

auto px = &std::as_const(x);
dynamic_reference rpx = px;
rpx.visit_ptr(PP::type<int>, f); // ekvivalentné f(std::move(px))
```

Na `std::move` pri druhom príklade nezáleží, keďže `visit_ptr` sa používa iba pri ukazovateľoch. Použitie je mienené tak, že funktor má akceptovať ukazovateľ hodnotou. Z dôvodu bezpečnosti by sa dala upraviť implementácia, aby najprv skopírovala ukazovateľ a potom volala funktor na dočasnej kópii, pretože pri aktuálnom chovaní dovoľujeme meniť hodnotu v odkazovanom ukazovateli vtedy, keď funktor má referenčný parameter, čo nie je určený zmysel `visit_ptr`. Na zmenu odkazovanej hodnoty je určený `visit`.

4.4.2 Objekt

Pomocou dynamickej referencie sme schopní odkazovať sa na objekt ľubovoľného typu pomocou jednej triedy. Teraz vytvoríme triedu, ktorá dokáže vlastniť objekt ľubovoľného typu.

Podobne ako pri dynamickej referencii si potrebujeme pamätať typ, ten bude tentokrát kategórie kompletný objektový typ. Ako druhé dáta potrebujeme pamäť, v ktorej bude sa držaný objekt nachádzať. Tu sa nevyhneme dynamickej alokácii. Maximum, čo môžeme spraviť je optimalizovať na malé veľkosti objektov. To urobíme tým spôsobom, že ukazovateľ na dynamickú pamäť vložíme do union spolu s poľom bajtov rovnakej veľkosti. V prípade, že držaný objekt sa zmestí veľkosťou a zarovnaním do pamäti, ktorú by inak zaberal ukazovateľ, uložíme ho priamo tam. Inak použijeme dynamickú alokáciu.

Dynamická alokácia musí byť vykonávaná pomocou `operator new`, pretože to je jediný spôsob, ako sa dá štandardne alokovať zarovnaná pamäť.

Vytvoríme konverznú funkciu na dynamickú referenciu. Dynamická referencia teda môže odkazovať aj na dynamický objekt.

Veľmi dôležitou schopnosťou, čo potrebujeme, aby dynamický objekt mal, je vytvorenie takzvanej plytkej kópie, teda vytvorenie nového dynamického objektu s rovnakými dátami. Pri tejto operácii nebude prebiehať žiadne volanie konštruktoru, keďže sa bude iba kopírovať pamäť po bajtoch. Táto operácia je teda vo všeobecnosti nebezpečná, pretože nedôjde k riadnemu začatiu života objektu. My túto operáciu budeme používať iba pri takzvaných *skalárnych* typoch, čo sú typy, kde táto operácia bezpečná je.

Hlavným spôsobom, ktorým chceme dynamický objekt vytvárať, je, že konštruktoru predáme bezparametrický funktor, ktorý vracia hodnotový typ. Volaním tohoto funktoru si dynamický objekt inicializuje svoju pamäť. K tomuto konštruktoru môžeme vyrobiť pomocnú statickú funkciu `create`, ktorá akceptuje typ a argumenty a inicializuje zavolaním konštruktoru.

```
auto a = dynamic_object([](){ return 5; });
auto b = dynamic_object::create(type<int>, 5);
```

```
// inicializácia s~int(5)
```

Výhoda použitia funktoru je dedukcia typu dynamického objektu z návratového typu.

Táto trieda sa bude sémanticky chovať ako objekt, ktorý vlastní. Preto nebude kopírovateľná a bude výhradným vlastníkom daného objektu. Tiež potrebuje v deštruktore volať deštruktor typu. Na tieto dve vlastnosti sme si už pripravili pomôcky: `movable` a `scoped`.

Dynamický objekt môže tiež držať okrem dát objektu aj kód o chybe; to sa bude hodiť pri volaní funkcií. Nikdy tieto dve informácie nebude potrebovať naraz, takže môžu byť v union.

Ďalší špeciálny stav, v ktorom sa môže dynamický objekt nachádzať, je stav `void`. Vtedy je objekt v korektnom stave, no nedrží hodnotu. Takýto objekt vytvoríme napríklad pri volaní funkcie s deklarovaným návratovým typom `void`.

4.4.3 Premenná

Vytvorili sme slabo typované úložiská pre referencie aj pre objekty. Premenné môžu byť vo všeobecnosti oboch druhov. To isté platí pre návratový typ funkcií. Vytvoríme sum typ dynamickej referencie a dynamického objektu, ktorý nazveme `dynamic_variable`. Bolo by tiež možné obe dynamické triedy spojiť už pri návrhu do jednej, čím by sa mohla získať určitá optimalizácia. Zastávame názor, že návrh s dvoma samostatnými triedami a jedným sum typom je prehľadnejší, hoci by toto rozhodnutie malo byť ešte preskúmané pri pokusoch o vylepšenie výkonu.

4.5 Rozhranie deskriptorov funkcií

Na rozdiel od typov, funkcie nie sú v štandarde tak jednoznačne zatriedené do kategórií. Kategórie si musíme vytvoriť sami. V prvom rade sú funkcie dvoch druhov: tie, ktoré sú deklarované v namespace, a tie deklarované v triede. Keďže v našej introšpekcii sa budeme chovať ku konštruktorom ako ku funkciám, funkcie, ktoré súvisia s triedami, sú buď členské funkcie, alebo konštruktory. Aplikujú sa na ne všetky vlastnosti funkcií, jediný rozdiel je v tom, na akých miestach a akým spôsobom sa tieto vlastnosti uplatňujú. Členské funkcie sa ďalej delia na statické a nestatické, pričom nestatické môžu byť ešte špeciálneho druhu: konverzné.

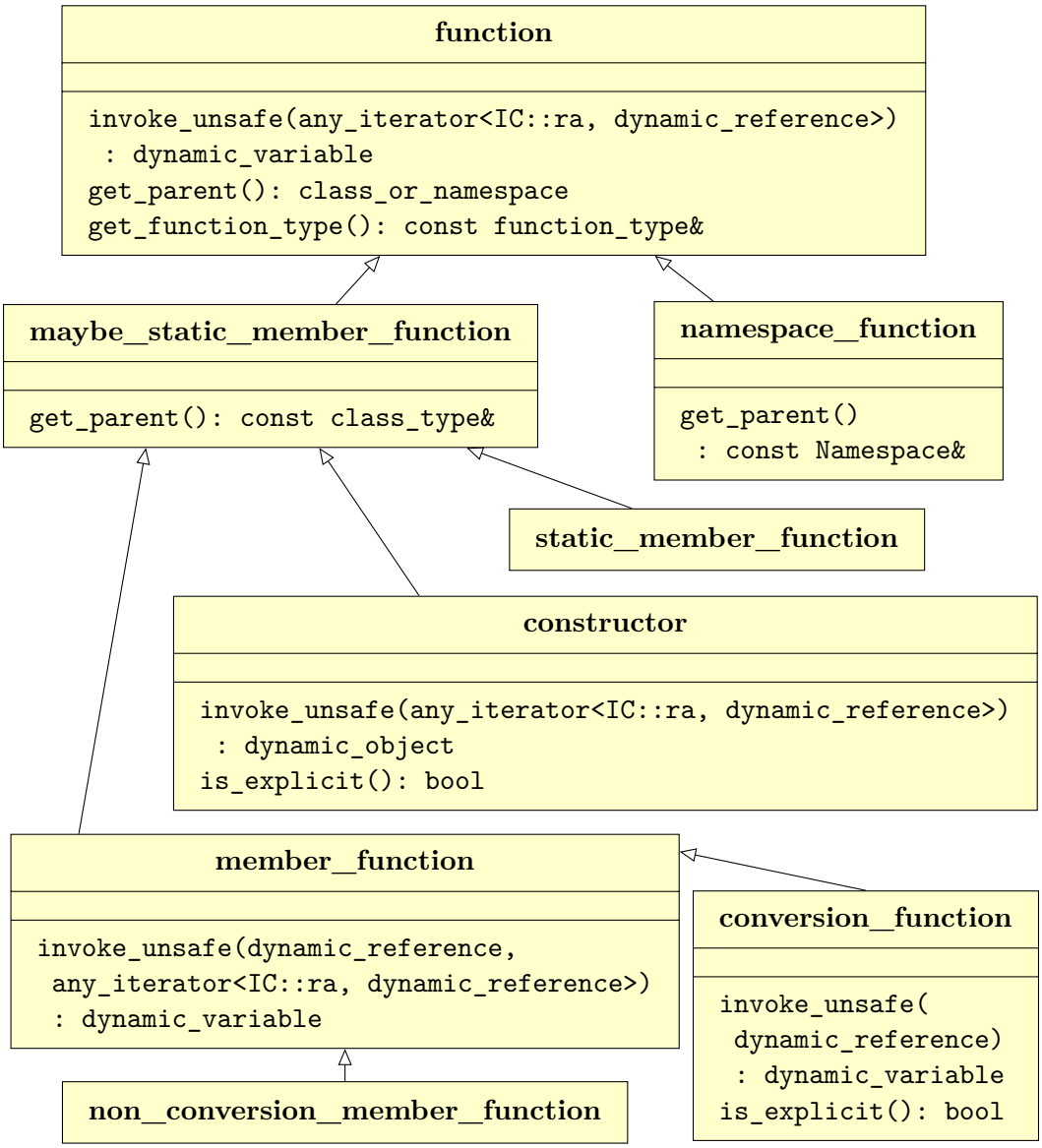
Spoločnou vlastnosťou všetkých funkcií je, že ich môžeme volať. Volanie funkcií v C++ obnáša zopár komplexných mechanizmov, takže tie delegujeme na iné triedy. Deskriptory funkcií budú ako argument volania prijímať iterátor na dynamické referencie, o ktorých predpokladajú, že sú rovnakých typov ako sú parametre. Zodpovednosť za konverziu a kontrolu počtu argumentov má volajúci. Toto rozhodnutie plynie z faktu, že v C++ sa funkcie nikdy nevolajú tak, že užívateľ určí jednu konkrétnu funkciu, ale vždy sa vyberá najlepšia z určitej množiny tzv. kandidátov. Je zbytočné pre deskriptor vytvárať volanie so všetkými korektnými vlastnosťami, pretože ide iba o špeciálny prípad všeobecnejšej operácie volania, keď má množina kandidátov veľkosť 1. Toto volanie nazveme z uvedených dôvodov `invoke_unsafe`.

Vypísanie mena funkcie so sebou nesie svoje špecifiká, ale detaily vynecháme, keďže ide o triviálny návrh.

Namespace funkcie majú za rodiča vždy namespace, zvyšné funkcie majú za rodiča triedu.

Ďalšie rozdiely sú vo volaniach. Konštruktory nikdy nevracajú referenciu ani `void`, takže ich volanie môže vrátiť dynamický objekt. Členské funkcie majú jeden špeciálny argument, objekt triedy, ktorej je funkcia členom. Konverzné funkcie akceptujú iba tento argument, keďže sú bezparametrické.

Konštruktory a konverzné funkcie môžu byť navyše deklarované ako explicitné.

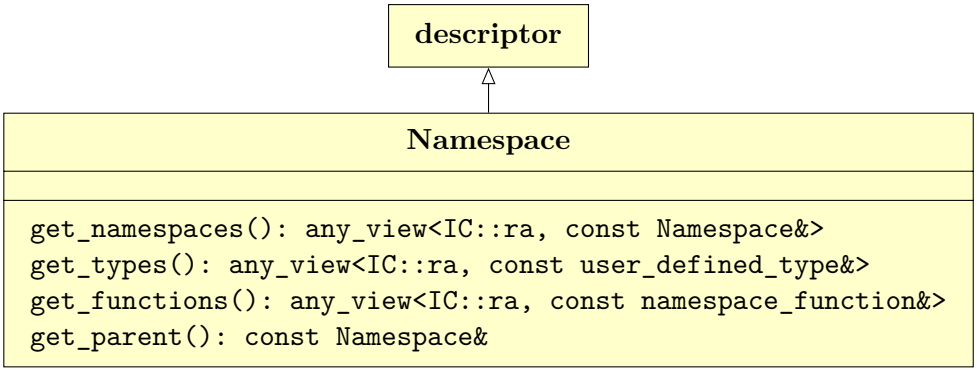


Obr. 4.7: Hierarchia deskriptorov funkcií

4.5.1 Rozhranie deskriptoru namespace

Namespace sú v istom zmysle kontajnery na úrovni jazyka, takže jediné rozhranie ich deskriptoru je poskytnutie zoznamov ich členov. V namespace sa môžu nachádzať iné namespace, užívateľsky definované typy a namespace funkcie.

Rodičom namespace môže byť iba namespace.



Obr. 4.8: Hierarchia deskriptoru Namespace

4.5.2 Kovariancia návratového typu

Pri prepisovaní virtuálnej funkcie platí nasledujúce pravidlo: prepisujúca funkcia má rovnaký názov, rovnaké parametre, rovnaké cv a ref kvalifikátory a rovnaký alebo *kovariantný* návratový typ. V kontexte C++ sú dva typy kovariantné, ak ide o dve referencie alebo dva ukazovatele, pričom jeden smeruje na typ, ktorý je potomkom druhého. Návratový typ prepisujúcej funkcie musí byť podmnožinou návratového typu prepisovanej funkcie. To znamená, že prepisujúca funkcia musí mať návratový typ smerujúci na potomka. Zmysel tohoto chovania je, že špeciálnejšia implementácia funkcie môže mať aj špeciálnejší návratový typ.

Vezmime si príklad funkcií `get_parent` vo `function` a `namespace_function`. function vracia rodiča ako `class_or_namespace` a prepisujúca funckcia vracia `const Namespace&`. Tieto dve funkcie nespĺňajú podmienky jazyka o kovariancii, teda išlo by o chybný program, ak by sme naše hierarchie prepísali do kódu. To je škoda, pretože sémanticky je typ `const Namespace&` podmnožinou typu `class_or_namespace`. Toto je jedno miesto, kde by bola potrebná podpora sum typov od jazyka.

Tento problém vyriešime tak, že tieto funkcie sa prepisovať nebudú. Funkcii z triedy predka pridáme parameter, ktorému určíme default argument. Tak sa pri volaní z kontextu predka dá funkcia zavolať ako bez argumentov. Potom vytvoríme novú virtuálnu funkciu s rovnakým názvom ale bez parametrov. Funkciu potomka implementuje pomocou špeciálnejšej funkcie.

```
template <typename T> struct covariance_tag {};
struct Base
{   virtual sum_type<X, ...> f(P..., covariance_tag<Base> = {}) = 0;
};

struct Derived : Base
{   virtual const X& f(P...) = 0;
private:
    sum_type<X, Y> f(P... p, covariance_tag<Base>) override final
    { return f(PP_F(p)...); }
};
```

4.6 Implementácia deskriptorov

Rozhrania deskriptorov implementujeme ako triedy šablónované podľa tagu danej entity, takže jednej reflektovanej entite bude zodpovedať jedna trieda. Objekt tejto triedy bude jedinečný deskriptor danej entity. Tento objekt uložíme do `metadata`, pod vlastnou tag šablónou `descriptor`.

```
struct S {};

// .meta
PPR_META auto metadata<descriptor<S>> = descriptor_S{};
```

Kruhová závislosť

Príkladom častých metadát, ktoré budeme generovať pre každú entitu je rodič. Navyše vytvorme vymyslené metadáta `children`. Konkrétne také metadáta nebudeme zaznamenávať pri žiadnej entite, no každý rodič musí mať zoznam, respektíve zoznamy svojich detí.

```
struct S { struct T {}; };

// .meta
PPR_META auto metadata<descriptor<S>> = descriptor_S{};
PPR_META auto metadata<descriptor<S::T>> = descriptor_S_T{};
PPR_META auto metadata<children<S>> =
    forward_as_array(reflect_descriptor(PP::type<S::T>));
PPR_META auto& metadata<parent<S::T>> = reflect_descriptor(PP::type<S>)
```

Problém s priamym odkazovaním medzi metadátami je, že musíme najprv vytvoriť deskriptory, aby sa mali metadáta na čo odkazovať. To nám spôsobí nepríjemné poradie pri generovaní. Namiesto toho by sme sa mohli odkazovať cez tagy.

```
struct S { struct T {}; };

// .meta
PPR_META auto metadata<parent<S::T>> = PP::type<S>;
PPR_META auto metadata<descriptor<S::T>> = descriptor_S_T{};
PPR_META auto metadata<children<S>> = PP::type_tuple<S::T>;
PPR_META auto metadata<descriptor<S>> = descriptor_S{};
```

Takto môžeme oddialiť potrebu existencie objektu deskriptoru. Niekde ale stále musíme vytvoriť pole detí. To urobíme v statickej premennej implementačnej triedy deskriptoru, preto potrebujeme, aby vytvorenie deskriptoru dieťaťa predchádzalo vytvoreniu deskriptoru rodiča a aby ostatné informácie o entite predchádzali jej deskriptor. Odkaz na rodiča neuložíme do statickej premennej, ale budeme ho získavať pomocou `reflect` iba v definícii funkcií. Tak prelomíme kruhovú závislosť medzi deskriptormi.

Základná štruktúra

Implementácie budeme nazývať s prefixom `basic_`. Nelistové triedy z hierarchie dostanú šablónovanú implementáciu s parametrom navyše, ktorý bude určovať predka. Väčšina funkcií rozhrania potrebuje jednu informáciu z metadát, ktorú implementujúca funkcia získa pomocou `reflect`.

```
struct base               { virtual I1 info1() = 0; };
struct middle1 : base     { virtual I2 info2() = 0; };
struct middle2 : base     { virtual I3 info3() = 0; };
struct derived1 : middle1 { virtual I4 info4() = 0; };
struct derived2 : middle1 { virtual I5 info5() = 0; };

template <typename Tag, typename Base>
struct basic_base : Base
{ I1 f() override final { return reflect(type<tags::info1<Tag>>); } };
template <typename Tag, typename Base>
struct basic_middle1 : basic_base<Tag, Base>
{ I2 f() override final { return reflect(type<tags::info2<Tag>>); } };
template <typename Tag>
struct basic_middle2 : basic_base<Tag, middle2>
{ I3 f() override final { return reflect(type<tags::info3<Tag>>); } };
template <typename Tag>
struct basic_derived1 : basic_middle1<Tag, derived1>
{ I4 f() override final { return reflect(type<tags::info4<Tag>>); } };
template <typename Tag>
struct basic_derived2 : basic_middle1<Tag, derived2>
{ I5 f() override final { return reflect(type<tags::info5<Tag>>); } };
```

Pohľad na členov

Niektoré funkcie vyžadujú návrat random access `any_view` na deskriptory. Metadáta obsahujú n-ticu tagov, ale vďaka knižnici *PP* môžeme jednoducho mapovať do poľa skrz `reflect`, čím získame kontajner, ktorý nám vie poskytnúť požadovaný pohľad.

```
template <typename Tag> class basic : public base
{   static constexpr auto desc_array =
        tuple_map_to_array(type<const desc&>,
                           reflect_descriptor,
                           reflect(tags::descs<Tag>));
public:
    any_view<IC::ra, const desc&> get_descs() override final
    { return desc_array; }
};
```

`invoke_unsafe`

V `invoke_unsafe` potrebujú implementácie deskriptorov funkcií vytvoriť dynamickú premennú inicializovanú výrazom volania funkcie, pričom potrebujú získať argumenty z iterátoru na dynamické referencie. Najprv potrebuje implementácia konvertovať iterátor na n-ticu jednoduchou obalovou triedou, ktorá deleguje indexovaciu funkciu n-tice na indexovaciu funkciu random access iterátoru. Potom

potrebuje zavolať na túto n-ticu dynamických referencií a n-ticu typov parametrov zip skrz `cast_unsafe`. Tak dostane n-ticu silno typovaných referencií na argumenty. Potom zavolá funkciu s argumentami z n-tice pomocou `PP::tuple_apply`.

```
return dynamic_variable::create([...]() -> decltype(auto)
{   auto args = PP::tuple_zip_with_pack(
        [](dynamic_reference ref, auto t) -> auto&&
        { return ref.cast_unsafe(t); },
        !PP::make_view_tuple(PP::tuple_count_value_t(parameter_types),
                             arg_iterator),
        parameter_types);

    return PP::tuple_apply(PP_F(f), std::move(args));
});
```

`operator!` vytvorí z n-tice referencií n-ticu hodnôt pomocou copy a move konštruktorov. To je potrebné, pretože interne pri volaniach funkcií vzniknú dočasné objekty `dynamic_reference`, ktorých život by skončil skôr, ako potrebujeme.

4.7 Volanie funkcií s konverziami

Volanie funkcie funguje v C++ nasledovne:

1. Vytvorí sa zoznam kandidátov podľa kontextu a spôsobu volania.
2. Filtrujú sa kandidáti, ktorý nemôžu vytvoriť korektné volanie.
3. Zo zvyšných funkcií (*viable functions*), sa nájde maximálna podľa nutných konverzií. Ak existuje práve jedna, volá sa, inak je program chybný.

Tento mechanizmus sa nazýva overload resolution. Vytvorenie kandidátov závisí od kontextu, čo môže byť napríklad meno, ktoré bolo pri volaní uvedené, a namespace, v ktorom sa volanie nachádzalo. Preto prvý krok necháme na volajúceho a zo zvyšných dvoch krokov vytvoríme funkciu `overload_resolution`.

Parametre budú pohľad na kandidátov a pohľad na typy argumentov. Vo všeobecnosti ovplyvňujú rozhodovanie overload resolution aj iné parametre, špeciálne: konverzia, ktorá má nastať na návratovej hodnote, a či sú povolené užívateľsky definované konverzie. Pri dynamickom volaní neuvažujeme konverziu na návratovej hodnote a povolené sú všetky konverzie. Interne ale môžu vznikať volania, ktoré potrebujú tieto parametre nastaviť.

Keďže pri volaní funkcie môžu nastávať konverzie, návratový typ operácie `overload_resolution` nemôže byť iba funkcia. Musíme k nej pribaliť aj konverzie pre každý argument a konverziu pre návratovú hodnotu; spolu vytvoria triedu, ktorá sa bude volať `viable_function`. Predpis funkcie teda bude:

```
viable_function overload_resolution(PP::concepts::view auto&& candidates,
                                    PP::concepts::view auto&& argument_types,
                                    auto&& return_value_sequences,
                                    bool can_use_user_defined)
```

Pri volaní funkcie je nutné inicializovať každý parameter príslušným argumentom. Parameter môže byť referenčný alebo hodnotový, čo vytvára dva veľmi odlišné druhy inicializácie. V štandarde sa tieto druhy nazývajú *reference* inicializácia a *copy* inicializácia. Na proces inicializácie si vytvoríme funkciu s menom `initialization_sequence`, ktorá akceptuje typ parametru, typ argumentu a to, či môže použiť užívateľsky definované konverzie. Typ parametru môže byť referenčný, hodnotový, alebo *implicit object parameter*. Tento tretí špeciálny typ parametru má prvý parameter pri volaní členskej funkcie, teda viaže sa naň objekt, na ktorom sa uskutočňuje volanie. Pre tieto tri druhy parametru vytvoríme sum typ.

`initialization_sequence` vracia takzvanú *(implicitnú) konverznú sekvenciu.*

```
implicit_conversion_sequence initialization_sequence(
    parameter_type_olr_reference target_type,
    const reference_type& initializer_type,
    bool can_use_user_defined)
```

Konverzná sekvencia je postupnosť konverzií, ktoré sa postupne aplikujú na hodnotu jedného typu, aby vznikla hodnota iného typu. Sú dva druhy konverzných sekvencií: *štandardná* a *obyčajná*. Štandardná sa skladá z niekoľko štandardných konverzií, ktoré sú jednoznačne definované štandardom. Obyčajná sekvencia sa skladá, v poradí, zo štandardnej sekvencie, užívateľsky definovanej konverzie a druhej štandardnej sekvencie. Užívateľsky definovaná konverzia je buď konverzná funkcia alebo konštruktor. Vytvoríme si triedy, ktoré budú tieto pojmy zachytávať: `standard_conversion_sequence` a `implicit_conversion_sequence`. Pôjde o veľmi primitívne triedy, ktoré má za úlohu zaplniť ten, kto ich vytvára. Hlavnými funkcionalitami, ktoré implementujú, je vzájomné porovnanie a postupné zavolanie všetkých konverzií v poradí s vytvorením výsledného objektu. Triedy ukladajú informácie nutné na porovnanie konverzií tak, aby bola operácia porovnania čo najjednoduchšia. Celá práca určenia týchto informácii je delegovaná na miesto vytvorenia objektov týchto tried.

Porovnanie konverzií je centrálny mechanizmus celého volania funkcií. Štandard špecifikuje podmienky, za ktorých je jedna konverzná sekvencia „lepšia“ ako druhá. Tak vytvára usporiadanie, na ktorom hľadá overload resolution jedno maximum. Kvôli ignorácii niektorých vlastností jazyka neimplementujeme všetky podmienky porovnania, pretože niektoré sa na našu podmnožinu jazyka neaplikujú. Z tých, ktoré sa aplikujú, sme vynechali len jednu, ide o pravidlo s číslom 12.2.4.3.4.4 v štandarde. Toto pravidlo sa týka konverzií na predka alebo potomka s medzistupňom.

Pri invokácii konverznej sekvencie môže vzniknúť dočasný objekt, ktorého život musí skončiť až po volaní funkcie, takže rozhranie konverzie bude takéto:

```
dynamic_reference convert(dynamic_reference arg, dynamic_variable& temp);
```

Potrebujeme zaplniť štruktúry pre konverzné sekvencie. Užívateľsky definovaná konverzia sa vždy vyberie rekurzívnym overload resolution, takže pri nej pôjde iba o priradenie deskriptoru, ktorý nám vráti OR. Štandardnú konverziu implementujeme ako ukazovateľ na funkciu, ktorá akceptuje dynamickú referenciu a vracia dynamický objekt, respektíve referenciu. Funkcie na vytvorenie týchto

konverzií pridáme do rozhrania deskriptorov typov. Druhy štandardných konverzií sú presne popísané v štandarde. Napríklad, do `function_type` pridáme virtuálnu funkciu `function_noexcept_conversion`, ktorá dokáže odobrať `noexcept` z ukazovateľa na funkciu. v implementácii poznáme typ zdrojovej hodnoty. Vďaka charakteru tejto konkrétnej konverzie vieme vyvodiť aj cieľový typ. Potom použijeme `visit`, respektíve `visit_ptr` z `dynamic_reference` na vytvorenie správnej funkcie.

Pri konverzii aritmetických typov nevieme vyvodiť cieľový typ z deskriptoru. Keďže je iba konečný počet aritmetických typov, môžeme si vytvoriť v kompilačnom čase tabuľku všetkých možných konverzií a za behu pomocou `dynamic_cast` určiť, o aký typ ide, a vybrať jednu z nich.

Zaujímavé sú konverzie viacúrovňových ukazovateľov, keďže pri nich potrebujeme skúmať štruktúru typu do arbitrárnej hĺbky. Vytvoríme štruktúru, ktorá bude zachytávať viacúrovňový ukazovateľ ako postupnosť modifikátorov. Popis tejto postupnosti spolu s pravidlami pre jej porovnanie sa nachádza v štandarde. Táto konverzia, nazývaná *kvalifikačná*, zodpovedá konverzii `const_cast`. Predpokladáme, že pri `const_cast` sa hodnota v pamäti nemení a mení sa iba typ, takže `standard_conversion_sequence` nepotrebuje volať žiadnu skutočnú konverziu, stačí len vytvoriť z dynamického objektu plytkú kópiu a nastaviť jej správny typ.

`candidate_functions`

Pre zjednodušenie používania OR vytvoríme triedu `candidate_functions`, ktorá reprezentuje kandidátov volania funkcie. Dá sa skonštruovať z pohľadu na `const function&` a ponúka funkcie na orezanie množiny, napríklad podľa mena alebo počtu parametrov. Ako svoj hlavný účel ponúka funkciu `invoke`, ktorá na množine spustí OR. Ak OR uspeje, spustí konverziu na každom argumente a zavolá `invoke_unsafe` z deskriptoru. Ak neuspeje, vytvorí dynamickú premennú s neplatným stavom. Túto triedu používa napríklad `Namespace` vo funkcii `invoke_qualified`, ktorá sa pre namespace `N` a názov funkcie `f` chová ako `::N::f(...)`. Táto funkcia interne vyrobí objekt `candidate_functions` zo zoznamu svojich funkcií, ореže množinu podľa mena a zavolá `invoke`.

Trieda `candidate_functions` tiež slúži ako pohľad na `const function&`.

4.7.1 Optimalizácia

OR môže byť veľmi zdĺhavý proces, takže je vhodné umožniť optimalizáciu. Vyhnúť sa OR nemôžeme, keďže iba tak docielime korektné chovanie podľa štandardu jazyka. No keďže OR nezávisí na konkrétnych hodnotách argumentov, iba na ich typoch, môžeme si ho spočítať dopredu. Ak užívateľ zadá niektoré zoznamy typov argumentov, s ktorými chce volanie na množine kandidátov používať, môžeme dopredu spustiť OR. Pri volaní potom stačí iba porovnať, či argumenty majú rovnaké typy ako niektorý z dopredu spočítaných zoznamov. Ak sa nenájde zhoda, stále sa dá záložne zavolať OR pre typy použitých argumentov. Aktuálna implementácia toto záložné riešenie nepoužíva a namiesto toho vráti pri nezhode chybnú dynamickú premennú. Túto funkcionalitu vložíme do triedy `viable_functions`.

4.8 Polymorfná dynamická referencia

Aby sme mohli demonštrovať využitie tohoto projektu, cieľ nám predkladá implementovať dynamický visitor. Najprv si ukážme, ako vyzerá implementácia tradičného visitor návrhového vzoru.

```
struct D1; struct D2;
struct visitor {
    virtual void visit(D1&) = 0;
    virtual void visit(D2&) = 0;
};
struct B {
    virtual void accept(visitor&) = 0;
    virtual ~B() = default;
};

struct D1 : B { void accept(visitor& v) override { v.visit(*this); } };
struct D2 : B { void accept(visitor& v) override { v.visit(*this); } };
```

Ďalej môžeme implementovať triedu visitor a špecifikovať chovanie pre každého potomka.

```
class printing_visitor : public visitor {
    std::ostream& out;
public:
    printing_visitor(std::ostream& out) : out(out) {}
    void visit(D1&) override { out << "D1\n"; }
    void visit(D2&) override { out << "D2\n"; }
};
```

Použitie:

```
void print(B& b, ostream& out) { printing_visitor v(out); b.accept(v); }
print(*make_unique<D1>()); // "D1"
print(*make_unique<D2>()); // "D2"
```

Cieľ od nás vyžaduje toto:

```
struct B {};
struct D1 : B {};
struct D2 : B {};

void print(/* ? */ b, ostream& out) { /* introspection */ }
print(*make_unique<D1>()); // "D1"
print(*make_unique<D2>()); // "D2"
```

Mohli by sme použiť dynamickú referenciu:

```
void print(dynamic_reference b, ostream& out)
{   printing_visitor v(out);
    candidate_functions(type::reflect(PP::type<printing_visitor>)
                        .get_member_functions())
```

```
    .trim_by_name("visit")
    .invoke({v, b});
}
```

Použitie dynamickej referencie má dve veľké nevýhody. Prvou je, že zanáša introšpekciu do predpisu funkcie, teda do rozhrania programu. Lepšie by bolo, ak by použitie introšpekcie bol iba implementačný detail a predpis funkcie obsahoval iba referenciu na predka.

Druhou nevýhodou je, že užívateľ musí držať referencie na objekty v dynamických referenciách, aby nestratil informáciu o dynamickom type. Akonáhle by priradil napríklad do `unique_ptr<B>`, informácia o type by bola stratená. Vyžadujeme teda takýto predpis:

```
void print(B& b, ostream& out) { /* introspection */ }
```

S takýmto predpisom je dynamický visitor neimplementovateľný. Dôvodom je, že objekty `D1` a `D2` nenesú žiadnu informáciu o svojom type, keďže nie sú *polymorfné*. Polymorfný typ obsahuje aspoň jednu virtuálnu funkciu. Na určenie typu z objektu by mohli triedy implementovať takúto funkciu:

```
struct B { virtual type& get_type() = 0; };
struct D1 : B { type& get_type() { return type::reflect(PP::type<D1>); } };
struct D2 : B { type& get_type() { return type::reflect(PP::type<D2>); } };
```

Takto máme všetky požadované informácie, no porušili sme zásadu, že nemôžeme zasahovať do existujúceho kódu. Musíme predpokladať, že podoba daných tried je fixná. Neostáva nám nič iné, ako sa spoľahnúť, že užívateľ poskytol aspoň virtuálny deštruktor, čím sa triedy stávajú polymorfnými.

Z objektu môžeme získať jeho dynamický typ použitím operátoru `typeid` alebo `dynamic_cast`. Obe operácie spolu tvoria mechanizmus jazyka *RTTI*, ktorý niektorý užívatelia C++ v prekladačoch vypínajú. Na tento problém neexistuje riešenie. Buď musíme zasiahnuť do existujúceho kódu a použiť virtuálne funkcie, alebo sa musíme spoľahnúť na RTTI. Keďže `dynamic_cast` vyžaduje jeden konkrétny typ, museli by sme ich skúšať postupne po jednom. Použijeme radšej `typeid`. `typeid` vracia objekt typu `type_info`, ktorý obsahuje informáciu o dynamickom type výrazu. Potrebujeme teda vytvoriť mapovanie na naše deskriptory. To urobíme konverziou na `type_index`, ktorý má definované usporiadanie. Ten bude slúžiť ako kľúč v mape, ktorej hodnoty budú odkazy na deskriptory typov.

Ako rozhranie pre túto mapu vytvoríme v *PP*reflection takúto statickú funkciu:

```
static const non_union_class_type& reflect_polymorphic(std::type_index);
```

Jej definícia sa bude generovať v metadátach. Vytvorí statickú mapu medzi hodnotami `type_index` a deskriptormi a pri zavolaní sa na ňu odkáže. Pre jednoduchšiu prácu si vytvoríme triedu `type_info_map`, ktorá obaľuje `std::map` s príjemnejším rozhraním pre naše použitie. Implementácia `reflect_polymorphic` teda bude vyzerať takto:

```
static const auto map = type_info_map(PP::type_tuple<Classes...>);
return map.get(type);
```

Zoznam všetkých tried vyplní generátor metadát.

Ako posledný krok potrebujeme pripraviť takzvaný *upcast*, čo je pretypovanie v hierarchii dedičnosti smerom ku potomkovi. To môžeme implementovať ako virtuálnu funkciu na deskriptore triedy. So schopnosťou upcast môžeme vytvoriť funkciu, ktorá akceptuje referenciu na polymorfný objekt a vráti dynamickú referenciu ukazujúcu na objekt dynamického typu vstupného objektu:

```
dynamic_reference dynamic_polymorphic_reference(auto&& obj)
{   constexpr auto T = ~PP_DECLTYPE(obj);

    if constexpr (PP::is_non_union_class(T))
        return reflect_polymorphic(typeid(obj))
               .upcast(type::reflect(T))(PP_F(obj));
    else
        return PP_F(obj);
}
```

Teraz môžeme implementovať dynamický visitor:

```
struct B { virtual ~B() = default; };
struct D1 : B {};
struct D2 : B {};

void print(B& b, ostream& out)
{   printing_visitor v(out);
    candidate_functions(type::reflect(PP::type<printing_visitor>)
                        .get_member_functions())
    .trim_by_name("visit")
    .invoke({v, dynamic_polymorphic_reference(b)});
}
```

4.9 Generácia metadát

Kritickou časťou celej architektúry projektu je automatická generácia metadát. Musíme teda vytvoriť program, ktorý na vstupe dostane zdrojový text C++ a zachytí jeho štruktúru v podobe metadát.

V podstate potrebujeme prekladač C++, no s iným výstupom. Dôvody, prečo nevytvárať vlastný prekladač, asi netreba uvádzať; minimálne by to poprelo zmysel doterajšej práce, keďže náš nový prekladač by mohol implementovať jazyk C++ upravený o introšpekciu bez potreby knižnice.

Ale ak nevytvoríme program, ktorý je schopný rozumieť zdrojovému kódu C++, musíme na túto prácu nejako využiť už existujúci prekladač. Clang a GCC oba podporujú možnosť pripojiť počas prekladu vlastný plugin. Nedokázali sme preskúmať, ako by sa dal vytvoriť plugin pre GCC. Clang má dostatočne dobrú dokumentáciu, z ktorej je zrejmé, že sa k nemu dá vytvoriť plugin, ktorý by dokázal generovať metadáta pre introšpekciu. Vytvoríme teda Clang plugin *PP*reflector.

4.9.1 Minimálna implementácia

Potrebovali by sme, aby plugin prechádzal *AST*, ktorý Clang vybuduje. AST je grafová dátová štruktúra, ktorú buduje prekladač zo zdrojového textu. Každej deklarácii zodpovedá vrchol, takže potrebujeme navštevovať vrcholy a na základe informácií v nich uložených vypisovať metadáta.

Na to, aby bol náš plugin v Clang rozpoznaný, musíme vytvoriť určitú minimálnu implementáciu. Musíme vytvoriť triedu, ktorá implementuje rozhranie `clang::PluginASTAction`; nazvime ju `action`. Navyše potrebujeme vytvoriť statickú premennú, ktorá náš plugin registruje.

```
class action : public clang::PluginASTAction
{   std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
        clang::CompilerInstance& CI,
        clang::StringRef) override final
    { return std::make_unique<consumer>(CI); }

    bool ParseArgs(const clang::CompilerInstance&,
                   const std::vector<std::string>&) override final
    { return true; }
    clang::PluginASTAction::ActionType getActionType() override final
    { return clang::PluginASTAction::AddAfterMainAction; }
};

static clang::FrontendPluginRegistry::Add<PPreflector::action> X("", "");
```

`action` potrebuje vytvoriť ukazovateľ na `clang::ASTConsumer`, takže potrebujeme implementovať aj ten.

```
struct consumer : clang::ASTConsumer
{
    explicit consumer(clang::CompilerInstance&) {}
    void HandleTranslationUnit(clang::ASTContext& context) override final
    { /* ... */ }
};
```

`HandleTranslationUnit` sa zavolá pre každú prekladovú jednotku s vygenerovaným AST ako argumentom. Na prechod týmto AST si vytvoríme triedu `visitor`, ktorá implementuje `clang::RecursiveASTVisitor`. Cez tohoto predka dostaneme do `visitor` funkciu, ktorá akceptuje celé AST — to triede `visitor` predá `consumer`, a členskú funkciu, ktorá sa bude volať pre každý vrchol daného stromu. `visitor` tiež použijeme na ukladanie všetkých dát, ktoré potrebujú zotrvať medzi prechodmi cez vrcholy.

```
struct visitor : clang::RecursiveASTVisitor<visitor>
{   /* data */
    explicit visitor(clang::CompilerInstance& ci) {}
    bool VisitDecl(clang::Decl* declaration) { /* ... */ return true; }
};
```

4.9.2 Štruktúra

Teraz máme k dispozícii funkciu `VisitDecl`, ktorá sa zavolá pre každý vrchol AST. Z parametru `clang::Decl` vieme určiť všetky potrebné informácie, ktoré potrebujeme do metadát zaznamenať.

Jeden problém, ktorý ešte máme, je poradie prechádzania vrcholov. To si vo `visitor` môžeme nastaviť na preorder alebo postorder. Bohužiaľ, kvôli tvaru našich metadát nám nevyhovuje ani jedno. Uvažujme takýto vstup:

```
namespace N { namespace X{} }
namespace N { namespace Y{} }
```

Dôležité je si uvedomiť, že vrcholy AST sú deklarácie, nie entity. V AST sa teda bude vrchol o namespace `N` nachádzať dvakrát, pričom prvýkrát bude hlásiť ako svoje deklarácie deklaráciu namespace `X` a v druhom vrchole sa bude nachádzať informácia o vnorenej deklarácii `Y`. Ak by sme vypisovali rovno vo funkcii `VisitDecl`, nedostali by sme nami požadovaný tvar metadát, kde musia všetky vnorené entity predchádzať nadradenú.

Tento problém majú našťastie iba namespace. Definícia triedy síce obsahuje vnorené entity, no môže sa vyskytovať iba raz.

Pre jednoduchosť vytvoríme triedu každému druhu entity, spolu ich nazveme *obaly*. Obaly budú mať na starosti vypísať metadáta o danej entite. Metadáta majú pravidelný tvar, takže sa dá jednoducho vytvoriť spoločné rozhranie všetkých obalov. Nadradený obal bude vždy obsahovať svoje vnorené obaly a rekurzívne na ne najprv zavolá funkciu vypisujúcu metadáta. Na koniec vypíše nadradený obal svoje metadáta. Problém s viacerými deklaráciami jedného namespace vyriešime tak, že `visitor` bude obsahovať mapu medzi deklaráciou namespace a obalom daného namespace. Clang nám dovoľuje pri deklarácii namespace vyžiadať odkaz na prvú deklaráciu. `visitor` bude používať v mape iba túto prvú deklaráciu. Ak zistí, že kľúč sa v mape už nachádza, bude vkladať vnorené obaly do hodnoty, ktorú vráti mapa. Ak sa kľúč v mape nenachádza, znamená to, že sme na daný namespace narazili prvýkrát a treba založiť nový obal, ktorý pridáme do obalu jeho nadradeného namespace. Keďže všetky entity sú nakoniec obsiahnuté v globálnom namespace, stačí ak si `visitor` bude ukladať iba jeho obal.

Na vytvorenie mapy medzi `type_index` a deskriptormi si `visitor` musí zaznamenávať každú triedu, ktorú objaví. Na koniec vypíše ich názvy do šablónového argumentu `type_index_map`.

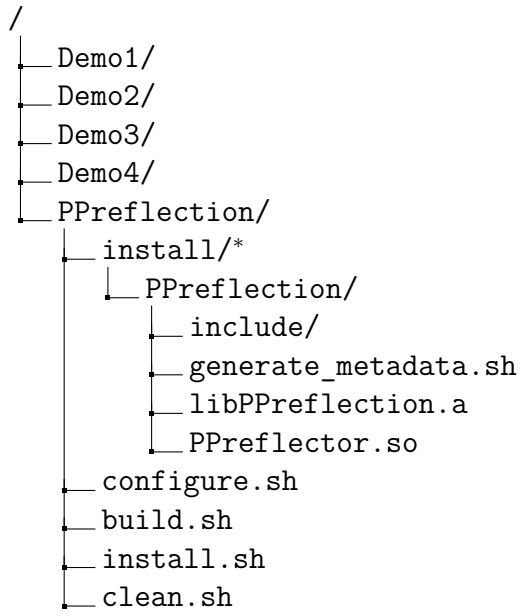
Ukrytie entít

Kvôli nedokonalostiam návrhu je pri prechode vstupného zdrojového textu generátorom definované makro preprocesora `PPREFLECTOR_GUARD`. Pri problematických entitách, ktoré nie je projekt schopný správne reflektovať je možné použiť takýto konštrukt pre ukrytie entít pred generátorom:

```
#ifndef PPREFLECTOR_GUARD
void problematic_function(templated<T>);
#endif
```

5. Použitie

Táto kapitola sa zaoberá vysvetlením použitia celého systému vytvoreného v tejto práci. Vytvorený projekt a príklady použitia sa nachádzajú v prílohe, ktorej štruktúra je nasledovná:



Obr. 5.1: Štruktúra prílohy

5.1 Preklad *PP*reflection a *PP*reflector

5.1.1 Požiadavky

Kvôli použitiu najnovšieho štandardu, C++20, ktorý nie je v prekladačoch naplno implementovaný, je zatiaľ možné preložiť projekt iba pomocou prekladača GCC 11 s argumentom `-std=c++20`. Keďže *PP*reflector je plugin do Clang, na jeho preklad musia byť nainštalované LLVM 12 a Clang 12 vývojové balíčky. Na preklad používame CMake spolu s Ninja build systémom. Skripty sú písane pre shell Bash.

5.1.2 Postup

V adresári projektu `/PPreflection/` sú pripravené skripty `configure.sh`, `build.sh` a `install.sh` určené na automatický preklad. Najprv treba spustiť `configure.sh`. Ten pripraví potrebné súbory pre preklad. Potom spustenie `build.sh` preloží *PP*reflection aj *PP*reflector. Spustenie `install.sh` pripraví inštalačný adresár `install/`. Ten bude potrebný pri demonštračných projektoch.

Pravdepodobná chyba v Clang

Clang pravdepodobne obsahuje chybu vo verzii 12, kde je v jednom hlavičkovom súbore na dvoch miestach pri deklarácii konštruktoru šablónovej triedy

použitý názov triedy vrátane šablónových argumentov. To GCC 11 hlási pri preklade *PP*reflector ako chybu. Riešenie je upraviť hlavičkový súbor a odstrániť šablónové argumenty z deklarácií.

5.2 Preklad projektu s introšpekciou

5.2.1 Požiadavky

Pre použitie knižnice *PP*reflection je nutný štandard C++20. Kvôli použitiu najnovšieho štandardu je preklad projektov používajúcich introšpekciu zatiaľ možný iba s GCC 11. Generácia metadát je vykonávaná spustením prekladača Clang 12.

5.2.2 Príklad

Predpokladajme projekt, kde máme deklarované namespace s vnorenými triedami a chceme vytvoriť funkciu, ktorá pre meno namespace vypíše jeho vnorené triedy.

```
// print_namespace_types.hpp
namespace N { struct A {}; struct B {}; struct C {}; }
namespace M { struct D {}; struct E {}; }

#ifndef PPREFLECTOR_GUARD
void print_namespace_types(std::string_view name, std::ostream& out);
#endif

// print_namespace_types.cpp
#include "print_namespace_types.hpp"

#include "print_namespace_types.cpp.meta"

#ifndef PPREFLECTOR_GUARD
void print_namespace_types(std::string_view name, std::ostream& out)
{ /* introspection */ }
#endif

// main.cpp
#include "print_namespace_types.hpp"

int main()
{   print_namespace_types("N", std::cout);
    print_namespace_types("M", std::cout);
    return 0;
}
```

Detaily implementácie funkcie `print_namespace_types` nie sú pre preklad dôležité.

Do súboru, ktorý používa introšpekciu, je potrebné vložiť pomocou `#include` súbor s metadátami, ktorého názov je celý názov súboru plus prípona `.meta`. `PPREFLECTOR_GUARD` slúži na skrytie častí kódu, ktoré používajú namespace `std` alebo *PP*reflection. Tieto dva namespace generátor metadát nerozpoznáva a nemôže na vstupe dostať kód, ktorý ich používa.

Na preklad potrebujeme adresár `install/PPreflection/`, ktorý sa vytvára pri preklade knižnice. Obsahuje hlavičkové súbory, skript na generáciu metadát, statickú knižnicu *PP*reflection a plugin *PP*reflector. Potrebujeme ho skopírovať do adresára nášho projektu.

Ďalej potrebujeme vytvoriť súbory s metadátami. Súbor `main.cpp` introšpekciu nepoužíva, takže stačí vytvoriť metadáta pre `print_namespace_types.cpp`. To urobíme pomocou skriptu `generate_metadata.sh`. Ten akceptuje ako prvý parameter cestu k *PP*reflector pluginu, v tomto prípade `PPreflection/PPreflector.so`, a ako druhý parameter cestu k súboru, z ktorého treba vytvoriť metadáta.

Keď máme vytvorené metadáta, môžeme začať preklad. Projekt potrebuje linkovať ako statickú knižnicu *PP*reflection. Tiež potrebujeme nastaviť adresár pre objavenie hlavičkových súborov `include/`. Príklad v CMake:

```
add_executable(ExecutableName "main.cpp" "print_namespace_types.cpp")
target_compile_features(ExecutableName PUBLIC cxx_std_20)
target_include_directories(ExecutableName PUBLIC
    "\${CMAKE_CURRENT_LIST_DIR}/PPreflection/include")
target_link_libraries(ExecutableName
    "\${CMAKE_CURRENT_LIST_DIR}/PPreflection/libPPreflection.a")
```

Tento príklad sa nachádza v prílohe pod názvom `Demo1`.

5.2.3 Demonštračné projekty

V prílohe sú priložené tri demonštračné projekty využívajúce introšpekciu a jeden projekt pre .NET.

- `Demo1` ukazuje postup pri preklade s introšpekciou.
- `Demo2` demonštruje schopnosti introšpekcie.
- `Demo3` meria výkon C++ a *PP*reflection.
- `Demo4` meria výkon C#.

`Demo4` obsahuje skripty `build.sh` a `clean.sh` určené na preklad a vymazanie vytvorených súborov, respektíve. Spustiteľný súbor má meno `Demo4` a vytvorí sa v priečinku projektu spustením `build.sh`. Na preklad je potrebný .NET 5.0.

Pred prekladom prvých troch projektov je nutné skopírovať vygenerovaný adresár `/PPreflection/install/PPreflection/` do adresáru projektu.

Každý z troch projektov používajúcich *PP*reflection má vytvorené štyri skripty na ich automatický preklad. Súčasťou skriptu `build.sh` je vždy aj generácia metadát, takže na preklad stačí spustiť skripty v tomto poradí: `configure.sh`, `build.sh`, `install.sh`. Potom sa v priečinku projektu objaví spustiteľný súbor s názvom zhodným s názvom projektu. Na vymazanie súborov vytvorených pri preklade sa používa skript `clean.sh`.

5.3 Dokumentácia

V tejto časti sa nachádza podrobnejšia dokumentácia vybraných dôležitých tried z *PP*reflection.

Kompletnú dokumentáciu celého projektu je možné získať spustením príkazu `doxygen doxy_config` v priečinku `/PPreflection/`. Hlavná stránka HTML dokumentácie sa nachádza v `/PPreflection/doc/html/index.html`.

5.3.1 `dynamic_reference`

Dynamická referencia reprezentuje odkaz na ľubovoľný objekt alebo funkciu.

Rozhranie

```
dynamic_reference(const dynamic_reference&);
dynamic_reference(auto&&);
dynamic_reference& operator=(const dynamic_reference&);
auto get_type() const;
auto cast_unsafe(PP::concepts::type auto t) const -> PP_GET_TYPE(t)&&;
decltype(auto) visit(PP::concepts::type auto, auto&&) const;
decltype(auto) visit_ptr(PP::concepts::type auto, auto&&) const;
```

`dynamic_reference(const dynamic_reference& r)`

Triviálny kopírovací konštruktor, vytvorí referenciu odkazujúcu tam ako `r`.

`dynamic_reference(auto&& r)`

Vytvorí dynamickú referenciu na `r`. Je implicitný, takže nie je nutná explicitná konverzia napríklad pri volaní funkcie, ktorá akceptuje `dynamic_reference`. To využívame pri všetkých funkciách `invoke`, ktoré akceptujú pohľad na dynamické referencie. Vďaka pravidlám jazyka je do týchto funkcií možné vložiť ľubovoľný argument iného typu, ktorý sa automaticky konvertuje na dynamickú referenciu pomocou tohoto konštruktoru.

Je obmedzený tak, že typ `r` nemôže byť dynamická referencia, objekt ani premenná.

`operator=(const dynamic_reference&)`

Triviálne kopírovacie priradenie. Zmení, na čo referencia ukazuje. Referencie v C++ takúto schopnosť nemajú, v tomto zmysle ide o hybrid medzi referenciou a ukazovateľom.

`get_type()`

Vráti deskriptor implementujúci `reference_type`, ktorý označuje typ danej dynamickej referencie.

```
int i;
dynamic_reference r = i;
std::cout << r.get_type(); // "int&"
```

`cast_unsafe(PP::concepts::type auto t)`

Vráti silno typovanú referenciu na objekt odkazovaný dynamickou referenciou. Ak `t` reprezentuje referenčný typ, vrátená referencia je toho typu. Ak `t` reprezentuje hodnotový typ `T`, vrátená referencia je typu `T&&`. Interne využíva `reinterpret_cast`, takže konverzia je korektná, iba ak vedie na rovnaký typ, akého je interne daná dynamická referencia.

```
int i = 4;
dynamic_reference r = i;
std::cout << i.cast_unsafe(PP::type<int&>); // "4"
```

`visit(PP::concepts::type auto t, auto&& f)`

Spustí unárny funktor `f` s argumentom typu reprezentovaného `t` s pridanými cv a ref kvalifikátormi na základe dynamickej referencie.

```
struct f { void operator()(auto&)  { std::cout << "L"; }
           void operator()(auto&&) { std::cout << "R"; } };

int i;
dynamic_reference r = i;
r.visit(PP::type<int>, f{}); // "L"
r = std::move(i);
r.visit(PP::type<int>, f{}); // "R"
```

`visit_ptr(PP::concepts::type auto t, auto&& f)`

Podobné ako `visit`. Predpokladá, že dynamická referencia ukazuje na `cv T*`, pričom `t` reprezentuje `T`. cv sa určí z dynamickej referencie. Spustí `f` s argumentom `cv T*&&`.

```
struct f { void operator()(      auto*) { std::cout << "N"; }
           void operator()(const auto*) { std::cout << "C"; } };

int i;
int* p = &i;
dynamic_reference r = p;
r.visit(PP::type<int>, f{}); // "N"
int* p = &std::as_const(i);
r = p;
r.visit(PP::type<int>, f{}); // "C"
```

5.3.2 `dynamic_object`

Dynamický objekt reprezentuje objekt ľubovoľného typu.

Môže sa nachádzať v troch stavoch: *neplatný*, `void` a *platný*. V stave *neplatný* navyše poskytuje informáciu o druhu chyby, ktorá nastala. Iba v stave *platný* obsahuje nejaký objekt.

Rozhranie

```
dynamic_object();
dynamic_object(dynamic_object&&);
explicit dynamic_object(PP::concepts::invocable auto&&);
dynamic_object& operator=(dynamic_object&&);
cv_type<complete_object_type> get_cv_type() const;
const complete_object_type& get_type() const;
operator dynamic_reference() const;
explicit operator bool() const;
invalid_code get_error_code() const;
bool is_void() const;

static dynamic_object create_void();
static dynamic_object create_invalid(invalid_code);
static dynamic_object create(PP::concepts::type auto, auto&&...);
```

`dynamic_object(dynamic_object&& o)`

Dynamický objekt je výhradným vlastníkom svojej pamäte, takže sa dá iba move konštruovať. Tento konštruktor uvedie objekt `o` do nedefinovaného stavu, takže sa nedá ďalej použiť.

`dynamic_object(PP::concepts::invocable auto&& i)`

Pre `X:=PP_F(i())` tento konštruktor vytvorí dynamický objekt inicializovaný s `decltype(X)(X)`.

`operator=(dynamic_object&&)`

Rovnaká sémantika ako má move konštruktor.

`operator dynamic_reference()`

Vytvorí dynamickú referenciu na dynamický objekt.

`operator bool()`

Vráti `false` práve vtedy, keď je v neplatnom stave.

`get_error_code()`

Ak je v neplatnom stave, vráti kód o stave. Inak vráti `invalid_code::none`.

`is_void()`

Vráti `true` práve vtedy, keď je v stave `void`.

`create_void()`

Vytvorí dynamický objekt v stave `void`.

`create_invalid(invalid_code code)`

Vytvorí dynamický objekt v neplatnom stave s kódom `code`.

`create(PP::concepts::type auto t, auto&&... args)`

Vytvorí dynamický objekt ako `PP_GT(t)(PP_F(args)...)`.

5.3.3 `dynamic_variable`

Reprezentuje dynamickú premennú, sum typ dynamickej referencie a dynamického objektu.

Rozhranie

```
explicit dynamic_variable(dynamic_reference);
explicit dynamic_variable(dynamic_object&&);
dynamic_variable(dynamic_variable&&);
dynamic_variable& operator=(dynamic_variable&&);
explicit operator bool() const;
dynamic_object::invalid_code get_error_code() const;
cv_type<type> get_type() const;
operator dynamic_reference() const;

static dynamic_variable create_void();
static dynamic_variable create_invalid(dynamic_object::invalid_code);
static dynamic_variable create(auto&&);
```

`dynamic_variable(dynamic_reference)`

Vytvorí dynamickú premennú reprezentujúcu referenciu.

`dynamic_variable(dynamic_object&&)`

Vytvorí dynamickú premennú reprezentujúcu objekt.

`dynamic_variable(dynamic_variable&& v)`

Vykradne `v`. Ak `v` obsahovalo dynamickú referenciu, ostane nezmenené. Ak obsahovalo dynamický objekt, daný objekt sa presunie.

`operator=(dynamic_variable&&)`

Rovnaká sémantika ako pri move konštruktore.

`operator bool()`

Ak obsahuje dynamickú referenciu, vráti `true`. Ak obsahuje dynamický objekt, vráti `operator bool` dynamického objektu.

`get_error_code()`

Ak obsahuje dynamickú referenciu, vráti `invalid_code::none`. Ak obsahuje dynamický objekt, vráti chybový kód dynamického objektu.

`get_type()`

Vráti dvojicu cv a typ. Ak obsahuje dynamickú referenciu na `T&&`, vráti `(cv::none, T&&)`. Ak obsahuje dynamický objekt typu `cv T`, vráti `(cv, T)`.

`operator dynamic_reference()`

Vráti kópiu dynamickej referencie, ak obsahuje referenciu. Ak obsahuje objekt, vráti dynamickú referenciu naň. Táto funkcia poskytuje prístup ku objektu, ktorý reprezentuje dynamická premenná.

`create_void()`

Vytvorí dynamickú premennú obsahujúcu dynamický objekt vo `void` stave.

`create_invalid(dynamic_object::invalid_code code)`

Vytvorí dynamickú premennú obsahujúcu dynamický objekt v neplatnom stave s kódom `code`.

`create(auto&& f)`

`f` musí byť funktor zavolateľný bez argumentov.

Ak `f` vracia referenčný typ, vytvorí sa dynamická premenná obsahujúca dynamickú referenciu na návratovú hodnotu volania funktoru.

Ak vracia hodnotový typ, vytvorí sa dynamická premenná obsahujúca dynamický objekt s hodnotou volania funktoru.

Ak vracia `void`, vytvorí sa dynamická premenná obsahujúca dynamický objekt v stave `void`.

5.3.4 `candidate_functions`

Kandidáti reprezentujú množinu funkcií, ktorá sa dá zavolať. Volanie funkcie je v C++ definované v skutočnosti na množine, takže táto trieda dáva to správne rozhranie pre implementáciu korektného chovania podľa štandardu.

Trieda navyše spĺňa koncept pohľadu (na `const function&`).

Rozhranie

```
using IL = std::initializer_list<dynamic_reference>;

explicit candidate_functions(PP::concepts::view auto&&);
candidate_functions& trim_by_name(PP::string_view);
candidate_functions& trim_by_exact_argument_count(PP::size_t);
dynamic_variable invoke(const IL&) const;
dynamic_variable invoke(PP::concepts::view auto&&) const;
```

`candidate_functions(PP::concepts::view auto&&)`

Inicializuje kandidátov z pohľadu na deskriptory funkcií.

`trim_by_name(PP::string_view name)`

Oreže z množiny všetky funkcie, ktoré sa nevolajú `name`.

`trim_by_exact_argument_count(PP::size_t count)`

Oreže z množiny všetky funkcie, ktoré majú iný počet parametrov ako `count`.

`invoke(PP::concepts::view auto&&)`

Ako argument akceptuje pohľad na dynamické referencie na argumenty volania. Interne spustí overload resolution. Ak OR zlyhá, vráti dynamickú premennú obsahujúcu dynamický objekt v neplatnom stave s príslušným kódom o chybe. Ak nezlyhá, vráti výsledok volania vybranej funkcie.

`invoke(const IL&)`

Zavolá predošlý `invoke`. Poskytuje možnosť použiť inicializáciu so zloženými zátvorkami.

5.3.5 `viable_functions`

Realizovateľné funkcie. Reprezentujú množinu funkcií s už spustením overload resolution a pripravenými konverznými sekvenciami. Rozhraním sú podobné ako kandidáti, ide predovšetkým o optimalizáciu.

Rozhranie

```
using IL = std::initializer_list<dynamic_reference>;
template <typename T> concept v~= PP::concepts::view<T>;
template <typename T> concept T = PP::concepts::tuple<T>;

viable_functions(V auto&&, v~auto&&);
viable_functions(V auto&&, T auto&&...);
dynamic_variable invoke(V auto&& arguments) const;
dynamic_variable invoke(const IL& arguments) const;
```

`viable_functions(V auto&& functions, v auto&& argument_lists)`

Akceptuje pohľad na deskriptory funkcií a pohľad na pohľady na deskriptory referenčných typov. Druhý parameter reprezentuje množinu všetkých možných postupností typov argumentov, ktoré chce užívateľ pri volaní používať.

Inicializuje množinu s `functions` a spustí overload resolution pre každý z pohľadov na referenčné typy.

`viable_functions(V auto&& functions, T auto&&... argument_tuples)`

Poskytuje príjemnejšie rozhranie ako predošlý konštruktor. Ako druhý argument akceptuje n-ticu n-tíc referenčných typov.

`dynamic_variable invoke(V auto&& arguments) const`

Zavolá funkciu s `arguments`. Funkcia sa nevyberie pomocou OR, ale iba pomocou porovnania s postupnosťami typov argumentov, ktoré boli pri konštrukcii poskytnuté.

`dynamic_variable invoke(const IL& arguments) const`

Zavolá predošlý `invoke`. Poskytuje možnosť použiť inicializáciu so zloženými zátvorkami.

5.4 Demonštrácia

V tejto časti vytvoríme program demonštrujúci schopnosti knižnice. V prílohe sa tento projekt nachádza pod názvom `Demo2`.

Vytvoríme „polovičný“ visitor návrhový vzor. Predpokladajme nasledujúce triedy:

```
struct B { virtual ~B() = default; };
struct D1 : B {};
struct D2 : B {};
```

K nim vytvoríme visitor:

```
struct visitor
{   void visit(B&);
    virtual void visit(D1&) = 0;
    virtual void visit(D2&) = 0; };
```

V inom súbore vytvoríme implementáciu visitoru. Pre túto ukážku napíšeme do definície triedy aj definície funkcií.

```
struct printing_visitor : visitor
{   std::ostream& out;
public:
    printing_visitor(std::ostream& out) : out(out) {}
    using visitor::visit;
    void visit(D1&) override { out << "D1"; }
    void visit(D2&) override { out << "D2"; } };
```

Potom môžeme vytvoriť `main`, v ktorom použijeme konkrétnu implementáciu `visitor`, ale referencie, ktoré máme k dispozícii sú na typ `B`.

```
std::unique_ptr<B> f(bool x)
{   if (x) return std::make_unique<D1>();
    else   return std::make_unique<D2>();
}
int main()
{   printing_visitor v(std::cout);
    v.visit(*f(true));  // "D1"
    v.visit(*f(false)); // "D2"
    return 0;
}
```

Úlohou je už len implementovať `visitor::visit(B&)`. Použijeme upcast na dynamický typ. Na volanie použijeme `candidate_functions`, navyše v kombinácii s `viable_functions` pre väčšiu optimalizáciu.

```
void visitor::visit(B& b)
{
    static const auto viables = PPreflection::viable_functions(
        PPreflection::candidate_functions(
            Preflection::type::reflect(PP::type<visitor>)
                .get_member_functions())
            .trim_by_name("visit"),
        PP::type_tuple<visitor&, D1&>,
        PP::type_tuple<visitor&, D2&>);

    viables.invoke({*this, PPreflection::dynamic_polymorphic_reference(b)});
}
```

6. Výsledky

6.1 Popis merania

Keďže dynamické volanie funkcií cez introšpekciu je porovnateľné s mechanizmom `dynamic` zo C#, rozhodli sme sa porovnať ich rýchlosť. Oba spôsoby sme navyše porovnali s virtuálnymi volaniami v príslušnom jazyku.

Vytvorili sme teda štyri implementácie s ekvivalentným chovaním. Ide o visitor návrhový vzor. Existuje jedna trieda predka, päť jej potomkov, rozhranie visitoru a jedna jeho implementácia. Na začiatku programu vytvoríme objekty potomkov. Pri každom sa náhodne vyberie, aký z piatich typov objekt bude mať. Tieto objekty vložíme do zoznamu odkazov na predka. V C++ použijeme `unique_ptr`, v C# stačí obyčajná referencia. Potom prejdeme v cykle všetky objekty, postupne ich navštevujeme virtuálnymi volaniami a meriame čas. Potom prejdeme zoznam znovu, tentokrát s dynamickým navštevovaním, a meriame čas.

Implementácie v C++ a C# sa nachádzajú v prílohe pod názvami `Demo3` a `Demo4`, respektíve.

Pri implementácii s *PP*reflection sme vytvorili dva varianty; jeden s optimalizáciou pomocou dopredného vymenovania všetkých možných zoznamov typov argumentov, druhý bez. Pri neoptimalizovanej verzii mal zoznam objektov 100 000 prvkov, pri všetkých zvyšných implementáciách sme prechádzali 10 000 000 prvkov.

Každú implementáciu sme spustili 15-krát, pričom sme zahodili prvé tri merania a potom najrýchlejšie a najpomalšie zo zvyšných 12. Ostalo teda desať meraní, z ktorých sme vytvorili aritmetický priemer času na 10 000 000 volaní.

Programy boli spustené na počítači s nasledovnými charakteristikami:

Operačný systém	Windows 11 Pro 64-bit, WSL2 Ubuntu 21.10
Procesor	Intel Core i7 6700K @ 4 GHz
RAM	16 GB @ 1333 MHz

6.2 Interpretácia výsledkov

Metóda	**Priemerný čas (ms)**	**Nárast**
C++ `virtual`	82,6	—
C# `virtual`	209,2	2,5
C# `dynamic`	813,3	3,9
*PP*reflection (s optimalizáciou)	4691,5	5,8
*PP*reflection	499400,0	106,4

Tabuľka 6.1: Porovnanie rýchlosti *PP*reflection

Ako prvé si môžeme všimnúť, že bez optimalizácie má *PP*reflection takmer nepoužiteľne zlý výkon. To napovedá, že buď sme overload resolution implementovali veľmi slabo, alebo neexistuje implementácia, ktorá by spúšťala OR pri každom volaní a stále mala rozumný výkon.

Po optimalizácii sú výsledky približne 106-krát lepšie, no keď sa pozrieme na ekvivalentnú implementáciu v rovnakom jazyku, stále vidíme viac ako 50-násobné spomalenie. Nie je to fér porovnanie, pretože naša implementácia má v skutočnosti viac schopností ako obyčajné virtuálne volanie, takže isté spomalenie sa očakáva.

Primerané by bolo porovnať ju s `dynamic` v jazyku C#. Tam vidíme iba nárast o faktor 5,8. Je teda možné, že ak by sme sa pokúsili opraviť ešte niekoľko nedostatkov v implementácii knižnice, boli by sme schopní dosiahnuť podobné výsledky ako má C# `dynamic`.

Druhé rozumné porovnanie je medzi nárastami rýchlosti vrámci jedného jazyka. V C# je `dynamic` 3,9-krát pomalší ako virtuálne volanie. *PP*reflection je 56,8-krát pomalšie. Teda, ak by sa naša knižnica zrýchlila 14,5-krát, dostali by sme rovnakú cenu za dynamické volanie v C++ ako v C#.

7. Záver

Vytvorili sme systém pre podporu introšpekcie v C++. Nie je potrebné upravovať prekladač, keďže doplnenie jazyka sa vykonáva generáciou zdrojového textu, ktorý dopĺňa potrebné chýbajúce informácie o entitách jazyka. Návrhovo sa nám podarilo vytvoriť systém tak, že existujúci kód nemusí byť nijako upravený, aby mohol byť reflektovaný. Pri implementácii tohoto cieľa sme narazili na pár problémov, no sú riešiteľné bez zásadnej zmeny návrhu.

Reflektujeme takmer všetky nešablónové entity jazyka. Tie entity ktoré nereflektujeme, sme sa rozhodli ignorovať z dôvodov plynúcich z rozsahu práce, nie kvôli implementačným problémom.

Implementovali sme dynamické volanie funkcií s takmer všetkými súvisiacimi pravidlami jazyka. Ide hlavne o overload resolution a implicitné konverzie. Tieto mechanizmy sú v C++ oproti iným jazykom špeciálne zložité a nenašli sme projekt, ktorý by v C++ implementoval introšpekciu vrátane týchto pravidiel.

Schopnosť spustiť introšpekciu v kompilačnom čase pomocou mechanizmu `constexpr` sme umožnili všade okrem operácií súvisiacich s dynamickým volaním funkcií. Nevyriešili sme problém, či dynamické volanie funkcií je vôbec v princípe možné s aktuálnym štandardom C++20 vykonávať v kompilačnom čase.

Výkon dynamického volania funkcií sme demonštrovali v kapitole 6. Bez optimalizácie, pri ktorej je nutné uviesť všetky potenciálne kombinácie typov argumentov, sme dosiahli slabé výsledky. S optimalizáciou sú výsledky už porovnateľné s ekvivalentným mechanizmom z jazyka C#, obzvlášť keď zvážime priestor, ktorý sme pre ďalšie vylepšenia v tejto oblasti projektu vytvorili.

Celkovo má projekt hneď niekoľko nedostatkov, ktoré ho činia málo použiteľným v praktickej situácii. Pri preskúmaní týchto chýb sme zatiaľ neodhalili principiálny problém s architektúrou alebo návrhom projektu. Veľká časť problémov je pravdepodobne spôsobená tým, že sa pohybujeme na okraji podporovaných schopností jazyka prekladačmi, teda tieto nedostatky by sa mali časom zmierniť ďalším vývojom prekladačov. Veríme, že je možné tento projekt po ďalšej práci uviesť do stavu, v ktorom by mohol byť prakticky aplikovaný.

7.1 Možné vylepšenia

V tejto kapitole rozoberieme nedostatky a možné rozšírenia práce; preto je užitočná najmä pre potenciálnych vývojárov *PP*reflection. Pri nedostatkoch uvedieme ich príčinu a pri rozšíreniach uvedieme motiváciu pre implementáciu. Pri oboch uvedieme navyše aj krátky rozbor riešenia.

7.1.1 Dlhá kompilácia

Najväčším problémom pri aktuálnom stave práce je dlhý kompilačný čas. Na GCC 11.1 trvá preklad jednej prekladovej jednotky používajúcej introšpekciu rádovo *stovky* sekúnd. Na porovnanie: jednotka porovnateľného rozsahu, v ktorej nie je použitý *PP*reflection, sa prekladá pri rovnakých podmienkach niekoľko desatín sekundy. Takýto veľký nárast pri v kompilačnom čase by najskôr prevážil

všetky výhody použitia *PP*reflection v akomkoľvek rozumnom projekte, a preto by oprava tohoto nedostatku mala dostať najvyššiu prioritu pri ďalšom vývoji.

Možných dôvodov má tento problém hneď niekoľko. Prvým by mohla byť súborová štruktúra projektu. Projekt používa veľmi jemné delenie na jednotlivé hlavičkové súbory. To spôsobuje, že prekladač musí často pracovať s diskom. Túto situáciu navyše zhoršuje fakt, že pre každú prekladovú jednotku sa čítajú hlavičkové súbory zvlášť, a teda najskôr dochádza k zbytočnému opakovaniu rovnakej práce, keďže veľa všeobecných hlavičkových súborov je použitých v takmer každej jednotke.

Priamočiare riešenie by bolo spojiť celý projekt do jedného hlavičkového súboru. Tak by sa pre každú prekladovú jednotku čítal iba jeden súbor z disku. Trochu sofistikovanejšie riešenie by bolo použiť moduly z C++20, ktoré riešia problém opakovaného spracúvania rovnakých súborov tak, že ich prekladač vopred preloží. Moduly by sa tiež navyše mohli použiť pri metadátach. Bohužiaľ, podpora modulov nie je zatiaľ dostatočná na hlbšiu analýzu toho, ako by ich použitie mohlo pre tento projekt vyzerať, a čím by ho zlepšilo.

Druhou možnou príčinou dlhotrvajúcej kompilácie môže byť veľké množstvo *template instantiations* (TI). Template instantiation je činnosť prekladača, pri ktorej sa generuje špecializácia šablóny pre konkrétne šablónové argumenty. Každá TI je práca pre prekladač, teda vyšší počet TI znamená dlhší kompilačný čas. TI pochádzajú hlavne z knižnice *PP*, ale tiež z niektorých konštruktov nachádzajúcich sa v *PP*reflection. Väčšinou ide o situáciu, kde je potrebné získať prvok parameter packu na určitom indexe. Na toto knižnica používa rekurzívne riešenia, ktoré pre n-tý index vytvoria $O(n)$ TI. Veľa TI vzniká tiež z dôvodu rozsiahleho použitia `PP::type`, `PP::value` a podobných metaprogramovacích pomôcok. Napríklad, pre jednu premennú typu `PP::tuple<T...>` so `sizeof...(T) == 16` a jeden prístup k 15. prvku sa vytvorí približne 118 TI.

Tento problém by bolo možné najjednoduchšie vyriešiť s podporou prekladača, kde by prekladač poskytol nástroj na získanie prvku parameter packu v konštantnom čase. Zatiaľ sa takýto nástroj nachádza iba v Clang. Problém takého riešenia je tiež nestabilita, keďže nejde o štandardnú funkciu jazyka. Inak by bolo možné na niektorých miestach použiť sofistikovanejšie riešenia, ako je priama rekurzia. `PP::make_value_sequence` je napríklad implementovaný tak, že vytvára iba $O(\log n)$ TI pre zoznam indexov dlhý n.

Ďalší problém spôsobujúci dlhú kompiláciu súvisí taktiež s knižnicou *PP*. Keďže v štandardnej knižnici zatiaľ nie je `constexpr` všetko, čo by mohlo byť, *PP* implementuje svoje verzie štandardných tried, ako napríklad `vector` alebo `optional`. Je veľmi pravdepodobné, že knižnica vytvára neefektívny kód nielen za behu, ale aj z hľadiska kompilačného času, keďže na vývoj *PP* bol vynaložený iba zlomok práce oproti implementáciám štandardnej knižnice.

Dobré riešenie zatiaľ neexistuje, keďže štandard neponúka `constexpr` verzie dôležitých funkcií a konštruktorov. Kým do štandardu nepríde ich podpora, ostáva iba vylepšovať knižnicu *PP*, respektíve použiť inú, ekvivalentnú knižnicu.

7.1.2 Ignorácia neverejných členov

Program ignoruje neverejné členy a neverejných predkov tried. Pri predkoch sa ignoruje fakt, že sú predkami, nie celá trieda predka. Tento nedostatok spôsobuje

nekorektné chovanie pri dynamickom volaní funkcií cez introšpekciu, kde sa môže namiesto korektného zlyhania vybrať nie najlepšia konverzia.

Prístup ku členom funguje v C++ takým spôsobom, že všetky mechanizmy, ktoré ho potrebujú zvažovať, ho uvažujú vždy na konci. Vo všeobecnosti tieto mechanizmy fungujú tak, že sa najprv vytvorí množina kandidátov, z nej sa vyberie najlepší, a ak ten nie je prístupný, program je nesprávny (*ill-formed*). Program sa môže stať ill-formed aj skôr počas daného mechanizmu, to ale neovplyvňuje tento aspekt projektu. Cieľom kontroly prístupnosti na úplnom konci je, aby sa zmenou prístupnosti nemenilo chovanie. Inak povedané, ak pre nejaký zdrojový kód uvažujeme množinu všetkých kódov, ktoré sa z neho dajú vytvoriť iba zmenou prístupností, tak všetky korektné kódy v tejto množine vytvoria po preklade chovaním identické programy.

Pre účely dynamického volania funkcií sa chová *PP*reflection ako prekladač. Predpokladajme, že je implementovaný správne až na ignoráciu neprístupných členov. Potom platí: pre zdrojový text, ktorý je podľa štandardu korektný, sa chová *PP*reflection identicky ako správny prekladač. Dôkaz: Pri korektnom texte by správny prekladač vybral z množiny kandidátov C práve jedného, ktorý by bol lepší ako všetky zvyšné a bol by verejný. *PP*reflection vyberá z $\{c \in C \mid c \text{ verejný}\}$. Keďže kandidát vybraný správnym prekladačom je verejný, zvažuje ho aj *PP*reflection. Keďže predpokladáme, že inak je *PP*reflection správny prekladač, zvolí ho tiež ako maximálny $\square$. Prípad, kedy by správna implementácia nepreložila program je samozrejme otvorený, a kvôli ignorácii prístupnosti môže dôjsť k nesprávnemu chovaniu. Dôležité je, že kvôli tomuto nedostatku sa chová introšpekcia inak ako štandard iba v prípade, že vstup bol chybný.

Dôvodom tejto odchýlky je zjednodušenie implementácie, čím sme získali cenný čas na implementáciu iných zaujímavejších častí projektu. K rozhodnutiu tiež prispel malý dopad na korektnosť a tiež priamočiarosť nápravy rozhodnutia.

Riešením by bolo prestať používať adresy funkcií ako tagy a upraviť konverzie na referenciu na predka tak, aby kontrolovali prístup. To by najskôr vyžadovalo zavedenie dodatočných metadát o prístupe.

7.1.3 Chýbajúca podpora premenných

V projekte chýba podpora pre premenné. Analogicky k ostatným reflektovaným entitám by malo byť možné: enumerovať deskriptory dátových členov danej triedy, získať z objektu a deskriptoru premennej hodnotu ako dynamickú referenciu.

Táto funkcionalita je relatívne nezávislá na iných schopnostiach knižnice, teda nebola implementovaná z časových dôvodov. Premenné by boli dôležité napríklad pri serializácii alebo kompatibilite s existujúcim kódom, ktorý premenné využíva priamo.

Implementácia introšpekcie premenných by mala vyzerať analogicky ku introšpekcii členských funkcií.

7.1.4 Chýbajúca podpora šablón

Podpora pre šablóny nie je úplne implementovateľná z obmedzení daných jazykom. Niektoré vlastnosti by ale aj tak mohli byť reflektovateľné, napríklad

meno šablóny alebo informácie o jej parametroch.

Ak by sme boli ochotní akceptovať, že užívateľ by bol povinný vymenovať všetky kombinácie šablónových argumentov pre každú šablónovú entitu, ktorú by si prial reflektovať, dalo by sa uvažovať aj nad omnoho silnejšou podporou šablón.

Kvôli chýbajúcej podpore pre šablóny nereflektujeme ani ich špecializácie. To má napríklad za následok, že ak funkcia používa ako parameter špecializáciu šablóny, nemôžeme ju reflektovať, pretože knižnica by nemala potrebné údaje o type parametru.

Knižnica nevidí v metadátach rozdiel medzi informáciami o obyčajnej triede a informáciami o špecializácii šablónovej triedy. Úlohou by bolo vyriešiť, ako donútiť generátor produkovať metadáta pre každú špecializáciu.

7.1.5 Atribúty

Jedným, respektíve celou skupinou možných rozšírení, je pridať do projektu podporu pre atribúty. *PP*reflector by ich spracoval a zahrnul do metadát. V knižnici by sa potom mohla rozšíriť funkcionalita deskriptorov na základe týchto metadát.

Toto sú niektoré možné druhy atribútov:

`[[ignore]]`

Inštrukcia pre introšpekciu ignorovať danú entitu.

`[[priority(p)]]`

Nastaví konverznej funkcii prioritu, ktorá sa použije v overload resolution pri nejednoznačných konverziách.

Užívateľsky definované atribúty

Bolo by možné detekovať aj ľubovoľný atribút. Potrebné by bolo vytvoriť nový typ deskriptoru, ktorý by niesol informácie o atribúte. Každý deskriptor by potom vedel enumerovať atribúty svojej entity.

7.2 Ostatné

Táto časť slúži ako enumerácia zvyšných problémov, ktoré sa v projekte nachádzajú. Ide buď o problémy, ktoré majú minimálny dopad na funkciu, alebo o problémy, ktoré sú jednoducho riešiteľné. Poradie ich uvedenia je arbitrárne.

Aliasy

Nepodporujeme introšpekciu aliasov. Jej doplnenie by malo byť triviálne, podobne ako pri premenných.

Nekompletné typy

Nepreskúmali sme fungovanie nekompletných typov, ktoré kvôli tomu ignorujeme. Pravdepodobne by bolo možné implementovať nekompletné typy ako ďalšiu kategóriu deskriptorov.

Move konštruovateľné parametre

Ak je parameter pri dynamickom volaní funkcií hodnotového typu, musí byť move konštruovateľný. Mohli by sme použiť podobný princíp ako v knižnici *PP*, kde namiesto priameho použitia argumentu používame bezparametrický funktor.

Slabá implementácia dynamickej premennej

Dynamická premenná je implementovaná veľmi primitívne, čo môže spôsobovať výkonnostné problémy. Riešením by mohlo byť spojenie dynamických referencií a objektov do jednej triedy.

Ignorácia `std` a *PP*reflection

Nereflektujeme štandardnú knižnicu ani knižnicu *PP*reflection. To má za následok, že nie je možné reflektovať ani funkcie, ktoré používajú triedy týchto knižníc ako parametre. Tiež to spôsobuje, že je nutné vložiť makro preprocesora do reflektovaného kódu. Toto rozhodnutie je preventívne, nie je principiálny dôvod, prečo tieto knižnice nereflektovať.

Zlá optimalizácia dynamického objektu

Dynamický objekt je schopný kvôli optimalizácii ukladať svoj objekt v statickej pamäti. Pri move konštruovaní dynamického objektu vzniká problém, že takýto proces nie je dovolený pre všetky objekty jazyka. Naše pravidlá pre voľbu tejto optimalizácie sú príliš slabé; je nutné zabezpečiť aby bol objekt v statickej pamäti *trivially copyable*.

Default parametre

Nepodporujeme default parametre, teda vždy vyžadujeme aby sa počet argumentov zhodoval s počtom parametrov. Implementácia tohoto mechanizmu by mala byť triviálna, stačí upraviť príslušné miesta v overload resolution a na mieste s volaním funkcie vytvoriť v kompilačnom čase tabuľku pre všetky možné počty argumentov.

Variadické funkcie

Nepodporujeme variadické funkcie. Tento mechanizmus je v C++ málo používaný. Neskúmali sme ani, ako by ich podpora vyzerala.

Vylepšenie `viable_functions`

Optimalizácia pomocou `viable_functions` v aktuálnom stave funguje tak, že typy argumentov sa musia presne zhodovať s vopred uvedenými. To by sa dalo vylepšiť, aby bola povolená malá odchýlka. Tiež by bolo možné zaviesť usporiadanie medzi postupnosťami typov argumentov, aby bolo možné v zozname vyhľadávať v logaritmickom čase. Nie je zrejmé, či pri veľkostiach zoznamov, aké sú pri tomto procese bežné, by binárne vyhľadávanie prinieslo nejaký úžitok.

Výnimky

Na niektorých miestach ignorujeme možnosť výnimiek. Doplnenie zvládania výnimiek by nemalo vyžadovať nijaký zásah do návrhu, išlo by čisto o rozšírenie schopností.

Zoznam použitej literatúry

Bräutigam, K. (2015). Expedient logging for c++ using reflection. Diploma thesis, TU Wien.

de Bayser, M. a Cerqueira, R. (2012). A system for runtime type introspection in C++. In de Carvalho Junior, F. H. a Barbosa, L. S., editors, *Programming Languages*, pages 102–116, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33182-4.

Demers, F.-N. a Malenfant, J. (1995). Reflection in logic, functional and object-oriented programming: a short comparative study. In *Proceedings of the IJCAI*, volume 95, pages 29–38. Citeseer.

JetBrains (2020). The state of developer ecosystem 2020. `https://www.jetbrains.com/lp/devecosystem-2020`. Prístup: 9. 7. 2021.

Loikkanen, T. (2019). Property systems in graphics frameworks. Bachelor's thesis, Tampere University.

Muschevici, R., Potanin, A., Tempero, E. a Noble, J. (2008). Multiple dispatch in practice. *ACM SIGPLAN Notices*, **43**(10), 563–582.

Zoznam obrázkov

Zoznam tabuliek