Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Klemen Marković

Refleksija v programskem jeziku C++

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2020

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Programski jezik C++ ima v primerjavi z novejšimi objektno usmerjenimi jeziki zgolj najbolj osnovno podporo za refleksijo. Preglejte trenutne zmožnosti refleksije v programskem jeziku C++ in obstoječe knjižnice, ki dodajo refleksijo v C++. Implementirajte lastno knjižnico za refleksijo v programskem jeziku C++, ki bo izboljšala refleksijo v jeziku C++, in jo primerjajte z obstoječimi rešitvami.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	Application programming interface	Aplikacijski programski vmesnik
BFD	Binary file descriptor	Opisnik binarnih datotek
CRTP	Curiously recurring template pattern	Radovedno ponavljajoč šablonski vzorec
CV	Const and volatile qualifier	Kvalifikator const in volatile
dbghelp	Debug helper	Pomagač razhroščevanja
GCC	GNU Compiler Collection	Zbirka prevajalnikov GNU
HTTP	HyperText Transfer Protocol	Protokol za prenos hiperteksta
JSON	Java Script Object Notation	Objektna notacija Java Script
LLVM	Low level virtual machine	Nižjenivojski navidezni stroj
MSVC	Microsoft Visual C++ compiler	Prevajalnik Microsoft Visual C++
REST	Representational state transfer	Reprezentativen prenos stanja
RTTI	Run time type information	Informacije o tipih v času izvajanja
SDK	Software development kit	Orodje za razvoj programske opreme
SFINAE	Substitution failure is not an error	Problem pri zamenjavi ni napaka

Povzetek

Naslov: Refleksija v programskem jeziku C++

Avtor: Klemen Marković

Programski jezik C++ je v zadnjem desetletju s standardi C++11, C++14 in C++17 postal prenovljen in moderen programski jezik. Kljub vsem pridobitvam programski jezik C++ po definiciji še vedno ni refleksijski programski jezik, kar pa se bo v bližnji prihodnosti morda spremenilo. V diplomskem delu je predstavljen uradni predlog standardizirane refleksije v programskem jeziku C++. Prav tako je v diplomskem delu narejena primerjava med nekaterimi obstoječimi refleksijskimi knjižnicami in refleksijsko knjižnico, ki smo jo razvili za to diplomsko delo. Nova refleksijska knjižnica se od obstoječih knjižnic razlikuje v tem, da ima nekatere dodatne zmožnosti, ki so naravnane v poenostavljeno uporabo, možnost uporabe na večji množici obstoječe programske kode, večjo varnost pri uporabi in boljšo učinkovitost pri izvajanju. Z nekaj primeri je predstavljena smiselnost in praktičnost uporabe tovrstne refleksije v programskem jeziku C++.

Ključne besede: C++, refleksija.

Abstract

Title: Reflection in C++ programming language

Author: Klemen Marković

C++ programming language has become renewed and modern programming language in the last decade with standards C++11, C++14 and C++17. Even though the language has many new features, it is still not a reflective language by definition, which may change in the near future. In this thesis, the official proposal for standardized reflection in C++ is presented. In addition, comparison between some already existing reflection libraries and a reflection library, which was implemented for the purpose of this thesis, is also presented. The new reflection library has some additional and unique features, which address more simplified usage, capability to use the library on a larger set of existing code, better usage safety and better execution performance. With some examples, the practicality of such reflection in C++ programming language is also demonstrated.

Keywords: C++, reflection.

Poglavje 1

Uvod

Refleksija v programskih jezikih je širok pojem. Obstaja več definicij, kaj refleksija je. Poglejmo si nekaj teh definicij:

- *Refleksija je sposobnost programa, da med svojim izvajanjem spreminja podatke, ki predstavljajo stanje programa [11].*
- *Refleksija je poglavitna sposobnost programa, da opazuje in spreminja svojo programsko kodo kot tudi vse aspekte programskega jezika (sintakso, semantiko ali implementacijo), tudi med izvajanjem [11].*
- *Refleksija je sposobnost procesa, da preveri in spremeni svojo strukturo in obnašanje [23].*

Lahko bi rekli, da je zbirnik refleksijski jezik, saj je sestavljen iz ukazov, ki so v pomnilniku shranjeni na enak način kot podatki [23]. S spreminjanjem podatkov lahko spremenimo ukaze in s tem potek izvajanja programa. Tovrstna koda je poznana tudi kot samo spreminjajoča se koda (angl. self-modifying code), ki pa je zaradi varnostnih razlogov na modernih arhitekturah ni več mogoče izvajati.

S pojavom prvih programskih jezikov, ki so delovali na višjem nivoju kot zbirni jeziki, je zmožnost refleksije izginila [23]. Razlog za to je, da se ob prevajanju programske kode v zbirnik oziroma strojno kodo večina informacij o sami strukturi izvorne programske kode izgubi. Refleksija se

je zopet pojavila v objektno usmerjenih programskih jezikih, kot sta Java in C#, in v nekaterih drugih programskih jezikih, ki danes niso več tako popularni.

V objektno usmerjenih programskih jezikih nam refleksija omogoča možnost pregleda polj in metod nekega razreda v času izvajanja programske kode [7]. Prav tako nam omogoča ustvarjanje objektov, pridobivanje ali spreminjanje vrednosti polj in klicanje metod, pri čemer razred objekta, polje ali metodo izberemo po imenu šele v času izvajanja in ne v trenutku prevajanja programske kode [23]. Tovrstni refleksiji med drugim pravimo tudi refleksija v času izvajanja (angl. run-time reflection). Lahko nam pomaga pri pisanju generične programske kode za vse refleksibilne razrede, saj imajo vsi tovrstni razredi enak vmesnik, preko katerega lahko pridobivamo refleksijske informacije in spreminjamo stanje objektov s spreminjanjem razrednih polj in klicanjem razrednih metod. Z določenimi orodji, ki jih refleksija omogoča [2], lahko programer z manj programske kode bolje izrazi svoj namen in s tem prihrani pri pisanju ponavljajoče se programske kode, ki bi bila sama sebi namen (angl. boilerplate code).

Primer, kjer nam refleksija lahko pomaga, je razčlenjevalnik ukazne vrstice (angl. command line parser) [15]. Namesto da za vsako strukturo ukazne vrstice napišemo specifičen razčlenjevalnik, ki v času izvajanja napolni določeno strukturo glede na določene opcije v ukazni vrstici, lahko napišemo generični razčlenjevalnik ukazne vrstice, ki razčleni ukazno vrstico glede na podano strukturo. To stori na način, da v času izvajanja analizira polja vhodne strukture z uporabo refleksije. Ime posameznega polja lahko ponazarja ime posamezne opcije, tip polja pa lahko ponazarja način uporabe posamezne opcije. S tem smo zmanjšali soodvisnosti med vhodno strukturo in razčlenjevalnikom. Če spremenimo vhodno strukturo, nam razčlenjevalnika ni potrebno ponovno prevajati. Spreminjanje in dodajanje opcij v ukazni vrstici je poenostavljeno na spreminjanje in dodajanje polja v strukturi.

Drugi primer, kjer nam refleksija lahko pomaga na podoben način, je generični strežnik REST [15]. Brez refleksije bi za vsak klic, ki je poslan

strežniku REST, morali narediti transformacijo iz klica HTTP na določeno funkcijo v programski kodi. Pri tem bi morali razstaviti vhodne parametre klica HTTP, jih podati funkciji, sestaviti vrnjeno vrednost funkcije in jo vrniti klicatelju HTTP. To pomeni, da bi za vsak novi klic morali spreminjati implementacijo strežnika REST. Z uporabo refleksije lahko razvijemo generični strežnik REST, pri čemer funkcionalnost strežnika opišemo z dodatnim razredom na podoben način, kot smo opisali opcije ukazne vrstice z vhodno strukturo v prejšnjem primeru. Vsaka metoda razreda lahko ponazarja klic na strežniku REST. Argumenti metode lahko ponazarjajo podatke, ki jih klic REST sprejme, vrnjena vrednost pa lahko ponazarja vrednost, ki jo klic REST vrne. Če hočemo dodati ali spremeniti klic REST, nam ni potrebno spreminjati implementacije strežnika REST, ampak spremenimo razred, ki opisuje funkcionalnost. S tem smo zmanjšali soodvisnost med implementacijo strežnika REST in funkcionalnostjo, ki jo strežnik ponuja, obenem pa smo tudi razbremenili programerja, saj lahko z manj programske kode bolje izrazi svoj namen.

Tovrstnih primerov je še veliko več. Na tem mestu bi omenili nekaj orodij, ki jih refleksija omogoči v programskem jeziku Java: Hibernate [9], JAX-RS [6], Spring [17]. Vsem orodjem je skupno, da za doseg ciljev uporabljajo refleksijo.

V programskem jeziku C++ brez refleksije ni mogoče razviti podobnih orodij, kot jih imamo v programskih jezikih Java in C#. Posledično ima programski jezik C++ kar nekaj pomanjkljivosti. Pri prenosu podatkov iz enega procesa v drugega se velikokrat srečamo s problemom serializacije podatkov. Z refleksijo sta serializacija in deserializacija vrednosti vseh tipov generalizirani, saj obstajajo le osnovni in sestavljeni tipi, v programskem jeziku C++ pa moramo serializacijo in deserializacijo razviti sami za vsak tip posebej. Prav tako ni enostavne rešitve za implementacijo objektno relacijskega preslikovalnega mehanizma. Največkrat takšne rešitve [13] vsebujejo generator programske kode, ki razčleni izvorno programsko kodo in generira dodatno programsko kodo, ki implementira dejansko preslikavo. Zaradi želje, da bi

programski jezik C++ postal refleksijski programski jezik, se je pojavil uradni predlog [21], ki bi programskemu jeziku C++ dodal zmožnost refleksije v času prevajanja (angl. compile time reflection) [4]. Predlog še ni potrjen za prihajajoči standard C++20, je pa v stanju, ki omogoča, da bo refleksija postala del standarda C++23 [8].

Namen diplomskega dela je prikazati, da je tudi v programskem jeziku C++ mogoče uporabiti refleksijo za podobne cilje kot v programskih jezikih Java in C#. Čeprav programski jezik C++ ni refleksijski programski jezik, obstaja kar nekaj knjižnic, ki omogočijo, da se programski jezik C++ uporabi kot refleksijski programski jezik. Dosedanje razširitve standarda so dodale nekaj novih zmožnosti, s katerimi je omogočena refleksija določenih aspektov programskega jezika C++, s čimer je omogočena implementacija tovrstnih refleksijskih knjižnic.

Cilj diplomskega dela je, da bi uporabili refleksijsko knjižnico v manj sklopljenem sistemu, ki je sestavljen iz knjižnic, ki se dinamično nalagajo v procesni prostor po potrebi. Vsaka knjižnica oziroma komponenta sistema vsebuje implementacijo določenih razredov, ki se jih lahko uporabi preko določenih vmesnikov, ne da bi uporabnik razredov pri prevajanju potreboval njihovo izvorno programsko kodo. Želimo si, da bi vsi razredi imeli poleg določenih vmesnikov še vmesnik, ki ponuja refleksijo, s čimer se uporaba razredov lahko razširi preko meje programskega jezika C++. Primer tega je izvajanje programske kode C++ z uporabo skriptnega jezika, kar lahko zmanjša potrebo po prevajanju, saj lahko del logike, ki se bolj pogosto spreminja, prestavimo ven iz programske kode C++, kjer obdržimo samo funkcionalnost, za katero je temeljno, da se izvaja čimbolj učinkovito. Stremimo k temu, da določene komponente sistema, ki so razvite v programskem jeziku C++, lahko predstavimo v obliki strežnikov REST, da bi za njihovo uporabo potrebovali le odjemalec REST. Pri tem bi nam lahko pomagale nekatere obstoječe refleksijske knjižnice, vendar imajo nekatere pomanjkljivosti, zaradi katerih jih ne moremo uporabiti. Odločili smo se, da razvijemo novo refleksijsko knjižnico, ki:

- je bolj enostavna za uporabo;
- je bolj varna za uporabo;
- omogoča pisanje generične programske kode za vse refleksibilne razrede z uporabo polimorfizma;
- je dovolj učinkovita v primerjavi z ostalimi knjižnicami.

V 2. poglavju so predstavljene trenutne refleksijske zmožnosti programskega jezika C++. V 3. poglavju so predstavljene poglavitne razlike med refleksijo v času prevajanja in refleksijo v času izvajanja. V 4. poglavju sta narejeni podrobna analiza uradnega predloga za standardizirano refleksijo in analiza obstoječih refleksijskih knjižnic. V 5. poglavju je predstavljena implementacija nove refleksijske knjižnice kot tudi primerjava nove knjižnice z nekaterimi obstoječimi knjižnicami.

Poglavje 2

Obstoječe refleksijske zmožnosti v programskem jeziku C++

2.1 Sledi tipov

Sledi tipov (angl. type traits) omogočijo, da v šablonski programski kodi (angl. template code), kjer upravljamo z množico tipov, ki so parametri šablon, lahko pridobivamo določene informacije o tipih.

V programski kodi 2.1 vidimo implementacijo šablonske funkcije *check*, ki uporabi šablonsko strukturo *std::is_floating_point*, da ugotovi, če je podani tip T tip s plavajočo vejico. Podobnih šablonskih struktur, ki vračajo informacije o tipih, je veliko. Lahko nam pomagajo pri ugotavljanju:

- Ali je podan tip enak določenemu tipu?
- Ali je podan tip razred?
- Ali je podan tip v obliki leve ali desne reference (angl. lvalue or rvalue reference)?
- Ali podan razred deduje določen razred?

- Ali ima podan razred privzet konstruktor?
- Ali ima podan razred konstruktor za kopiranje (angl. copy constructor)?

```
#include <iostream>
#include <type_traits>
using namespace std;
class MyClass {};

template <typename T>
void check()
{
    cout <<
        is_floating_point<T>::value <<
        endl;
}

void example()
{
    check<MyClass>();
    check<float>();
    check<int>();
}
```

Programska koda 2.1: Sledi tipov.

Večina sledi tipov je postala na voljo s standardom C++11. Vsak novi standard prinese nekaj novih sledi tipov.

2.2 Dedukcija tipov

Dedukcija tipov (angl. type deduction) omogoča, da iz določene vrednosti pridobimo tip vrednosti. Na voljo je novi izraz *decltype*, ki za argument

pričakuje vrednost kateregakoli tipa, vrne pa nam tip podane vrednosti. Vrnjen tip lahko uporabimo v šablonski programski kodi oziroma kjerkoli v programski kodi, kjer navajamo tipe. Posledica tega je, da imamo na voljo tudi izraz *auto*, ki ga lahko uporabimo namesto določenega tipa pri deklaracijah spremenljivk za namen poenostavitve programske kode. Tako izraz *decltype* kot izraz *auto* sta postala na voljo s standardom C++11.

V programski kodi 2.2 pri deklaraciji spremenljivke *b* in *c* uporabimo dedukcijo tipov.

```
1  auto example() {
2      int a = 5;
3      decltype(a) b = 10 + a;
4      auto c = 15 + b;
5      return c;
6  }
```

Programska koda 2.2: Dedukcija tipov.

2.3 Delna in eksplicitna specializacija šablon

Delna in eksplicitna specializacija šablon (angl. partial and explicit template specialization) sta na voljo že od prvega standarda C++, ki je vseboval šablone.

V programski kodi 2.3 vidimo način, kako ugotoviti, ali je podan tip tip vrednosti, tip kazalca, tip desne reference, tip vektorja ali točno določen tip *long*.

```
1  #include <vector>
2  using namespace std;
3
4  template <typename T>
5  struct MyTypeTrait {
```

```
6       static void what() {
7           cout << "T is a value" << endl;
8       }
9   };
10  template <typename T>
11  struct MyTypeTrait<T*> {
12      static void what() {
13          cout << "T is a pointer" << endl;
14      }
15  };
16  template <typename T>
17  struct MyTypeTrait<T&&> {
18      static void what() {
19          cout << "T is a rvalue reference" << endl;
20      }
21  };
22  template <typename T>
23  struct MyTypeTrait<vector<T>> {
24      static void what() {
25          cout << "T is a vector of values" << endl;
26      }
27  };
28
29  template <>
30  struct MyTypeTrait<long> {
31      static void what() {
32          cout << "T is long" << endl;
33      }
34  };
35
36  void example() {
37      MyTypeTrait<int>::what();
38      MyTypeTrait<double &&>::what();
39      MyTypeTrait<float *>::what();
40      MyTypeTrait<vector<bool>>::what();
41      MyTypeTrait<long>::what();
```

```
42 }
```

Programska koda 2.3: Delna in eksplicitna specializacija šablon.

Sicer bi lahko te informacije dobili tudi z uporabo sledi tipov. Ko sledi tipov niso na voljo, lahko s specializacijami šablon dodajamo nove sledi.

2.4 Refleksija razrednih metod

Če pri vseh do sedaj naštetih zmožnostih uporabimo še šablone s spreminjajočim številom argumentov (angl. variadic templates), lahko tipe razrednih metod razstavimo na tip vrnjene vrednosti in tipe vhodnih argumentov. Pri tem lahko ugotovimo tudi, kakšne kvalifikatorje uporablja določena razredna metoda. Programski jezik C++ ponuja kvalifikatorje CV (angl. CV qualifiers) in referenčne kvalifikatorje (angl. reference qualifiers), s katerimi kvalificiramo, v kakšnih situacijah se razredne metode izvajajo. Tako lahko določimo, da se določena metoda izvede samo v primeru, ko je objekt razreda konstanten (angl. const) ali nestalen (angl. volatile) oziroma uporabljen kot leva ali desna referenca.

V programski kodi 2.4 uporabimo šablonsko strukturo *MyMethodTraits*, ki za parameter sprejme tip razreda in tip razredne metode [14]. Splošen primer šablonske strukture je prazna struktura, poleg pa imamo še dve delni specializaciji, pri kateri je ena namenjena metodam brez kvalifikatorjev, druga pa je namenjena metodam s kvalifikatorjem *const*. V šablonskih strukturah definiramo tipe, ki refleksirajo tip vrnjene vrednosti in tipe vhodnih argumentov. Prav tako definiramo konstanto *isConst*, ki jo nastavimo glede na uporabljeno specializacijo. Ker so vhodni argumenti v obliki paketa parametrov (angl. parameter pack), jih preoblikujemo v heterogeni vsebovalnik *tuple*, saj moramo paket parametrov v tem kontekstu razširiti. Poleg imamo še šablonsko funkcijo *printTypesHelper*, ki z uporabo sistema RTTI izpiše imena tipov, ki so del paketa parametrov. Pri uporabi šablonske strukture

MyMethodTraits uporabimo dedukcijo tipa, da iz metodnega kazalca, ki kaže na metodo *Foo::Bar*, dobimo tip metodnega kazalca, ki ga šablonska struktura pričakuje. Programska koda bo izpisala, ali metoda *Foo::Bar* uporablja kvalifikator *const*, tip vrnjene vrednosti in tipe vhodnih parametrov.

```
1  template <typename Class, typename Return, typename... Args>
2  struct MyMethodTraits;
3
4  template <typename Class, typename Return, typename... Args>
5  struct MyMethodTraits<Class, Return(Class::*)(Args...)> {
6      using ReflectedClass = Class;
7      using ReflectedReturnType = Return;
8      using ReflectedArgs = std::tuple<Args...>;
9      static const bool isConst = false;
10 };
11
12 template <typename Class, typename Return, typename... Args>
13 struct MyMethodTraits<Class, Return(Class::*)(Args...) const> {
14     using ReflectedClass = Class;
15     using ReflectedReturnType = Return;
16     using ReflectedArgs = std::tuple<Args...>;
17     static const bool isConst = true;
18 };
19
20 template <typename... Args>
21 void printTypesHelper(std::tuple<Args...>) {
22     int dummy[] = {
23         (cout << typeid(Args).name() << endl, 0)...
24     };
25 }
26
27 class Foo {
28 public:
29     int bar(int arg1, float arg2) const;
30 };
31
```

```
32  void example() {
33      Foo foo;
34
35      cout <<
36          "Method Foo::bar() " <<
37          (MyMethodTraits<Foo, decltype(&Foo::bar)>::isConst ?
38              "is a const method" :
39              "is not a const method") <<
40          endl;
41
42      cout <<
43          typeid(MyMethodTraits<Foo, decltype(
44              &Foo::bar
45          )>::ReflectedReturnType).name() <<
46          endl;
47
48      printTypesHelper(
49          MyMethodTraits<
50              Foo,
51              decltype(&Foo::bar)
52          >::ReflectedArgs{}
53      );
54  }
```

Programska koda 2.4: Refleksija razrednih metod.

V tem primeru se bo izpisalo naslednje:

```
Method Foo::bar() is a const method
int
int
float
```

2.5 Koncept SFINAE

Programski jezik C++ vsebuje koncept SFINAE. Namen koncepta SFINAE je, da se pri prevajanju šablonske programske kode zavrže tiste instance šablon, ki v prototipu vsebujejo programsko kodo, ki se ne prevede. To omogoči uporabo iste šablonske programske kode za širšo množico tipov. V programski kodi 2.5 vidimo enostaven primer uporabe koncepta SFINAE.

```
1  template <typename T>
2  void overload(T *t) {
3  }
4  template <typename T>
5  void overload(T &t) {
6  }
7  //void overload(int&* t) {} // error
8
9  void example() {
10     int i;
11     overload<int&>(i);
12 }
```

Programska koda 2.5: Koncept SFINAE.

Imamo dve šablonski prekrivni metodi z imenom *overload*, kjer ena sprejme kazalec, druga pa referenco tipa *T*. Ker uporabimo instanco šablonske metode *overload<int&>*, bi pri prvi prekrivni metodi (*void overload(int&* t)*) brez koncepta SFINAE prevajalnik javil napako, saj ni dovoljeno uporabiti kazalec na referenco. Ker prevajalnik izloči prvo prekrivno metodo, se v tem primeru pokliče druga prekrivna metoda (*void overload(int &&t)*), ki za argument sprejme desno referenco.

Ta koncept nam prav tako omogoči, da v času prevajanja preverimo, ali obstaja določena razredna metoda. V programski kodi 2.6 vidimo, da imamo dve strukturi, kjer ima vsaka svojo metodo [14]. Šablonska funkcija *fooOrBar* uporabi novo zmožnost, ki jo ponuja standard C++14, in sicer da

lahko specificiramo vrnjeno vrednost na koncu funkcijskega prototipa. To nam omogoči, da lahko definiramo tip vrnjene vrednosti glede na tip *T*, ki je parameter šablonske funkcije. Če je šablonska funkcija parametrizirana s tipom *hasBar*, se bo prva instanca šablonske funkcije *fooOrBar* zavrgla, saj v samem prototipu poskuša ugotoviti tip vrnjene vrednosti metode *foo*, ki ne obstaja. Iz tega razloga se bo izbrala druga instanca, ki se ne bo zavrgla, saj poskuša ugotoviti tip vrnjene vrednosti metode *bar*, ki obstaja in je v tem primeru *void*. S tem smo dosegli, da se s šablonsko funkcijo *fooOrBar* za strukturo *hasFoo* pokliče metoda *foo*, za strukturo *hasBar* pa se pokliče metoda *bar*. Če uporabimo strukturo *hasBoth*, se programska koda zaradi dvoumnosti (angl. ambiguity) ne bo prevedla.

```
1  template<typename T>
2  auto fooOrBar(T const& t) -> decltype(t.foo()) {
3      return t.foo();
4  }
5  template<typename T>
6  auto fooOrBar(T const& t) -> decltype(t.bar()) {
7      return t.bar();
8  }
9  void example() {
10     struct HasFoo { void foo() const {} } hf;
11     struct HasBar { void bar() const {} } hb;
12     struct HasBoth : HasFoo, HasBar {} hfb;
13
14     fooOrBar(hf);
15     fooOrBar(hb);
16     //fooOrBar(hfb); // error
17 }
```

Programska koda 2.6: Koncept SFINAE.

Podobno lahko dosežemo tudi malo drugače z uporabo standarda C++17. V programski kodi [14] 2.7 uporabimo šablonsko spremenljivko *hasFoo*, ki je

za splošen primer nastavljena na *false*. Z delno specializacijo in uporabo koncepta SFINAE jo nastavimo na *true*, če tip T vsebuje metodo *foo*. Izraz *std::declval* je obraten izraz izraza *decltype*, ki nam glede na podan tip vrne objekt tega tipa brez njegovega ustvarjanja, saj se izraz *std::declval* uporablja samo v neevaluiranem kontekstu (angl. unevaluated context). Pogoj v času prevajanja *if constexpr* nam omogoči, da se programska koda s klicem na metodo *foo* definira samo takrat, ko metoda obstaja, sicer se programska koda definira brez klica na metodo *foo*. Brez takšnega pogoja v času prevajanja se programska koda ne prevede, saj ne more vsebovati klica na metodo, ki ne obstaja.

```
template<typename, typename = void>
constexpr bool hasFoo = false;

template<typename T>
constexpr bool hasFoo<
    T,
    void_t<
        decltype(declval<T>().foo())
    >
> = true;

template <typename T> void callFooIfItExists(T t) {
    if constexpr (hasFoo<T>) {
        cout <<
            "T contains foo(), calling foo()" <<
            endl;
        t.foo();
    }
    else {
        cout <<
        "T does not contain foo()" <<
        endl;
    }
}
```

```
25
26  void example() {
27      struct HasFoo { void foo() const {} } hf;
28      struct HasBar { void bar() const {} } hb;
29      struct HasBoth : HasFoo, HasBar {} hfb;
30
31      callFooIfItExists(hf);
32      callFooIfItExists(hb);
33      callFooIfItExists(hfb);
34  }
```

Programska koda 2.7: Uporaba pogoja v času prevajanja.

Razlika med prvim in drugim primerom je v tem, da je v slednjem pogoj v telesu metode, zaradi česar ne moremo uporabiti koncepta SFINAE, saj ga lahko uporabimo samo v prototipu metode. Pogojni stavek v času prevajanja *if constexpr* je postal na voljo s standardom C++17.

Kot prikazano, programski jezik C++ vsebuje bogato množico manjših zmožnosti, s katerimi je mogoče razviti večjo zmožnost v obliki refleksijske knjižnice. Uradni predlog, da programski jezik C++ postane refleksijski programski jezik, prav tako prinaša množico manjših zmožnosti, ki poenostavijo razvoj refleksijske knjižnice in omogočijo še tisto, kar danes ni mogoče.

Poglavje 3

Refleksija v času prevajanja ali v času izvajanja?

Preden pojasnimo, kaj pomeni refleksija v času prevajanja, moramo najprej razumeti, kaj pomeni, če se programska koda izvede v času prevajanja.

Programski jezik C++ je s standardom C++11 pridobil določilnik *constexpr* (angl. *constexpr* specifier), ki določa, da se programska koda izvede v času prevajanja, če so izpolnjeni vsi pogoji. Lahko se ga uporabi pri deklaraciji spremenljivk in funkcij ter pri deklaraciji statičnih razrednih polj, razrednih metod in konstruktorjev. Pogoji za izvršitev v času prevajanja so, da se pri deklaraciji spremenljivk oziroma klicu funkcij uporabijo samo vrednosti, ki so znane v času prevajanja (literali), ali spremenljivke in funkcije, ki so deklarirane z določilnikom *constexpr*. Če pogoji niso izpolnjeni, se bo programska koda kljub temu prevedla, izvršitev pa se bo premaknila v čas izvajanja.

Ker je šablonska funkcija *square* v programski kodi 3.1 deklarirana z določilnikom *constexpr* in klicana z literalom, jo prevajalnik izvede v času prevajanja. To pomeni, da bo prevajalnik že v času prevajanja kvadriral vhodni argument in generiral prevedeno kodo s končnim rezultatom. Če metoda ne bi bila označena z določilnikom *constexpr*, bi se kvadriranje izvedlo v času izvajanja programa, kot bi sicer pričakovali.

```
template<typename Type>
constexpr Type square(Type type) {
    return(type * type);
}
void example() {
    cout << square(5) << endl;
    cout << square(3.14f) << endl;
}
```

Programska koda 3.1: Šablonska funkcija, ki se izvede v času prevajanja.

Ko govorimo o refleksiji v času prevajanja ali pa o refleksiji v času izvajanja, govorimo o tem, ali se refleksijska koda izvede v času prevajanja ali v času izvajanja. Prednost refleksije v času prevajanja je predvsem v tem, da se v času izvajanja nič dodatnega ne izvaja, s čimer se izboljša učinkovitost.

Programski jezik C++ je po svoji filozofiji naravnan tako, da poskušajo prevajalniki C++ programsko kodo prevesti v takšno strojno kodo, ki bi bila čim bolj učinkovita pri izvajanju. Iz tega razloga razredne metode niso privzeto virtualne in jih je zato potrebno dodatno označiti, saj je izvajanje virtualnih metod nekoliko dražje od izvajanja metod, ki niso virtualne. Če se vsa refleksijska programska koda prestavi v čas prevajanja, je to torej iz vidika učinkovitosti v duhu filozofije programskega jezika C++.

Če se vsa refleksijska programska koda izvede v času prevajanja, pa s tem nekaj izgubimo. Nekaterih aspektov programske kode ni mogoče refleksirati v času prevajanja. Refleksija v času prevajanja lahko refleksira samo tiste aspekte programske kode, ki so znani v času prevajanja. Z refleksijo v času prevajanja zato ne moremo poklicati metode po imenu, pri čemer bi bilo ime metode znano šele v času izvajanja.

Refleksija v času prevajanja ne ustreza definiciji refleksije: *Refleksija je sposobnost programa, da med izvajanjem spreminja podatke, ki predstavljajo stanje programa [11].* Refleksijo v času prevajanja lahko opiše naslednja definicija: *Statična refleksija je poglavitna sposobnost programa, da v času*

prevajanja opazuje svojo programsko kodo, s čimer oblikuje svojo definicijo [18]. Refleksija v času prevajanja je torej konceptualno precej drugačna od refleksije v času izvajanja, čeprav v obeh primerih govorimo o refleksiji.

Velja pravilo, da z refleksijo v času prevajanja lahko razvijemo knjižnico za refleksijo v času izvajanja [15], obratno pa tega ne moremo storiti, saj se programska koda najprej prevaja in šele nato izvaja.

Ko se odločamo o tem, ali bi uporabili refleksijo v času prevajanja ali refleksijo v času izvajanja, moramo upoštevati tudi dejstvo, ali lahko vso programsko kodo prevajamo skupaj. Če se vrnemo nazaj na primer generičnega razčlenjevalnika ukazne vrstice, bi to pomenilo, da moramo pri uporabi refleksije v času prevajanja generični razčlenjevalnik prevajati skupaj s strukturo ukazne vrstice. Prednost takega pristopa je boljša hitrost izvajanja, vprašanje pa je, ali razlika v hitrosti lahko opraviči večjo sklopljenost programske kode v času prevajanja in daljši čas prevajanja.

Refleksija v času izvajanja je lažja za uporabo, saj je pri programiranju povsem naravno, da razmišljamo o tem, kako se bo programska koda izvajala. Ko uporabljamo razhroščevalnik, ga uporabljamo med izvajanjem programske kode, torej se z njim premikamo po programski kodi, ko se dejansko izvaja. Pri uporabi refleksije v času prevajanja pa moramo razmišljati tako, da razumemo, kakšno končno programsko kodo bo prevajalnik proizvedel.

Uradni predlog [21] za standardizirano refleksijo v času prevajanja v programskem jeziku C++ ne prinaša refleksije, kot jo poznamo v programskih jezikih Java in C#. Tako programski jezik Java kot C# uporabljata refleksijo v času izvajanja in koncept osnovnega razreda (angl. base class), pri čemer osnovni razred z definiranim vmesnikom omogoča refleksijo razreda.

Razširitev standarda v programskem jeziku C++ prinaša nov izraz, ki bo omogočil refleksijo na nivoju izrazov v času prevajanja. S tem nam programski jezik C++ prinaša osnovne gradnike, s katerimi bomo lahko lažje razvili knjižnico za refleksijo v času izvajanja, ki bi bila bolj podobna refleksiji v programskih jezikih Java in C#. Z osnovnimi gradniki bomo lahko razvili tudi specifično domensko knjižnico, kot sta npr. knjižnica za serializacijo ali

knjižnica za objektno relacijski preslikovalni mehanizem.

Knjižnica, ki smo jo razvili za to diplomsko delo, je knjižnica za refleksijo v času izvajanja. Od ostalih že obstoječih knjižnic se razlikuje v tem, da definira osnovni razred in se v tem pogledu zgleduje po refleksiji, kot jo poznamo v programskih jezikih Java in C#. Razlog za to odločitev je predvsem v tem, da bi imeli vsi razredi, ki jih je mogoče refleksirati, skupen vmesnik in s tem nekaj skupnega, kar nam z uporabo polimorfizma omogoča pisanje generične programske kode za vse refleksibilne razrede.

Poglavje 4

Pregled področja

V zadnjih letih je bilo na področju refleksije v programskem jeziku C++ izdanih veliko člankov, ki poskušajo standardizirati način uporabe refleksije v času prevajanja. Pri tej tematiki je kar nekaj različnih mnenj, predvsem v načinu pisanja programske kode, ki se mora izvesti v času prevajanja.

Začelo se je s predlogom [3] [4], da bi jezik dobil nov izraz *reflexpr*, preko katerega bi dobili refleksijske informacije o izrazu, ki ga podajamo. Po prvotnem predlogu bi izraz *reflexpr* vrnil tip, preko katerega bi pridobivali refleksijske informacije. Vrnjen tip lahko vsebuje statična polja, ki vsebujejo končne informacije, ali pa ga uporabimo kot parameter pri šablonskih strukturah, s čimer se pomikamo v globino pri pridobivanju refleksijskih informacij.

V programski kodi [22] 4.1 vidimo uporabo izraza *reflexpr*, ki vrne tip, ki ga lahko uporabimo pri specializaciji šablonske strukture *get_data_members*, s čimer dobimo nov tip, ki opisuje vse elemente neke strukture ali razreda. Problem tega pristopa je, da prevajalniki C++ vsak nov tip hranijo do konca prevajanja posamezne prevajalne enote [18]. Takšen pristop bi za prevajalnik lahko bila prevelika obremenitev, saj je namen tipov le začasne narave in izključno namenjen pomikanju v globino pri pridobivanje refleksijskih informacij. Drugi problem je, da je takšno programsko kodo težje razumeti, saj ne operira z običajnimi vrednostmi kot velika večina programske kode C++,

ampak s tipi. To pomeni, da je na veliko mestih namesto navadnih oziroma okroglih oklepajev potrebno uporabiti lomljene oklepaje, da ni mogoče uporabiti standardnih vsebovalnikov, da je oteženo razhroščevanje ipd.

```
1 struct Foo { int x; float y; };
2 using MetaInfo = reflexpr(Foo);
3 using Members = get_data_members_t<MetaInfo>;
```

Programska koda 4.1: Prvi predlog uporabe.

Potem se je pojavil drugi predlog [19], pri katerem bi izraz *reflexpr* namesto tipa vrnil vrednost, torej bi bil takšen način bližje pisanju običajne programske kode C++.

V programski kodi [22] 4.2 bi bila vrnjena vrednost objekt določenega tipa, torej bi s takšnim načinom namesto tipov uporabljali vrednosti, kar uporabo naredi bližjo običajnemu programiranju. Ker se tovrstna programska koda ne sme izvesti v času izvajanja, ampak v času prevajanja programske kode, bi izraz morali predznačiti z določilnikom *constexpr*, ki določuje, da se izraz izvede v času prevajanja, kar je tudi razlog za uporabo tipov in ne vrednosti v prvotnem predlogu.

```
1 struct Foo { int x; float y; };
2 constexpr auto meta_info = reflexpr(Foo);
3 constexpr auto members = get_data_members(meta_info);
```

Programska koda 4.2: Drugi predlog uporabe.

Problem tega pristopa je, da izraz *reflexpr* vrača vrednosti objektov razredov, ki so v določeni hierarhiji, kar lahko privede do problema rezanja objektov (angl. object slicing) [20]. Uporabljanje razredne hierarhije je smiselno pri uporabi referenc in kazalcev ter uporabi polimorfizma, pri uporabi vrednosti pa lahko privede do obnašanja, ki ni intuitivno. Programska koda 4.3 prikazuje rezanje objektov na način, ki ni intuitiven. Objekt *b2* bo vse-

boval polji *a* in *b* z vrednostma *1* in *2*. To se zgodi zato, ker privzet operator za prireditev vrednosti ni virtualen. Prav tako ga ne moremo dodati, saj operator, ki bi ga označili z določilnikom *constexpr*, ne more biti virtualen.

```
1  struct A {
2      int a;
3  };
4  struct B : public A {
5      int b;
6  };
7  void example10() {
8      B b1;
9      b1.a = 1;
10     b1.b = 1;
11     B b2;
12     b2.a = 2;
13     b2.b = 2;
14     A& a = b2;
15     a = b1;
16 }
```

Programska koda 4.3: Primer rezanja objektov.

Eden izmed razlogov, da ta pristop ni najboljši, je tudi ta, da je implementacija določilnika *constexpr* v prevajalnikih trenutno veliko bolj učinkovita v kombinaciji s skalarnimi vrednostmi kot vrednostmi objektov [20]. Drugi predlog prav tako ne rešuje problema, povezanega s preobremenitvijo prevajalnika zaradi uporabe prevelikega števila tipov.

Nato je nastal tretji predlog [20], da bi vsi meta objekti (vrednosti oziroma tipi, ki vračajo refleksijske informacije) postali vrednosti enega tipa *meta::info*. V programski kodi [22] 4.4 bi bil tip *meta::info* oprimek do abstraktnega sintaksnega drevesa prevajalnika v času prevajanja in na zunaj ne bi predstavljal nič drugega. Kritiki tega pristopa pravijo, da ne sledi smernicam objektno usmerjenega programiranja [16] niti ne smernicam močnega

tipiziranja, kar sta dve izmed glavnih lastnosti programskega jezika C++.

```
struct Foo { int x; float y; };
constexpr meta::info meta_info = reflexpr(Foo);
constexpr std::constexpr_vector<meta::info> members =
    get_data_members(meta_info);
```

Programska koda 4.4: Tretji predlog uporabe.

Ker ima vsak predlog nekaj prednosti in nekaj slabosti, se je pojavil še en pomemben članek [16], ki poskuša združiti prednosti vseh pristopov. V času pisanja je to tudi zadnji članek na tem področju. Predlagana rešitev bi uporabila parametrsko omejevanje (angl. parameter constraints), kar je nova prihajajoča zmožnost programskega jezika C++. Rešitev bi omogočila, da se za določeno vrednost oprimka *meta::info* lahko pokliče samo določeno metodo.

V programski kodi [16] 4.5 je prikazana uporaba parametrskega omejevanja z uporabo oprimka *meta::info*. Določilnik *consteval* deluje podobno kot *constexpr*, vendar se programska koda ne prevede, če izraza ni mogoče izvršiti v času prevajanja. Z izrazom *requires* omejimo vrednosti argumenta *i*.

```
struct type {
    consteval type(meta::info i) requires(meta::is_type(i));
};
struct class_ {
    consteval class_(meta::info i) requires(meta::is_class(i));
};
```

Programska koda 4.5: Predlog uporabe.

Poleg predlogov k razširitvi standarda C++ obstajajo tudi nekatere refleksijske knjižnice, ki se že uporabljajo. V nadaljevanju se bomo omejili na

dve tovrstni knjižnici, in sicer:

- RTTR (Run Time Type Reflection) [12]
- refl-cpp (Compile time Reflection for C++17) [1]

V času pisanja sta ti dve knjižnici na Googlovem iskalniku na prvih dveh mestih pri iskanju besedne zveze „C++ reflection“.

4.1 Uradni predlog za standardizirano refleksijo v času prevajanja

Pri opisovanju uradnega predloga se bomo omejili na prvotni predlog, ki uporablja tipe, saj je le-ta do potankosti definiran v trenutni tehnični specifikaciji [21]. Kot je že bilo omenjeno, predlog prinaša nov izraz *reflexpr*. Razlog za nov izraz je v tem, da bi bila refleksija nekaj novega, kar ne bi imelo vpliva na obstoječo programsko kodo [18]. To pomeni, da refleksija ne bi bila vključena za vsak razred, kot je to v programskih jezikih Java in C#, ampak bi na takšen način refleksirali samo tisto, kar bi hoteli.

Izraz *reflexpr* vrne neimenovan tip (angl. unnamed type), ki ustreza določenim konceptom. Namesto imenovanega tipa je vrnjen neimenovan tip, ki ustreza določenim konceptom. Koncepti v programskem jeziku C++ so nova zmožnost jezika in del specifikacije standarda C++20. Namen konceptov je, da omejijo uporabo vseh tipov v šablonah oziroma določijo, kateri tipi se lahko uporabljajo z določenimi šablonami. Iz stališča razširljivosti je dodajanje novih konceptov lažje kot spreminjanje obstoječih tipov, kar je razlog, da specifikacija ne omenja tipov, ki bi jih bilo potrebno v prihodnje razširjati, ampak omenja koncepte, ki jih bo v prihodnje mogoče dodajati. Ko jezik pridobi novo zmožnost (primer: lambda funkcije), je morda potrebno razširiti tudi refleksijske zmožnosti. To bi pomenilo, da se specifikaciji standardne knjižnice doda nov koncept in šablone, ki z novim konceptom upravljajo. Neimenovani tipi so sicer organizirani hierarhično in lahko ustrezajo enemu ali več konceptom hkrati. To je trenuten seznam konceptov, ki jih

specifikacija [21] omenja: *Object, ObjectSequence, TemplateParameterScope, Named, Alias, RecordMember, Enumerator, Variable, ScopeMember, Typed, Namespace, GlobalScope, Class, Enum, Record, Scope, Type, Constant, Base, FunctionParameter, Callable, Expression, ParanthesizedExpression, FunctionCallExpression, FunctionalTypeConversion, Function, MemberFunction, SpecialMemberFunction, Constructor, Destructor, Operator, ConversionOperator, Lambda, LambdaCapture.*

Vsi neimenovani tipi, ki jih vrača izraz *reflexpr*, ustrezajo konceptu *Object*. Programska koda 4.6 je del tehnične specifikacije in prikazuje prototipe šablonskih struktur za upravljanje s tipi, ki ustrezajo konceptu *Object*.

```
1 concept Object = /*...*/;
2 template <Object T1, Object T2> struct reflects_same;
3 template <Object T> struct get_source_line;
4 template <Object T> struct get_source_column;
5 template <Object T> struct get_source_file_name;
```

Programska koda 4.6: Koncept *Object*.

To pomeni, da bomo lahko pridobili informacije o tem, kje v programski kodi se nahajajo določene konstanta, spremenljivka, tip, funkcija in vse ostalo, kar bo mogoče refleksirati.

Programska koda 4.7 prikazuje prototipe šablonskih struktur za upravljanje s tipi, ki ustrezajo konceptu *Record*.

```
1 concept Record = /*...*/
2 template <Record T> struct get_data_members;
3 template <Record T> struct get_public_data_members;
4 template <Record T> struct get_accessible_data_members;
5 template <Record T> struct get_public_member_functions;
6 template <Record T> struct get_accessible_member_functions;
7 template <Record T> struct get_member_functions
```

```
8  template <Record T> struct get_member_types;
9  template <Record T> struct get_public_member_types;
10 template <Record T> struct get_accessible_member_types;
11 template <Record T> struct get_constructors;
12 template <Record T> struct get_destructor;
13 template <Record T> struct get_operators;
```

Programska koda 4.7: Koncept *Record*.

S konceptom *Record* bomo lahko pridobili seznam polj in metod neke strukture ali razreda. Mogoče bo pridobiti tudi informacije o zasebnih in zaščitenih elementih, torej bo mogoče zaobiti enkapsulacijo. Ravno zaradi tega obstajajo ločene šablonske strukture za pridobitev vseh ali pridobitev samo javnih elementov, kar bo orodjem za statično analizo programske kode omogočilo odkrivanje neprimernih zaobitev enkapsulacije [15].

Programska koda 4.8 prikazuje prototipe šablonskih struktur za upravljanje s tipi, ki ustrezajo konceptu *Named*. S tem konceptom bomo lahko pridobili ime v obliki niza za vse elemente, ki jih bo mogoče refleksirati.

```
1 concept Named = /*...*/
2 template <Named T> struct get_name;
3 template <Named T> struct get_display_name;
```

Programska koda 4.8: Koncept *Named*.

V času prevajanja bo mogoče pridobiti tudi kazalec na:

- globalno funkcijo ali spremenljivko
- polje ali metodo znotraj nekega razreda

Programska koda 4.9 prikazuje prototipe šablonskih struktur za upravljanje s tipi, ki ustrezajo konceptoma *Variable* in *Function*, in omogočajo pridobitev kazalcev.

```
concept Variable = /*...*/
template <Variable T> struct get_pointer;
...
concept Function = /*...*/
template <Function T> struct get_pointer;
```

Programska koda 4.9: Koncepta *Variable* in *Function*.

Sicer lahko kazalec na spremenljivko, razredno polje, funkcijo ali razredno metodo dobimo že danes brez refleksije, z refleksijo pa lahko dobimo tudi nekatere druge informacije, ki jih danes ne moremo. V primeru, ko je predmet šablonske strukture *get_pointer* globalna spremenljivka, struktura ne bo vsebovala polja *value*, če globalna spremenljivka ni bila statično inicializirana. Samo takrat, ko je globalna spremenljivka statično inicializirana, šablonska struktura *get_pointer* vsebuje polje *value*, ki je kazalec na globalno spremenljivko enakega tipa kot dejanska spremenljivka [21]. Pri pridobivanju kazalca z uporabo konceptov *Variable* in *Function* gre tudi za to, da je funkcionalnost refleksije celostna na enem mestu. Tako lahko bolj enostavno naštejemo vsa polja in metode nekega razreda, pri čemer za vsako polje in metodo pridobimo tudi kazalec.

Podobno dvojnost lahko vidimo pri konceptih *Type* in *Typed*. V programski kodi 4.10 vidimo prototipe šablon, ki upravljajo s tipi, ki ustrezajo konceptoma *Type* in *Typed*, v programski kodi 4.11 pa njihovo uporabo, da dosežemo dedukcijo tipa na enak način kot z uporabo izraza *decltype*.

```
concept Typed = /*...*/
template <Typed T> struct get_type;
...
concept Type = /*...*/
template <Type T> struct get_reflected_type;
```

Programska koda 4.10: Koncepta *Typed* in *Type*.

```
int v;
using v_m = reflexpr(v); // reflects v
using t_m = get_type_t<v_m>; // reflects int
get_reflected_type_t<t_m> x; // x is of type int
decltype(v) y; // y is of type int
```

Programska koda 4.11: Uporaba konceptov *Type* in *Typed* za dedukcijo tipa.

Šablonske strukture, ki smo jih pregledali v tem poglavju, bodo omogočile, da za določen razred pridobimo seznam vseh polj in metod ter njihovih kazalcev. S takšnimi zmožnostmi je omogočena implementacija knjižnice za refleksijo v času izvajanja, pri čemer knjižnici ne bo potrebno specificirati, katera polja in metode vsebuje posamezen razred.

Programska koda [15] 4.12 prikazuje primer uporabe. Izraz *reflexpr(S)* vrne neimenovan tip, ki ustreza konceptu *Record*, torej lahko uporabimo šablonsko funkcijo *get_public_data_members_t*. Funkcija bo prav tako vrnila neimenovan tip, ki pa v tem primeru ustreza konceptu *ObjectSequence*. S tem tipom lahko uporabimo šablonsko funkcijo *get_element_t*, ki vrne 1. element. Vrnjen element, ki refleksira prvo polje v strukturi *S*, definiramo s tipom, ki ga imenujemo *FirstDataMember*. Ker tip, ki smo ga imenovali *FirstDataMember*, ustreza konceptu *Named*, ga lahko uporabimo s šablonsko funkcijo *get_name_v*, ki vrne ime polja v obliki niza, torej *„i“*. Tip *FirstDataMember* ustreza tudi konceptu *Typed*, zato lahko uporabimo šablonsko funkcijo *get_type_t*, ki vrne tip, ki refleksira tip polja *FirstDataMember*. Iz slednjega tipa in z uporabo šablonske funkcije *get_name_v* dobimo niz *„int“*. V zadnji vrstici vidimo, da lahko deklariramo spremenljivko *j*, tip pa določimo z uporabo refleksije, torej istega tipa prvega polja v strukturi *S*. Če bi spremenili tip prvega polja v strukturi *S*, bi s tem avtomatično spremenili tudi tip deklarirane spremenljivke *j*.

```
struct S {
    int i;
};

using FirstDataMember =
    get_element_t<
        get_public_data_members_t<reflexpr(S)>,
        0
    >;

void fcn() {
    string memberName = get_name_v<FirstDataMember>;
    string typeName = get_name_v<get_type_t<FirstDataMember>>;

    get_reflected_type_t<
        get_type_t<FirstDataMember>
    > j = 3;

    ...
}
```

Programska koda 4.12: Primer iz konference C++ Now 2019.

Uradni predlog za refleksijo v času prevajanja prinaša še veliko drugih zmožnosti, ki so opisane v tehnični specifikaciji [21]. Kot je že bilo omenjeno, se bo po vsej verjetnosti spremenil način pisanja refleksijske kode, sama funkcionalnost pa naj bi ostala enaka ali pa zelo podobna.

4.2 Pregled refleksijskih knjižnic

Vse refleksijske knjižnice v programskem jeziku C++ imajo nekaj skupnega. Ker jezik nima popolne zmožnosti refleksije, imajo knjižnice postopek za registracijo posameznih elementov programske kode. Če želimo refleksirati določen razred, ga moramo registrirati z refleksijsko knjižnico. Nekatere

knjižnice so pri tem enostavnejše in nekatere manj. Poleg registracije elementov, ki jih želimo refleksirati, morajo knjižnice imeti tudi enostaven način za pridobivanje registriranih informacij.

Ker ima programski jezik C++ zmožnost refleksije določenih aspektov, nam to lahko poenostavi registracijo določenih elementov. Pri modernih refleksijskih knjižnicah nam zaradi tega pri registraciji razredov ni potrebno navajati tipov polj ali tipov argumentov metod, ampak moramo v večini primerov polja in metode samo našteti, kar precej poenostavi uporabo.

Enostavno registracijo elementov lahko vidimo tudi kot prednost pred tem, da registracije elementov ne bi imeli, saj nas lahko zavaruje pri tem, da spremenimo ime polja ali metode, pri čemer se določena programska koda z uporabo refleksije zanaša na spremenjeno ime polja ali metode, prevajalnik pa nas pri tem ne bo opozoril. Če moramo ime polja ali metode spremeniti na dveh mestih, se s tem morda bolj zavedamo posledic takšne spremembe. Registracijo elementov lahko razumemo tudi kot dodaten nivo enkapsulacije, s čimer lahko določamo, kaj je refleksibilno in kaj ne.

Nekatere knjižnice namesto ročne registracije elementov uporabljajo tudi generator registracijske programske kode. To je omogočeno z uporabo prevajalniške infrastrukture LLVM, ki z modularizacijo omogoča ločitev prevajalnika od razčlenjevalnika programske kode. Ker je mogoče razviti tudi module, pri katerih namen ni prevajanje programske kode, je mogoče razviti modul, ki iz razčlenjene programske kode uporabniku poda abstraktno sintaksno drevo za posamezno prevajalno enoto. Na tem mestu bi omenili knjižnico CppAST [5], ki se jo lahko vključi v programsko kodo C++ z namenom, da bi lahko pridobili vpogled v abstraktno sintaksno drevo določene programske kode. Uporaba takšne knjižnice nam lahko poenostavi razvoj avtomatičnega generiranja registracijske programske kode.

Ima pa uporaba takšne knjižnice tudi nekaj negativnih posledic. Programska koda, ki je predmet takšne knjižnice, mora biti v obliki, ki jo zahteva razčlenjevalnik kode Clang, ki je del prevajalniške infrastrukture LLVM. Če se programska koda prevede s prevajalnikom MSVC na platformi Win-

dows, še ne pomeni, da se bo prevedla tudi s prevajalnikom Clang/LLVM, saj nekateri prevajalniki strožje, nekateri manj sledijo standardu C++. Prevajalnik MSVC je manj strog kot prevajalnik Clang/LLVM in manj strog kot prevajalnik GCC, poleg tega pa je potrebno tudi upoštevati, da se velikokrat v programski kodi uporabi zaglavne datoteke, ki so napisane za določeno platformo (primer: Windows.h) in se zato ne prevedejo z vsemi prevajalniki.

Zaradi tega je pomembno, da refleksijska knjižnica ponuja enostaven postopek za registracijo elementov, saj je na takšen način vzdrževanje programske kode lažje, kot če v proces prevajanja programske kode vpeljemo generator registracijske programske kode, ki ne bo deloval v vseh okoliščinah.

V nadaljevanju bomo uporabili strukturo *ExampleStruct* in razred *ExampleClass*, pri katerih bo namen le-ta, da naredimo primerjavo funkcionalnosti in uporabe med obstoječimi refleksijskimi knjižnicami in knjižnico, ki smo jo razvili za to diplomsko delo. Struktura *ExampleStruct* in razred *ExampleClass* v programski kodi 4.13 uporabljata čim več različnih tipov v definiciji polj in prototipih metod. Razred *ExampleClass* je sestavljen tudi s strukturo *ExampleStruct*. Prav tako vidimo, da so poleg vrednosti pri nekaterih metodah uporabljene še reference in kazalci. Imamo tudi dve metodi z istim imenom (*add*) in dve metodi z istim imenom, argumenti in različnim kvalifikatorjem (*getValues*). Metoda *testPerformance* nam bo služila za primerjavo hitrosti izvajanja.

```
1  struct ExampleStruct {
2      int mainValue;
3      std::map<std::string, std::vector<int>> values;
4  };
5
6  class ExampleClass {
7  private:
8      float m_factor;
9      ExampleStruct m_struct;
10
11 public:
```

```
12      ExampleClass();
13      ExampleClass(std::string key, std::vector<int> values);
14      ExampleClass(const ExampleClass& other);
15      void clear() ;
16      void setFactor(int factor);
17      void add(std::string key, std::vector<int> values);
18      void add(const ExampleStruct& exampleStruct);
19      std::vector<float> getValues(const char *key) const;
20      std::vector<float> getValues(const char *key);
21      double testPerformance(std::string key, std::vector<int> values);
22  };
```

Programska koda 4.13: Testna struktura *ExampleStruct* in razred *ExampleClass*.

4.2.1 RTTR

Knjižnica RTTR ponuja refleksijo v času izvajanja s podobno funkcionalnostjo, kot jo ponuja programski jezik Java.

Programska koda 4.14 prikazuje registracijo razredov s knjižnico RTTR. Registracija razredov ni enostavna, saj se pri pisanju takšne programske kode lahko na veliko mestih zmotimo. Prav tako je pri registraciji elementov poleg kazalca, ki kaže na element, potrebno podati tudi niz, pod katerim se element registrira. Polje ali metodo razreda je mogoče registrirati pod drugačnim imenom, kot ga polje ali metoda ima. Uporabnik se pri podajanju niza lahko tudi zmoti, pri čemer ga prevajalnik ne bo opozoril. Če v razredu obstaja več prekrivnih metod, je potrebno uporabiti posebno šablonsko funkcijo, ki za parameter pričakuje tip metodnega kazalca, kar prinaša dodatno kompleksnost.

```
1   #include <rttr/registration>
2   using namespace std;
3
```

```
4  struct ExampleStruct
5  {
6      ...
7  };
8
9  struct ExampleClass
10 {
11     ...
12 };
13
14 RTTR_REGISTRATION
15 {
16     rttr::registration::class_<ExampleStruct>("ExampleStruct")
17         .constructor<>()
18         .property("mainValue", &ExampleStruct::mainValue)
19         .property("values", &ExampleStruct::values)
20         ;
21
22     rttr::registration::class_<ExampleClass>("ExampleClass")
23         .constructor<>()
24         .constructor<string, vector<int>>()
25         .constructor<const ExampleClass&>()
26         .method("clear", &ExampleClass::clear)
27         .method("setFactor", (&ExampleClass::setFactor))
28         .method("add",
29             rttr::select_overload<
30                 void(string, vector<int>)
31             >(&ExampleClass::add))
32         .method("add",
33             rttr::select_overload<
34                 void(const ExampleStruct&)
35             >(&ExampleClass::add))
36         .method("getValues",
37             rttr::select_overload<
38                 vector<float>(const char*)
39             >(&ExampleClass::getValues))
40         .method("getValues",
```

```
            rttr::select_overload<
                vector<float>(const char*) const
            >(&ExampleClass::getValues))
        .method(
            "testPerformance",
            &ExampleClass::testPerformance
        )
        ;
}
```

Programska koda 4.14: Primer registracije razredov s knjižnico RTTR.

V programski kodi 4.15 vidimo razširitev makroja *RTTR_REGISTRATION*. Deklarira se globalni objekt strukture *rttr__auto__register__* z imenom, ki se generira na podlagi vrstice z namenom, da bi bilo ime unikatno. Struktura *rttr__auto__register__* ima konstruktor, ki pokliče registracijsko funkcijo *rttr_auto_register_reflection_function_*, kar pomeni, da se bo registracijska funkcija izvedla avtomatično, saj se globalni objekt avtomatično ustvari, ko se program požene.

```
static void rttr_auto_register_reflection_function_();
namespace
{
    struct rttr__auto__register__
    {
        rttr__auto__register__()
        {
            rttr_auto_register_reflection_function_();
        }
    };
}
static const rttr__auto__register__
RTTR_CAT(auto_register__, __LINE__);
static void rttr_auto_register_reflection_function_()
{
```

```
16      rttr::registration::class_<MyClass>("ExampleStruct")
17      ...
18      ;
19
20      rttr::registration::class_<MyClass>("ExampleClass")
21      ...
22      ;
23  }
```

Programska koda 4.15: Razširjen makro *RTTR_REGISTRATION*.

Pri tem pa lahko nastane problem. Standard C++ ne definira vrstnega reda, po katerem se globalni objekti inicializirajo v primeru, ko imamo globalne objekte v različnih prevajalnih enotah, ki so del končne povezane izvršljive datoteke. To vključuje tudi vse prevajalne enote, ki so del vključenih statičnih knjižnic.

Standard C++ definira samo to, da se mora inicializacija globalnih objektov določene prevajalne enote zgoditi pred uporabo katerekoli funkcije ali globalnega objekta, ki je del te prevajalne enote. To pomeni, da se registracijska programska koda morda ne bo izvedla pred izvedbo funkcije *main*, ki je morda v drugi prevajalni enoti. Pri pridobivanju refleksijskih informacij za posamezen razred se lahko zgodi, da jih ne bomo dobili, če pred tem nismo poklicali nobene funkcije, razredne metode ali uporabili globalnega objekta iz prevajalne enote, ki vsebuje registracijsko programsko kodo. Problem je opazen takrat, ko se uporablja statične knjižnice ali pa so vse prevajalne enote del končne povezane izvršljive datoteke.

Pri dinamičnem nalaganju knjižnic ta problem ni opazen, saj za inicializacijo globalnih objektov poskrbi knjižnica za izvajanje C/C++ (angl. C/C++ run-time library), ki je del operacijskega sistema. Pri uporabi dinamičnega nalaganja knjižnic smo lahko prepričani, da bodo vsi globalni objekti ustvarjeni v trenutku, ko se knjižnica naloži v procesni prostor.

Če ne uporabljamo dinamičnega nalaganja knjižnic, problem na začetku

morda ne bo niti opazen, ko pa se pojavi, pa je vzrok zelo težko odkriti brez poznavanja tovrstne problematike. Problem je velikokrat omenjen kot neuspeh vrstnega reda statične inicializacije (angl. static initialization order fiasco).

Programska koda 4.16 prikazuje uporabo knjižnice RTTR. V prevajalno enoto vključimo definicijo strukture *ExampleStruct*, ki je definirana v zaglavni datoteki *ExampleStruct.h*. Razred *ExampleClass* ni definiran v zaglavni datoteki. V procesni prostor se naloži knjižnica *plugin_example.dll*, ki vsebuje implementacijo razreda *ExampleClass* ter registracijo strukture *ExampleStruct* in razreda *ExampleClass*. S klicem *library::get_types* dobimo seznam vseh struktur in razredov, ki so registrirani s knjižnico. V tem primeru sta to struktura *ExampleStruct* in razred *ExampleClass*. S klicem *type::get_by_name("ExampleClass")* dobimo objekt razreda, ki refleksira razred *ExampleClass* in s katerim lahko ustvarimo objekt razreda *ExampleClass*. Tukaj moramo biti pozorni na to, da lahko ustvarimo objekt razreda *ExampleClass*, kljub temu da razred *ExampleClass* v prevajalni enoti ni definiran oziroma ne vključujemo zaglavne datoteke, ki bi definirala razred *ExampleClass*.

```
1  #include <rttr/type>
2  #include <chrono>
3
4  #include "ExampleStruct.h"
5  using namespace std;
6  using namespace rttr;
7  library lib("plugin_example");
8
9  int main(int argc, char** argv)
10 {
11     if (!lib.load()) {
12         cerr <<
13             lib.get_error_string() <<
14             endl;
15         return -1;
```

```
16      }
17
18      for (type t : lib.get_types()) {
19          if (t.is_class() && !t.is_wrapper())
20              cout << t.get_name() << endl;
21      }
22
23      ExampleStruct exampleStructObj;
24      exampleStructObj.mainValue = 1;
25      exampleStructObj.values["A"] = { 1, 2, 3 };
26      type exampleClass = type::get_by_name("ExampleClass");
27
28      for (property p : type::get(exampleStructObj).get_properties()) {
29          cout << p.get_name() << endl;
30      }
31      for (method m : exampleClass.get_methods()) {
32          cout << m.get_name() << endl;
33      }
34
35      if (!type::get(exampleStructObj).set_property_value(
36          "mainValue",
37          exampleStructObj,
38          2
39      )) {
40          cerr << "Failed to set mainValue";
41          return -1;
42      }
43
44      variant exampleClassObj = exampleClass.create(
45          { string("B"), vector<int>{4, 5, 6} }
46      );
47      if (!exampleClassObj.is_valid()) {
48          cerr << "Failed to instantiate ExampleClass";
49          return -1;
50      }
51
52      variant ret = exampleClass.invoke(
```

```
53          "setFactor",
54          exampleClassObj,
55          { 2.0f }
56      );
57      if (!ret.is_valid()) {
58          cerr << "Failed to call setFactor(float)";
59          return -1;
60      }
61
62      ret = exampleClass.invoke(
63          "add",
64          exampleClassObj,
65          { string("C"), vector<int>{7, 8, 9} }
66      );
67      if (!ret.is_valid()) {
68          cerr
69              << "Failed to call add(string, vector<int>)";
70          return -1;
71      }
72
73      ret = exampleClass.invoke(
74          "add",
75          exampleClassObj,
76          { exampleStructObj }
77      );
78      if (!ret.is_valid()) {
79          cerr <<
80              "Failed to call add(const ExampleStruct &)";
81          return -1;
82      }
83
84      const char* str = "A";
85      ret = exampleClass.invoke(
86          "getValues",
87          exampleClassObj,
88          { str }
89      );
```

```
90      if (!ret.is_valid()) {
91          cerr
92              << "Failed to call getValues(const char *)";
93          return -1;
94      }
95
96      vector<float> values =
97          ret.get_wrapped_value<vector<float>>();
98
99      return 0;
100 }
```

Programska koda 4.16: Primer uporabe knjižnice RTTR.

Poleg ustvarjanja objektov brez definicije razreda lahko z objektom razreda, ki refleksira razred *ExampleClass*, pridobimo ali nastavimo vrednosti polj in kličemo metode objektov razreda *ExampleClass*. Metode lahko pokličemo z uporabo metode *type::invoke*, kjer moramo podati objekt, na katerem se metoda pokliče, in vse argumente, ki jih sprejme metoda, ki jo želimo poklicati. Če se podani tipi argumentov ne ujemajo s pričakovanimi tipi metode, se metoda ne bo poklicala. Na podoben način lahko z uporabo metod *type::get_property_value* in *type::set_property_value* pridobivamo ali nastavljamo vrednosti polj razredov.

Če želimo klicati metode razredov, moramo poznati tipe argumentov in vrnjene vrednosti razrednih metod v času prevajanja, sicer klicanje metode ne bo uspešno. V času izvajanja lahko izberemo, katera metoda se bo izvedla, tipi argumentov in vrnjene vrednosti pa so določeni v času prevajanja s klicem metode *method::invoke*. Podobno velja pri pridobivanju ali spreminjanju vrednosti polj razredov.

Na primeru prav tako vidimo, da moramo preverjati vrnjeno vrednost metode *method::invoke*, če želimo biti prepričani, da se je metoda uspešno poklicala, saj knjižnica ne uporablja izjem. Pri tem je treba poudariti, da v primeru napake pri klicanju metode ne moremo ugotoviti, zakaj je prišlo do

napake, saj vrnjena vrednost vsebuje samo informacijo o tem, ali se je metoda poklicala. To je eden izmed glavnih razlogov, zakaj takšne knjižnice ne moremo uporabiti, saj je pri uporabi zelo pomembno, da čim hitreje ugotovimo vzrok napake, če nočemo upočasnjevati razvoja.

Na koncu programske kode 4.16 vidimo tudi način pridobivanja vrnjene vrednosti metode *getValues*. Če bi v tem primeru zahtevali drugačen tip kot tip vrnjene vrednosti metode, bi s tem povzročili nedefinirano obnašanje, saj knjižnica brez preverjanja pretvarja z uporabo izraza *reinterpret_cast*.

Problem knjižnice RTTR je tudi v tem, da ne upošteva kvalifikatorjev razrednih metod. Če registriramo več prekrivnih metod z istim imenom, parametri in različnimi kvalifikatorji, se bo vedno poklicala tista metoda, ki je bila prva registrirana. Pravilno bi bilo, da se pokliče tista metoda, ki se po pravilih mora poklicati glede na kvalifikator objekta razreda, na kateri je metoda klicana.

4.2.2 refl-cpp

Knjižnica refl-cpp omogoča refleksijo v času prevajanja. To pomeni, da se vse refleksijske informacije za registrirane elemente definirajo v času prevajanja. Prav tako se vsa refleksijska programska koda izvede v času prevajanja.

Programska koda 4.17 prikazuje registracijo elementov za ekvivalenten primer, kot smo ga videli pri registraciji elementov s knjižnico RTTR. Makro *REFL_AUTO* s knjižnico registrira vse potrebne refleksijske informacije v času prevajanja. Razlog, zakaj se uporabi makro, je, da uporabniku poenostavi registracijo elementov, saj je registracijska programska koda, v katero se makro razširi, zelo kompleksna. Uporabnik knjižnice se mora naučiti, na kakšen način se uporabi registracijski makro. V tem pogledu je knjižnica refl-cpp veliko enostavnejša za uporabo od knjižnice RTTR.

```
1  #include "refl.hpp"
2
3  struct ExampleStruct
```

```
{
    ...
};

struct ExampleClass
{
    ...
};

REFL_AUTO(
    type(ExampleStruct, bases<>),
    field(mainValue),
    field(values)
)

REFL_AUTO(
    type(ExampleClass, bases<>),
    func(clear),
    func(setFactor),
    func(add),
    func(getValues),
    func(testPerformance)
)
```

Programska koda 4.17: Primer registracije elementov s knjižnico refl-cpp.

V programski kodi 4.17 vidimo, da pri prekrivnih metodah z istim imenom in različnimi argumenti ni potrebno registrirati vsake metode posebej in pri tem podajati informacije o tipih, kot je to pri knjižnici RTTR. Razlog je v tem, da knjižnica refl-cpp ne uporablja metodnih kazalcev, preko katerih bi se izvajale metode, pri čemer dejansko prihaja do dvoumnosti, saj je pri pridobivanju kazalca potrebno določiti, za katero prekrivno metodo ga želimo dobiti. Namesto tega se makro razširi v programsko kodo, ki pokliče dejansko metodo brez uporabe metodnega kazalca, pri čemer se bo vedno poklicala prava prekrivna metoda glede na uporabo, tako kot se to zgodi brez refleksije.

Klicanje brez metodnega kazalca je omogočeno na način, da se za vsako posamezno metodo vsakega razreda definira nov tip oziroma struktura s statično šablonsko metodo *invoke*, ki vsebuje klic na dejansko metodo. Parametri se lahko prenesejo brez potrebe po pretvarjanju, saj metoda *invoke* v tem primeru ni virtualna oziroma je šablonska in zato lahko sprejme poljubno število argumentov, ki so lahko kateregakoli tipa.

Če naredimo primerjavo s knjižnico RTTR, ugotovimo, da slednja ne definira ločenega tipa za vsako razredno metodo posebej, ampak za vsak tip razredne metode. Knjižnica RTTR v času izvajanja za vse registrirane metode (in polja) ustvari objekte, ki so različnega tipa glede na tip metode (in polja) in imajo enak vmesnik. Ti objekti zato namesto dejanskega klica metode vsebujejo metodni kazalec, preko katerega je omogočen klic metode. Ker morajo imeti vsi objekti za upravljanje z metodami v knjižnici RTTR nekaj skupnega, da lahko z njimi v času izvajanja upravljamo z uporabo polimorfizma, je metoda *invoke* virtualna, zaradi česar je potrebno imeti enoten vmesnik za vse različne metode in zaradi česar je potrebno argumente v času izvajanja pretvarjati v enotno obliko.

Programska koda 4.18 prikazuje razširjene makroje registracijske programske kode razreda *ExampleClass* (zaradi prevelike dolžine je programska koda skrajšana).

```
1  namespace refl_impl::metadata {
2      template<> struct type_info__<ExampleClass> {
3          typedef ExampleClass type;
4          static constexpr auto attributes{
5              ::refl::detail::make_attributes<
6              ::refl::attr::usage::type>(bases<>)
7          };
8          static constexpr auto name{
9              ::refl::util::make_const_string("ExampleClass")
10         };
11         static constexpr size_t member_index_offset = 5 + 1;
12         template <size_t N, typename = void> struct member {};
```

```
13
14          template<typename Unused__>
15          struct member<6 - member_index_offset, Unused__> {
16              typedef ::refl::member::function member_type;
17              static constexpr auto name{
18                  ::refl::util::make_const_string("clear")
19              };
20              static constexpr auto attributes{
21                  ::refl::detail::make_attributes<
22                      ::refl::attr::usage::type
23                  >()
24              };
25          public:
26              template<typename Self, typename... Args>
27              static constexpr auto invoke(
28                  Self&& self,
29                  Args&&... args
30              ) ->
31              decltype(
32                  std::declval<Self>().clear(
33                      ::std::declval<Args>()...
34                  )
35              ) {
36                  return ::std::forward<Self>(self).clear(
37                      ::std::forward<Args>(args)...
38                  );
39              }
40              template<typename... Args>
41              static constexpr auto invoke(Args&&... args) ->
42              decltype(
43                  ::refl::detail::head_t<type, Args...>::clear(
44                      ::std::declval<Args>()...
45                  )
46              ) {
47                  return
48                      ::refl::detail::head_t<type, Args...>::clear(
49                          ::std::forward<Args>(args)...
```

```
                    );
                }
                template <typename Dummy = void>
                static constexpr auto pointer() ->
                decltype(&::refl::detail::head_t<
                    type, Dummy
                >::clear) {
                    return &::refl::detail::head_t<
                        type, Dummy
                    >::clear;
                }

                template <typename Proxy> struct remap {
                    template <typename... Args> decltype(auto) clear(
                        Args&&... args
                    ) {
                        return Proxy::invoke_impl(
                            static_cast<Proxy&>(*this),
                            ::std::forward<Args>(args)...
                        );
                    }
                    template <typename... Args> decltype(auto) clear(
                        Args&&... args
                    ) const {
                        return Proxy::invoke_impl(
                            static_cast<const Proxy&>(*this),
                            ::std::forward<Args>(args)...
                        );
                    }
                };
            };

            template<typename Unused__> struct member<
                7 - member_index_offset, Unused__
            > {
                ...
            };
```

```
87
88          ...
89
90          static constexpr size_t member_count{
91              9 - member_index_offset
92          };
93      };
94  }
```

Programska koda 4.18: Razširitev registracijskih makrojev knjižnice refl-cpp.

Registracija posameznega razreda se začne s specializacijo šablonske strukture *type_info__* za razred, ki ga refleksiramo. V tem primeru se šablonska struktura specializira za razred *ExampleClass*. Znotraj te specializacije vidimo, da obstaja še šablonska struktura *member*, ki je specializirana z indeksom, ki ponazarja vrstni red polja oziroma metode. Šablonska struktura *member* ponuja funkcionalnost, ki omogoča refleksijo posameznega polja oziroma metode razreda. Tako vidimo, da ima vsaka specializacija šablonske strukture *member* nekatera polja in metode, ki omogočajo refleksijo določenega polja oziroma metode. Statična šablonska metoda *invoke* pokliče metodo, ki jo šablonska struktura *member* refleksira, brez uporabe metodnega kazalca. Za vsak razred, ki ga želimo refleksirati, bomo z registracijo definirali specializacijo razreda *type_info__*, ki vsebuje specializacije šablone *member* glede na registrirane elemente, ki so del razreda.

Prednost te knjižnice je, da je mogoče refleksirati tudi šablonske metode, saj ne uporablja metodnih kazalcev, ki jih po standardu C++ ni mogoče dobiti za šablonske metode. Slabost knjižnice refl-cpp pa se kaže predvsem v kompleksnosti uporabe. Morda to najbolje ilustriramo ob dejstvu, da ne moremo enostavno iterirati skozi vse registrirane elemente nekega razreda v času izvajanja.

V programski kodi 4.19 vidimo nepravilno uporabo knjižnice. Čeprav je šablonska struktura *type_info__<ExampleClass>::member* parameterizirana

z indeksom, za ta indeks ne moremo uporabiti spremenljivke i, ker je za parametre šablone mogoče uporabiti samo tisto, kar je znano v času prevajanja, spremenljivka i pa se določi in povečuje v času izvajanja.

```
ExampleClass example;
for (int i = 0; i < type_info__<ExampleClass>::member_count; i++)
{
    if (type_info__<ExampleClass>
        ::members<i>                          // ERROR
        ::name == "clear") {
            type_info__<ExampleClass>
                ::members<i>                  // ERROR
                ::invoke(example);
    }
}
```

Programska koda 4.19: Nepravilna uporaba knjižnice refl-cpp.

Za ta namen so sicer na voljo šablonske funkcije, ki poenostavijo uporabo. S programsko kodo 4.20 lahko izpišemo imena vseh elementov, ki so registrirani z razredom *ExampleClass*.

```
for_each(refl::reflect<ExampleClass>().members, [&](auto member) {
    cout <<
        "  " <<
        get_display_name(member) <<
        " (" <<
        member.name <<
        ") = ";
    cout << endl;
});
```

Programska koda 4.20: Prikaz vseh elementov razreda ExampleClass.

Programska koda 4.21 prikazuje skoraj ekvivalentno programsko kodo,

kot smo jo videli v primeru uporabe knjižnice RTTR. Knjižnica refl-cpp ne ponuja zmožnosti za ustvarjanje objektov, zaradi česar v prevajalno enoto vključimo tudi zaglavno datoteko *ExampleClass.h*, s čimer bomo lahko ustvarili objekt razreda *ExampleClass*. Če katerokoli metodo pokličemo z drugačnimi argumenti, kot jih metoda pričakuje, se programska koda ne bo prevedla.

```
#include "ExampleStruct.h"
#include "ExampleClass.h"

using namespace std;
using refl::reflect;
using refl::util::find_one;
using refl::util::for_each;

void example() {
    ExampleStruct exampleStructObj;
    exampleStructObj.mainValue = 1;
    exampleStructObj.values["A"] = { 1, 2, 3 };

    ExampleClass exampleClassObj(
        string("B"),
        vector<int>{4, 5, 6}
    );

    for_each(reflect<ExampleStruct>().members, [](auto m) {
        cout << get_display_name(m) << endl;
    });

    for_each(reflect<ExampleClass>().members, [](auto m) {
        cout << get_display_name(m) << endl;
    });

    constexpr auto propMainValue = find_one(
        reflect<ExampleStruct>().members,
        [](auto m) {
```

```
30              return m.name == "mainValue";
31          }
32      );
33      propMainValue(exampleStructObj, 5);
34
35      constexpr auto funcSetFactor = find_one(
36          reflect<ExampleClass>().members,
37          [](auto m) {
38              return m.name == "setFactor";
39          }
40      );
41      funcSetFactor(exampleClassObj, 2.0f);
42
43      constexpr auto funcAdd = find_one(
44          reflect<ExampleClass>().members,
45          [](auto m) {
46              return m.name == "add";
47          }
48      );
49      funcAdd(
50          exampleClassObj,
51          string("C"),
52          vector<int>{7, 8, 9}
53      );
54      funcAdd(exampleClassObj, exampleStructObj);
55
56      constexpr auto funcGetValues = find_one(
57          reflect<ExampleClass>().members,
58          [](auto m) {
59              return m.name == "getValues";
60          }
61      );
62      vector<float> values = funcGetValues(
63          exampleClassObj,
64          "A"
65      );
```

```
66 }
```

Programska koda 4.21: Primer uporabe knjižnice refl-cpp.

S knjižnico za refleksijo v času prevajanja ne moremo poklicati metode z drugačnimi argumenti, kot jih metoda pričakuje. Čeprav je tako bolj varno, saj bo prevajalnik preprečil prevajanje napačne programske kode, nam to prinese določene omejitve.

V primeru knjižnice za refleksijo v času izvajanja lahko pokličemo metodo z drugačnimi argumenti, kot jih metoda pričakuje, saj prevajalnik ni zmožen preveriti pravilnosti klica, za katerega se odločimo med izvajanjem programske kode. Razlog za to je v načinu implementacije metode *invoke*, ki je precej drugačen v primeru refleksijske knjižnice v času prevajanja in refleksijske knjižnice v času izvajanja. Glavna razlika je v tem, da s knjižnico v času prevajanja kličemo metode brez metodnih kazalcev in brez pretvarjanja argumentov, saj imamo za vsako metodo ločen tip s specifično implementacijo statične šablonske metode invoke, zaradi česar ne potrebujemo enotnega vmesnika in pretvarjanja argumentov. Pri knjižnici za refleksijo v času izvajanja pa imamo za vsako metodo ločen objekt, ki je ustvarjen v času izvajanja in vsebuje metodni kazalec z ostalimi informacijami. Ker je klic izveden z uporabo virtualne metode invoke, prevajalnik v času prevajanja zaradi uporabe polimorfizma ne ve, na katero metodo se klic nanaša, saj je to znano šele v času izvajanja. Iz tega razloga moramo v času izvajanja tipe argumentov preveriti, da v primeru neujemanja vrnemo napako in preprečimo nedefinirano obnašanje.

Knjižnica refl-cpp vsebuje tudi nekatera pomagala za čas izvajanja, ki omogočijo izbiro in klicanje metode na podoben način kot z uporabo knjižnice za refleksijo v času izvajanja. Programska koda 4.22 prikazuje klic metode *getValues* na način, ki omogoči, da se programska koda prevede tudi v primeru, ko se podani tipi argumentov ne ujemajo s pričakovanimi in v primeru, ko klicana metoda ne obstaja. V primeru neuspešnega klica bo metoda v

času izvajanja *refl::runtime::invoke* vrgla izjemo s sporočilom, da se podani argumenti ne ujemajo s pričakovanimi. Tudi knjižnica refl-cpp v primeru napake ne sporoči tipov argumentov, ki so bili klicu podani. Sicer pri uporabi metode *refl::runtime::invoke* ne moremo govoriti kot o delu knjižnice za refleksijo v času prevajanja, saj vsebuje tudi programsko kodo, ki se izvede med izvajanjem programa.

```
vector<float> values =
    runtime::invoke<vector<float>>(
        exampleClassObj,
        "getValues",
        "A"
    );
```

Programska koda 4.22: Primer klicanja metode s knjižnico refl-cpp z elementi refleksije v času izvajanja.

Programska koda 4.23 prikazuje implementacijo metode *refl::runtime::invoke*. Znotraj metode vidimo uporabo šablonske strukture *std::is_invocable_r_v*, ki je postala na voljo s standarom C++17 in s katero je mogoče v času prevajanja ugotoviti, če je določeno metodo mogoče poklicati z določenimi argumenti. Če je metodo mogoče poklicati s podanimi argumenti, je metoda kandidat za primerjavo imena metode v času izvajanja. Če se katerakoli metoda, ki jo je mogoče poklicati s podanimi argumenti, ujema s podanim imenom metode v času izvajanja, se metoda pokliče. V času prevajanja se izloči tiste metode, ki jih ni mogoče poklicati s podanimi parametri, v času izvajanja pa se iz ostanka metod naredi primerjava imena. To potrjuje pravilo, da s knjižnico za refleksijo v času prevajanja lahko implementiramo tudi knjižnico za refleksijo v času izvajanja.

```
template <typename U, typename T, typename... Args>
U invoke(T&& target, const char* name, Args&&... args)
{
```

```
4       using type = std::remove_reference_t<T>;
5       static_assert(refl::trait::is_reflectable_v<type>, "Unsupported
        ↪  type!");
6       typedef type_descriptor<type> type_descriptor;
7
8       std::optional<U> result;
9
10      bool found{ false };
11      for_each(type_descriptor::members, [&](auto member) {
12          using member_t = decltype(member);
13          if (found) return;
14
15          if constexpr (std::is_invocable_r_v<
16              U,
17              decltype(member),
18              T,
19              Args...
20          >) {
21              if constexpr (trait::is_field_v<member_t>) {
22                  if (!std::strcmp(member.name.c_str(), name)) {
23                      result.emplace(member(
24                          target,
25                          std::forward<Args>(args)...
26                      ));
27                      found = true;
28                  }
29              }
30              else if constexpr (trait::is_function_v<member_t>) {
31                  if (!std::strcmp(member.name.c_str(), name)) {
32                      result.emplace(member(
33                          target,
34                          std::forward<Args>(args)...
35                      ));
36                      found = true;
37                  }
38              }
39          }
```

```
40      });
41
42      if (found) {
43          return std::move(*result);
44      }
45      else {
46          throw std::runtime_error(std::string("The member ")
47              + type_descriptor::name.str() + "::" + name
48              + " is not compatible with the provided parameters or
                ↪  return type, is not reflected or does not exist!");
49      }
50  }
```

Programska koda 4.23: Implementaicija metode refl::runtim::invoke.

Iz neznanega razloga metoda *refl::runtime::invoke* ne zna poklicati metode, ki nič ne vrača, saj privzame, da vse metode nekaj vrnejo. Vsekakor bi bilo mogoče metodo popraviti na način, da bi delovala tudi za metode, ki nič ne vračajo.

Vprašanje, ki se tukaj postavlja, je, zakaj bi se sploh odločali o klicanju metod v času izvajanja? Odgovor je v tem, da si želimo manj sklopljen (angl. loosely coupled) sistem v času prevajanja. Stremimo k temu, da ima vsaka komponenta svoj razvojni cikel, da se ločeno prevaja, testira in dostavlja. Če hočemo imeti generični strežnik REST, ki v času prevajanja ni sklopljen z razredi, za katere bo ponujal funkcionalnost, obenem pa bi radi klicali metode teh razredov, moramo imeti zmožnost izbiranja in klicanja metod v času izvajanja.

Vedeti moramo, za kakšen namen bi uporabili knjižnico za refleksijo v času prevajanja, brez elementov refleksije v času izvajanja. Če se vrnemo k primeru generičnega razčlenjevalnika ukazne vrstice, to pomeni, da pri prevajanju določenega razčlenjevalnika ukazne vrstice potrebujemo tudi izvorno programsko kodo generičnega razčlenjevalnika. Enako velja v primeru uporabe generičnega strežnika REST z implementacijo določenega strežnika

REST. Če imamo izvorno programsko kodo obojega in ob spremembah enega ali drugega oboje prevajamo skupaj, je iz vidika učinkovitosti bolje uporabiti knjižnico za refleksijo v času prevajanja.

Knjižnico refl-cpp bi sicer lahko uporabili tudi kot knjižnico za refleksijo v času izvajanja z dodatnimi pomagali, ki jih knjižnica vsebuje za ta namen. Za uporabo v našem dinamičnem sistemu knjižnici refl-cpp manjka zmožnost, s katero bi lahko dinamično naložili komponente sistema v procesni prostor in ustvarili objekte razredov, ki niso definirani v prevajalni enoti ali katerikoli zaglavni datoteki, ki jo prevajalna enota vključuje.

Poglavje 5

Refleksijska knjižnica Jobjects++

Za to diplomsko delo smo razvili novo refleksijsko knjižnico z imenom Jobjects++ [10]. Knjižnica je namenjena uporabi refleksije v času izvajanja in je zato bolj podobna knjižnici RTTR kot knjižnici refl-cpp. Tako kot pri knjižnici RTTR je tudi pri knjižnici Jobjects++ potrebno registrirati razrede in njihove elemente, ki jih želimo refleksirati. Pri registraciji je sintaksa sicer enostavnejša od knjižnice RTTR, saj knjižnica Jobjects++ pri registraciji uporablja makroje, podobno kot knjižnica refl-cpp. Funkcionalnost refleksije je zelo podobna knjižnici RTTR, saj omogoča naštevanje razredov, ustvarjanje objektov, naštevanje in nastavljanje razrednih polj ter naštevanje in klicanje razrednih metod.

Poglavitna razlika med knjižnico RTTR in Jobjects++ je v tem, da morajo refleksibilni razredi v knjižnici Jobjects++ dedovati osnovni razred, ki vsebuje vmesnik za refleksijo. Na ta način imajo vsi refleksibilni razredi nekaj skupnega in je zato omogočeno pisanje generične programske kode za vse refleksibilne razrede z uporabo polimorfizma. Poleg tega skupni vmesnik omogoča tudi serializacijo, deserializacijo, vsebuje metodo *toString* in ima zelo podoben vmesnik za refleksijo kot osnovni razred *Object* v programskem jeziku Java, zaradi česar ima knjižnica tudi takšno ime.

5.1 Opis razvoja in uporabljena orodja

Za razvoj smo uporabili razvojno okolje MS Visual Studio 2019 16.4.6 na platformi Windows. V konfiguraciji prevajalnika smo nastavili minimalni standard C++17 in eksperimentalen preprocesor, ki bolj točno sledi standardu C++. Pri razvoju smo uporabili knjižnico jsoncpp, ki ponuja funkcionalnost JSON, in knjižnico C++ REST SDK od podjetja Microsoft, ki nam je pomagala pri razvoju strežnika REST.

Programsko kodo smo prevajali tudi s prevajalnikom GCC 9.2.1 na platformi Linux, saj ima knjižnica Jobjects++ drugačno implementacijo registra razredov za platformo Windows in Linux, pri čemer informacije o refleksiranih razredih pridobimo z naštevanjem simbolov v prevedeni in povezani knjižnici ali izvršljivi datoteki.

Pri prevajanju s prevajalnikom GCC smo opazili večje število sintaktičnih napak, saj je prevajalnik GCC strožji pri upoštevanju standarda C++ kot prevajalnik MSVC.

Pri razvoju smo bili agilni in sprva razvili prototip, ki je implementiral približno enako funkcionalnost knjižnice RTTR skupaj s testno programsko kodo. Ko smo bili zadovoljni z načinom uporabe, smo razvili generični razčlenjevalnik ukazne vrstice in generični strežnik REST. Ko smo s tem potrdili koncept diplomskega dela, smo programsko kodo razdelili na knjižnico in komponente, ki knjižnico uporabljajo. To so odjemalec, testna komponenta, generični razčlenjevalnik ukazne vrstice in generični strežnik REST.

Za testiranje knjižnice smo uporabili programsko kodo, ki uporablja refleksijsko knjižnico. Vsakič, ko smo naredili določeno spremembo v knjižnici, smo na novo prevedli vso programsko kodo, ki uporablja refleksijsko knjižnico, in preverjali, ali se pri izvajanju izpiše enaka vsebina.

Na koncu razvoja smo izmerili hitrost določene funkcionalnosti in rezultate primerjali s knjižnico RTTR in refl-cpp. Potrebnih je bilo še nekaj poglavitnih sprememb, da smo bili s hitrostjo povsem zadovoljni.

5.2 Uporaba knjižnice

5.2.1 Dedovanje osnovnega razreda

V programski kodi 5.1 vidimo način, po katerem obstoječi razred spremenimo, da postane refleksibilen. Vsak razred, ki bi ga radi refleksirali, mora dedovati šablonski razred *Reflectable*, ki je parametriziran z razredom, ki ga refleksiramo. Pri tem je treba poudariti, da gre za vzorec CRTP.

```
1  #include "Reflectable.h"
2  struct ExampleStruct : public Reflectable<ExampleStruct> {
3  ...
4  };
5
6  class ExampleClass : public Reflectable<ExampleClass> {
7  ...
8  };
```

Programska koda 5.1: Primer refleksibilnih struktur in razredov.

Ko dedujemo šablonski razred *Reflectable*, avtomatično dedujemo tudi vmesnik *Object*. V programski kodi 5.2 vidimo vmesnik *Object*. Z vmesnikom lahko pokličemo metodo *Object::getClass*, ki nam bo vrnila razred *Class*, ki je konceptualno ekvivalenten razredu *Class* v programskem jeziku Java. Vmesnik *Object* ponuja tudi metode za serializacijo in deserializacijo.

```
1  class Object {
2  public:
3      virtual const Class &getClass() const = 0;
4      virtual Json::Value serialize() const = 0;
5      virtual const char* toString() const = 0;
6      virtual void deserialize(Json::Value) = 0;
7      virtual void fromString(const char*) = 0;
8      virtual ~Object() {
9      }
```

```
10 };
```

Programska koda 5.2: Vmesnik *Object*.

5.2.2 Način registracije elementov

Strukture in razrede, ki bi jih radi refleksirali, se registrira z uporabo makrojev, ki so lahko v isti datoteki s končnico *cpp* ali v katerikoli drugi datoteki s končnico *cpp*. Priporočljivo je, da se vse registracijske makroje za posamezno skupno knjižnico ali izvršljivo datoteko premakne v temu namenjeno datoteko, ki jo lahko obravnavamo kot refleksijski opisnik knjižnice oziroma reflesijski opisnik izvršljive datoteke.

V programski kodi 5.3 vidimo makroje, s katerimi registriramo razrede in njihove elemente. Pri registraciji razredov in struktur naštejemo vsa polja in metode, ki bi jih radi refleksirali. Če imamo več prekrivnih metod z istim imenom, moramo specificirati tudi tip vrnjene vrednosti in argumentov. Privzet konstruktor, konstruktor za kopiranje in konstruktor za premikanje se registrirajo avtomatično, če obstajajo. Ostale konstruktorje, s katerimi želimo omogočiti ustvarjanje objektov, moramo registrirati. Ni nam potrebno registrirati vseh prekrivnih metod z istim imenom, argumenti in različnimi kvalifikatorji, saj knjižnica sama poišče vse prekrivne metode z enakimi argumenti in registrira vse tiste, ki obstajajo.

```
1 #include "Metadata.h"
2 #include "ExampleStruct.h"
3 #include "ExampleClass.h"
4 using namespace std;
5
6 REFLECT_TYPE_START(ExampleStruct)
7     FIELD(mainValue)
8     FIELD(values)
9 REFLECT_TYPE_END
```

```
10
11  REFLECT_TYPE_START(ExampleClass)
12      CONSTRUCTOR(ExampleClass, string, vector<int>)
13      METHOD(clear)
14      METHOD(setFactor)
15      METHOD(void, add, string, vector<int>)
16      METHOD(void, add, const ExampleStruct &)
17      METHOD(vector<float>, getValues, const char* key)
18      METHOD(testPerformance)
19  REFLECT_TYPE_END
```

Programska koda 5.3: Primer registracije elementov s knjižnico Jobjects++.

Sicer je mogoče imeti dvanajst prekrivnih metod z enakimi argumenti, pri čemer ne morejo vse obstajati istočasno. V programski kodi 5.4 vidimo vse možnosti prekrivnih metod z enakimi argumenti, ki jih standard C++ omogoča.

```
1   class Bar
2   {
3       void foo();
4       void foo() const;
5       void foo() volatile;
6       void foo() const volatile;
7
8       void fooRef() &;
9       void fooRef() const &;
10      void fooRef() volatile &;
11      void fooRef() const volatile &;
12
13      void fooRef() &&;
14      void fooRef() const &&;
15      void fooRef() volatile &&;
16      void fooRef() const volatile &&;
17  }
```

Programska koda 5.4: Prikaz vseh možnih kvalifikatorjev metod.

5.2.3 Registracija privatnih in zaščitenih elementov

Če bi želeli refleksirati tudi zasebna in zaščitena polja ter metode, bi morali v razred, ki ga refleksiramo, dodati makro, ki se razširi v prijateljsko povezavo z registracijsko funkcijo. Sicer ni priporočljivo zaobiti enkapsulacije preko refleksije, lahko pa nam pomaga pri izpisovanju notranjega stanja objektov za potrebe razhroščevanja. V programski kodi 5.5 vidimo, kako omogočiti registracijo zasebnih in zaščitenih elementov.

```
1 #include "Reflectable.h"
2
3 class ExampleClass : public Reflectable<ExampleClass> {
4     REFLECT_FULL_ACCESS;
5     ...
6 }
```

Programska koda 5.5: Omogočanje registracije privatnih in zaščitenih elementov.

5.2.4 Primer uporabe

V programski kodi 5.6 vidimo primer uporabe knjižnice Jobjects++ z ekvivalentno programsko kodo, kot smo jo videli v primeru uporabe knjižnice RTTR in refl-cpp.

```
1 #include "ClassRegistry.h"
2 #include "ExampleStruct.h"
3 using namespace std;
4
```

```
5  void example() {
6      try {
7          ClassRegistry registry("plugin_example");
8
9          for (Class clasz : registry.getClasses()) {
10             cout << clasz.getName() << endl;
11         }
12
13         ExampleStruct exampleStructObj;
14         exampleStructObj.mainValue = 1;
15         exampleStructObj.values["A"] = { 1, 2, 3 };
16
17         Class exampleClass =
18             registry.getClass("ExampleClass");
19
20         for (Field field : exampleStructObj.getClass().getFields()) {
21             cout << field.getName() << endl;
22         }
23         for (Method method : exampleClass.getMethods()) {
24             cout << method.getName() << endl;
25         }
26
27         exampleStructObj.getClass().getField(
28             "mainValue"
29         ).set(exampleStructObj, 2);
30
31         unique_ptr<Object> exampleClassObj =
32             exampleClass.newInstance(
33                 string("B"),
34                 vector<int>{4, 5, 6}
35             );
36
37         Method setFactorMethod = exampleClass.getMethod("setFactor");
38         setFactorMethod.invoke<void>(
39             *exampleClassObj,
40             2.0f
41         );
```

```
42
43          Method addMethod = exampleClass.getMethod("add");
44          addMethod.invoke<void>(
45              *exampleClassObj,
46              string("C"), vector<int>{7, 8, 9}
47          );
48          addMethod.invoke<void, const ExampleStruct&>(
49              *exampleClassObj,
50              exampleStructObj
51          );
52
53          Method getValuesMethod = exampleClass.getMethod("getValues");
54          vector<float> result =
55              getValuesMethod.invoke<vector<float>>(
56                  *exampleClassObj,
57                  "A"
58              );
59
60          cout <<
61              exampleClassObj->toString() <<
62              endl;
63      }
64      catch (const Exception &e) {
65          cout << e.Message() << endl;
66      }
67  }
```

Programska koda 5.6: Primer uporabe knjižnice Jobjects++.

Poleg tega na koncu izpišemo notranje stanje objekta *exampleClassObj* z uporabo metode *Object::toString*, ki bo izpisala naslednje:

```
{
    "m_factor" : 2.0,
    "m_struct" :
    {
```

```
        "mainValue" : 2,
        "values" :
        [
            {
                "key" : "A",
                "value" : [1, 2, 3]
            },
            {
                "key" : "B",
                "value" : [4, 5, 6]
            },
            {
                "key" : "C",
                "value" : [7, 8, 9]
            }
        ]
    }
}
```

Razlika med zgornjim primerom in primerom, ki smo ga videli pri uporabi knjižnice RTTR, je v tem, da nam pri uporabi knjižnice Jobjects++ ni potrebno eksplicitno preverjati napak, saj uporabljamo izjeme. Prav tako je preprečeno nedefinirano obnašanje, saj nam knjižnica vrže izjemo v primeru, ko specificiramo drugačen tip, kot ga metoda pričakuje. To velja za vhodne argumente kot tudi za vrnjeno vrednost.

Če bi metodo *setFactor* poklicali s parametrom *2.0* namesto *2.0f*, bi nam knjižnica vrgla izjemo s sporočilom *„Method ExampleClass::setFactor(double) not found.“*, saj obstaja samo metoda *ExampleClass::setFactor(float)*. Knjižnica bo torej ob vsaki nepravilni uporabi vrgla izjemo s sporočilom, ki ga uporabnik razume dovolj dobro, da lahko popravi programsko kodo.

5.3 Arhitektura knjižnice

5.3.1 Šablonski razred Reflectable

Jedro knjižnice je šablonski razred *Reflectable*, ki ga vsak refleksibilni razred mora dedovati. V programski kodi 5.7 vidimo šablonski razred *Reflectable*, ki implementira metodo *Object::getClass*. Metoda *Object::getClass* ob prvem klicu ustvari statičen objekt *CClass*, ki je parametriziran z razredom, ki ga refleksiramo. Polega tega vidimo, da statični objekt razreda *CClass* ovijemo še s statičnim objektom ovojnega razreda *Class*, z namenom poenostavljene uporabe, saj razred *Class* vsebuje metode, ki so lažje za uporabo kot metode razreda *CClass*, kar bomo videli v nadaljevanju.

```
1  template <typename ClassType>
2  class Reflectable : public Object {
3  protected:
4      typedef ClassType ReflectedClass;
5  public:
6      const Class &getClass() const {
7          static const CClass<ClassType> reflection;
8          static const Class clasz(reflection);
9          return clasz;
10     }
11
12     Json::Value serialize() const {
13         ...
14     }
15
16     const char* toString() const {
17         ...
18     }
19
20     void deserialize(Json::Value value) {
21         ...
22     }
23
```

```
24      void fromString(const char* val) {
25          ...
26      }
27  };
```

Programska koda 5.7: Primer uporabe knjižnice Jobjects++.

5.3.2 Šablonski razred CClass

Šablonski razred *CClass* je razred, ki hrani vse refleksijske informacije za posamezen razred, s katerim je parametriziran oziroma ga refleksira. V programski kodi 5.8 vidimo definicijo šablonskega razreda *CClass*, ki implementira vmesnik *IClass* in vsebuje sledeče:

- ime razreda v obliki niza
- seznam razredov, v katere lahko pretvarjamo
- seznam polj
- seznam metod
- seznam konstruktorjev

Vidimo tudi, da imajo vsa polja, metode in konstruktorji enoten vmesnik *IField*, *IMethod* in *IConstructor*, da vse ustvarjene objekte lahko vstavimo v standardne homogene vsebovalnike za kasnejšo uporabo.

```
1  template<typename Class>
2  class CClass : public IClass {
3  private:
4      std::string m_name;
5      std::map<std::string, ICast*> m_castMap;
6      std::map<std::string, IField*> m_fieldMap;
7      std::map<std::string, IMethod*> m_methodMap;
```

```
8       std::map<size_t, IConstructor*> m_constructorMap;
9       std::vector<ICast*> m_castVector;
10      std::vector<IField*> m_fieldVector;
11      std::vector<IMethod*> m_methodVector;
12      std::vector<IConstructor*> m_constructorVector;
13
14      typedef Class ReflectedClass;
15
16  public:
17      const char* getName() const;
18
19      void addCasts(std::vector<ICast *> casts);
20      const ICast&
21      getCast(const char* signature, const char *name) const;
22
23      void addField(IField& field);
24      const IField&
25      getField(const char *name) const;
26      IField* const*
27      getFields(size_t& nFields) const;
28
29      void addMethodInvoker(IMethodInvoker& method);
30      const IMethod&
31      getMethod(const char *name) const;
32      IMethod* const*
33      getMethods(size_t& nMethods) const;
34
35      void addConstructor(IConstructor& constructor);
36      const IConstructor&
37      getConstructor(
38          size_t argsSignature,
39          const char* argsName
40      ) const;
41      IConstructor* const*
42      getConstructors(size_t& nConstructors) const;
43
44      template <typename ReflectedClass>
```

```
45      void registerMetadata();
46      CClass();
47  };
```

Programska koda 5.8: Šablonski razred *CClass*.

5.3.3 Skupni vzorec šablonskih razredov, vmesnikov in ovojnih razredov

V programskih kodah 5.9 in 5.10 vidimo vzorec med šablonskimi razredi *CClass*, *CField*, *CMethodInvoker* in *CConstructor*, vmesniki *IClass*, *IField* in *IMethodInvoker* ter ovojnimi razredi *Class*, *Field* in *Method*.

```
1   template<typename Class>
2   class CClass : public IClass {
3   ...
4   }
5
6   template<typename Class, typename Type>
7   class CField : public IField {
8   ...
9   }
10
11  template <typename Class, typename Return, typename... Args>
12  class CMethodInvoker : public IMethodInvoker {
13  ...
14  }
15
16  template <typename Class, typename... Args>
17  class CConstructor : public IConstructor {
18  ...
19  }
```

Programska koda 5.9: Vzorec šablonskih razredov, ki implementirajo enotne vmesnike.

```
class Class {
private:
    const IClass& m_class;
...
};

class Field {
private:
    const IField& m_field;
...
};

class Method {
private:
    const IMethod& m_method;
...
}
```

Programska koda 5.10: Vzorec ovojnih razredov, ki vsebujejo vmesnike.

Šablonski razredi implementirajo dejansko funkcionalnost in so parametrizirani s tipi (tip razreda, tip polja, tip vhodnih argumentov in vrnjene vrednosti metode), ki jih refleksirajo. Vse instance šablonskih razredov implementirajo skupne vmesnike, ki jih nato ovijejo ovojni razredi.

5.3.4 Vmesniki

Programska koda 5.11 prikazuje vmesnike *IClass*, *IField*, *IMethodInvoker* in *IConstructor*. Vidimo, da vmesniki uporabljajo samo osnovne tipe in vmesnike, ki jih sami definiramo.

```
1  class IClass {
2  public:
3      virtual const char*
4      getName() const = 0;
5
6      virtual const ICast&
7      getCast(const char* signature, const char* name) const = 0;
8
9      virtual const IField&
10     getField(const char* name) const = 0;
11
12     virtual IField* const*
13     getFields(size_t& nFields) const = 0;
14
15     virtual const IMethod&
16     getMethod(const char* name) const = 0;
17
18     virtual IMethod* const*
19     getMethods(size_t& nMethods) const = 0;
20
21     virtual const IConstructor&
22     getConstructor(size_t argsSignature, const char* argsName) const =
       ↪  0;
23
24     virtual IConstructor* const*
25     getConstructors(size_t& nConstructors) const = 0;
26
27     virtual ~IClass() {
28     }
29 };
30
31 class IField
32 {
33 public:
34     virtual const char*
35     getName() const = 0;
36
```

```
37      virtual IAdaptor&
38      getValue(const Object& obj) const = 0;
39
40      virtual void
41      setValue(Object& obj, IAdaptor& value) const = 0;
42
43      ...
44
45      virtual ~IField() {
46      }
47  };
48
49  class IMethodInvoker {
50  public:
51      virtual const char*
52      getName() const = 0;
53      ...
54      virtual IAdaptor*
55      invoke(Object &obj, IAdaptor **args) const = 0;
56      ...
57      virtual Json::Value
58      invokeSerialized(Object& obj, Json::Value args) const = 0;
59      ...
60      virtual ~IMethodInvoker() {
61      }
62  };
63
64  class IConstructor {
65  public:
66      ...
67      virtual Object&
68      newInstance(IAdaptor** args) const = 0;
69
70      virtual ~IConstructor() {
71      }
72  };
```

Programska koda 5.11: Vmesniki.

Namenoma ne uporabimo tipov iz standardne knjižnice, kot so *std::string*, *std::vector* ipd. Izjema je tip *Json::Value*, ki je del knjižnice jsoncpp. Razlog je v tem, da bi radi imeli čim večjo stopnjo kompatibilnosti med prevedenimi knjižnicami (komponentami). Vmesnik med drugim tudi definira metode, ki se jih lahko pokliče preko meje med skupnimi knjižnicami oziroma mejo med izvršljivo datoteko in skupno knjižnico (angl. component boundary), zaradi česar je zelo pomembno, da se le-ta s časom ne spreminja, saj bi v primeru spremembe vse prevedene komponente morali prevesti na novo. Če bi v našem sistemu mešali knjižnice (komponente), ki se povezujejo z različnimi verzijami standardne knjižnice, bi lahko izzvali nedefinirano obnašanje, saj je velikost vrednosti nekaterih tipov (*std::string*) lahko različna v različnih verzijah standardne knjižnice. Če torej uporabljamo samo vrednosti tipov, za katere vemo, da bo velikost v pomnilniku ostala enaka v vseh verzijah prevedenih komponent, in privzamemo, da bomo v vseh prevedenih komponentah uporabljali isto verzijo knjižnice jsoncpp, smo s tem zagotovili visoko stopnjo kompatibilnosti med prevedenimi komponentami. To pomeni, da bomo lahko v našem manj sklopljenem sistemu mešali komponente, ki so prevedene z različnimi konfiguracijami prevajalnika C++ (konfiguracija za razhroščevanje se razlikuje od konfiguracije za dostavo) in morda tudi komponente, ki so prevedene z različnimi verzijami prevajalnikov C++ ali celo s prevajalniki C++ različnih proizvajalcev, saj je oblika virtualne tabele metod v pomnilniku enaka med različnimi proizvajalci prevajalnikov C++, čeprav standard C++ tega ne določa.

5.3.5 Ovojni razredi

Knjižnica definira tudi ovojne razrede (angl. wrapper classes), ki poenostavijo uporabo, saj je uporaba vmesnikov zaradi omenjenih razlogov lahko nerodna. V programski kodi 5.12 vidimo ovojne razrede, ki uporabljajo tipe iz standardne knjižnice, saj s tem poenostavijo uporabo.

```
class Class {
private:
    const IClass& m_class;

public:
    Field getField(const char *name) const;
    std::vector<Field> getFields() const;

    Method getMethod(const char *name) const;
    std::vector<Method> getMethods() const;

    template <typename... Args>
    std::unique_ptr<Object> newInstance(Args... args) const;
    ...
};

class Field {
private:
    const IField& m_field;

public:
    const char* getName() const;

    template<typename Type>
    Type get(const Object &obj) const;

    template<typename Type>
    void set(Object &obj, Type value) const;
    ...
};

class Method {
private:
    const IMethod& m_method;

public:
    const char* getName() const;
```

```
38
39      template <typename Return, typename... Args>
40      Return invoke(Object& obj, Args... args) const;
41
42      template <typename Return, typename... Args>
43      Return invoke(const Object& obj, Args... args) const;
44      ...
45  };
```

Programska koda 5.12: Ovojni razredi.

Programska koda 5.13 prikazuje enostavno metodo *Class::getField* ovojnega razreda *Class*, ki pokliče metodo ovitega vmesnika *IClass::getField*, ki vrne referenco tipa *const IField&*. Metoda *Class::getField* nato ustvari in vrne objekt ovojnega razreda *Field*, s katerim ovije vrnjeno referenco tipa *const IField&*, saj konstruktor razreda *Field* kot argument sprejme referenco tipa *const IField&* in se zato implicitno ustvari.

```
1  Field Class::getField(const char *name) const {
2      return m_class.getField(name);
3  }
```

Programska koda 5.13: Ovojna metoda *Class::getField*.

Naloga ovojnih razredov je tudi ta, da se pri uporabi prenašajo kot vrednosti (ne kot reference ali kazalci), saj jih na ta način enostavneje dodajamo v standardne vsebovalnike. Poleg tega je naloga ovojnih razredov, da pri pridobivanju ali spreminjanju razrednih polj in klicanju razrednih metod pretvarjajo iz tipov, ki jih poda uporabnik, v adapterje, ki jih sprejemajo enotni vmesniki.

5.3.6 Adapterji

Šablonski razred *CAdaptor* sledi vzorcu adapterja in je ovojni razred kateregakoli tipa. Njegov namen je, da imamo enoten vmesnik za množico vseh različnih tipov, ki jih uporabljamo pri pridobivanju in nastavljanju vrednosti razrednih polj, klicanju razrednih metod in ustvarjanju objektov. Ker poleg vrednosti, ki jo ovijajo, vsebujejo še ime in ključ tipa, lahko v času izvajanja primerjamo podane tipe s pričakovanimi tipi. V programski kodi 5.14 vidimo vmesnik *IAdaptor* in šablonski razred *CAdaptor*, ki implementira vmesnik *IAdaptor* in ovija vrednost tipa, s katerim je šablonski razred parametriziran.

```
1  class IAdaptor {
2  public:
3      ...
4      virtual const char* getSignature() const = 0;
5      virtual const char* getName() const = 0;
6      virtual ~IAdaptor() {
7      }
8  };
9
10 template <typename Type>
11 class CAdaptor : public IAdaptor {
12 private:
13     Type m_value;
14
15 public:
16     ...
17     const Type &getValue() {
18         return m_value;
19     }
20     const char* getSignature() const;
21     const char* getName() const;
22     ...
23 };
```

Programska koda 5.14: Šablonski razred *CAdaptor*, ki implementira vmesnik *IAdaptor*.

Kot vidimo, lahko ovito vrednost dobimo, če poznamo tip vrednosti. To lahko naredimo po tem, ko varno pretvorimo kazalec (ali referenco) tipa *IAdaptor* v kazalec (ali referenco) tipa *CAdaptor*. Varno pretvorbo lahko izvedemo takrat, ko smo prepričani, da adapter ovija vrednost tipa, ki ga pričakujemo. Prepričani smo lahko po uspešni primerjavi ključa ovitega tipa s ključem tipa, ki ga pričakujemo.

5.3.7 Implementacija klicanja metod

V programski kodi 5.15 vidimo implementacijo metode *CMethodInvoker::invoke* in način pretvarjanja iz vmesnika *IAdaptor* v šablonski razred *CAdaptor*, iz katerega dobimo ovito vrednost. Metoda *CMethodInvoker::invoke* pričakuje objekt z vmesnikom *Object*, na katerem se metoda izvede, in argumente v obliki *IAdaptor***. V njej lahko vidimo pretvarjanje iz vmesnikov *IAdaptor* v šablonske razrede *CAdaptor*, preko katerih dobimo ovite vrednosti podanih argumentov. Prav tako vidimo pretvarjanje iz vmesnika *Object* v dejanski razred, iz katerega je podan objekt, da lahko na njem pokličemo metodo z metodnim kazalcem, ki smo ga shranili ob registraciji. Metoda *CMethodInvoker::invoke* vsebuje informacijo o tem, na kateri razred se metoda nanaša, katerih tipov argumente pričakuje in katerega tipa vrednost vrne, saj je njen razred *CMethodInvoker* parametriziran s tipom razreda, na katerega se metoda nanaša, s tipi argumentov, ki jih metoda sprejme, in tipom vrnjene vrednosti, ki jo metoda vrne.

```
1 template <typename Class, typename Return, typename... Args>
2 class CMethodInvoker : public IMethodInvoker {
3 private:
4     Return(Class::* m_method)(Args...);
5     ...
6 public:
```

```
7       template<typename Method, std::size_t... Index>
8       static IAdaptor* Invoke(
9           Method method,
10          Object& object,
11          IAdaptor** args,
12          std::index_sequence<Index...>
13      ) {
14          static thread_local std::byte =
15              retValBuffer[sizeof(CAdaptor<Return>)];
16          if constexpr (std::is_same<Return, void>()) {
17              (static_cast<Class&>(
18                  static_cast<Reflectable<Class>&>(
19                      object
20                  )).*method)(
21                      static_cast<CAdaptor<Args>&>(
22                          *args[Index]
23                      ).getValue()...
24                  );
25              return new(retValBuffer) CAdaptor<void>();
26          }
27          else {
28              return new(retValBuffer) CAdaptor<Return>(
29                  (static_cast<Class&>(
30                      static_cast<Reflectable<Class>&>(object)
31                  ).*method)(
32                      static_cast<CAdaptor<Args>&>(
33                          *args[Index]
34                      ).getValue()...
35                  )
36              );
37          }
38      }
39      IAdaptor* invoke(
40          Object& object,
41          IAdaptor **args
42      ) const {
43          return CMethodInvoker<
```

```
            Class,
            Return,
            Args...
        >::Invoke(
            m_method,
            object,
            args,
            std::index_sequence_for<Args...>{}
        );
    }
    ...
}
```

Programska koda 5.15: Implementacija metode *invoke*.

V programski kodi 5.16 vidimo še implementacijo metode *invoke* v ovojnem razredu *Method*. Ta metoda iz podanih argumentov, ki so lahko kateregakoli tipa, s pomočjo šablonske metode *BuildAdaptorArrayFromArgs* sestavi ovojne razrede *CAdaptor*. Pri tem smo pazljivi, da ne uporabimo kopice, in tako spomin za vse adapterje rezerviramo na skladu, kljub temu da vsebovalnik adaptorjev, ki jih metoda *IMethodInvoker::invoke* pričakuje, zaradi uporabe polimorfizma vsebuje kazalce.

```
class Method {
private:
    const IMethod& m_method;
    ...

    template <size_t... Index, typename... Adaptors>
    static std::array<IAdaptor*, sizeof...(Adaptors)>
    BuildAdaptorArrayFromArgs(
        std::index_sequence<Index...>,
        Adaptors&&... adaptors
    ) {
        if constexpr (sizeof...(Adaptors) > 0) {
```

```
            std::array<
                IAdaptor*, sizeof...(Adaptors)
            > result = { &adaptors... };
            return result;
        }
        else {
            std::array<IAdaptor*, 0> result = { 0 };
            return result;
        }
    }

public:

    template <typename Return, typename... Args>
    Return invoke(Object& obj, Args... args) const {
        static const size_t argsSignature =
            GetArgsSignature<Args...>(
                std::index_sequence_for<Args...>{}
            );
        static const std::string argsName =
            GetArgsName<Args...>();

        const IMethodInvoker& methodInvoker =
            m_method.getMethod(
                argsSignature,
                argsName.c_str() + 1,
                LValueRef
            );

        IAdaptor* retVal = methodInvoker.invoke(
            obj,
            BuildAdaptorArrayFromArgs(
                std::index_sequence_for<Args...>{},
                CAdaptor<Args>(args)...
            ).data()
        );
        if constexpr (!std::is_same<Return, void>()) {
```

```
50              return(static_cast<CAdaptor<Return> &>(
51                  *retVal
52              ).getValue());
53          }
54      }
55      ...
56  }
```

Programska koda 5.16: Implementacija metode *invoke* v ovojnem razredu.

5.3.8 Implementacija registracije elementov

Registracija elementov poskrbi, da imamo v času izvajanja na voljo objekte vseh instanc šablonskih razredov *CClass* za vse razrede, ki jih registriramo. Ustvarjeni objekti šablonskih razredov *CClass* vsebujejo vse refleksijske informacije za razrede, ki jih refleksirajo. V času prevajanja bo prevajalnik iz registracijske programske kode ustvaril instance šablonskih razredov *CClass*, *CField*, *CMethodInvoker* in *CConstructor* za tipe elementov, ki jih refleksiramo. V času izvajanja se bodo pri klicu konstruktorja šablonskega razreda *CClass* za posamezen razred ustvarili objekti instanciranih šablonskih razredov. Objekti, ki predstavljajo posamezne elemente razreda, se bodo nato vstavili v standardne vsebovalnike, ki so del objekta šablonskega razreda *CClass*, od koder jih v času izvajanja pridobivamo po potrebi.

Programska koda 5.17 prikazuje razširjene makroje za registracijo strukture *ExampleStruct*. Za strukturo, ki jo registriramo, se specializira konstruktor šablonskega razreda *CClass*. Ko se bo poklical konstruktor razreda *CClass*, se bo registracijska programska koda izvedla. Vidimo tudi, da se za vsak razred, ki ga registriramo, ustvari globalna funkcija, ki se začne z imenom *Factory* in nadaljuje z imenom razreda. Namen te globalne funkcije je, da nam vrne refleksijske informacije za razred, po katerem se globalna funkcija imenuje. Ker je globalna funkcija označena s ključno besedo *extern C*, jo prevajalnik v prevedeni prevajalni enoti označi s simbolom po stan-

dardu C, kar pomeni, da bo ime simbola enako imenu funkcije. Ko knjižnico dinamično naložimo v procesni prostor, lahko iz imena razreda, za katerega želimo dobiti refleksijske informacije, dobimo kazalec na globalno funkcijo, ki nam bo vrnila vmesnik *IClass* za šablonski razred *CClass*, ki je parametriziran z razredom, ki ga refleksiramo. Na platformi Windows za ta namen uporabimo API *GetProcAddress*, na platformi Linux pa API *dlsym*. Kazalec na globalno funkcijo za določen razred lahko dobimo tudi brez dinamičnega nalaganja knjižnic oziroma ga lahko dobimo za razrede, ki so del izvršljive datoteke, iz katere je proces pognan.

```
template<>
CClass<ExampleStruct>::CClass() : m_name("ExampleStruct") {
    registerMetadata<ExampleStruct>();
}
extern "C" EXPORT_API const IClass& Factory_ExampleStruct() {
    static const CClass<ExampleStruct> clasz;
    return clasz;
}
template<>
template <typename ReflectedClass>
void CClass<ExampleStruct>::registerMetadata()
{
    addCasts(newCasts<ExampleStruct, ExampleStruct>());

    if constexpr (std::is_default_constructible<
        ExampleStruct
    >::value) {
        addConstructor(
            newConstructor<ExampleStruct>()
        );
    }
    if constexpr (std::is_copy_constructible<
        Class
    >::value) {
        addConstructor(
```

```
26              newConstructor<
27                  ExampleStruct,
28                  const ExampleStruct &
29              >()
30          );
31      }
32      if constexpr (std::is_move_constructible<
33          Class
34      >::value) {
35          addConstructor(
36              newConstructor<
37                  ExampleStruct,
38                  ExampleStruct &&
39              >()
40          );
41      }
42
43      addField(newField<ReflectedClass>(
44          "mainValue",
45          &ReflectedClass::mainValue
46      ));
47      addField(newField<ReflectedClass>(
48          "valueMap",
49          &ReflectedClass::valueMap
50      ));
51  }
```

Programska koda 5.17: Razširitev registracijskih makrojev knjižnice Jobjects++.

Poleg tega lahko za določeno knjižnico ali izvršljivo datoteko naštejemo vse globalne funkcije, ki se začnejo z imenom *Factory*, s čimer dobimo seznam vseh refleksibilnih razredov, ki jih določena knjižnica oziroma izvršljiva datoteka implementira oziroma registrira. Na platformi Windows za naštevanje globalnih funkcij uporabimo API iz knjižnice dbghelp, na platformi Linux pa

za naštevanje simbolov uporabimo API iz knjižnice BFD.

S takšnim pristopom nimamo globalnih objektov, ki vstavljajo refleksijske informacije v globalno stanje vseh refleksijskih informacij, ko se program požene, kot je to v primeru knjižnice RTTR. Namesto tega imamo za vsak refleksibilen razred pripadajočo globalno funkcijo, do katere lahko dostopamo tako, da v izvršljivi datoteki ali skupni knjižnici poiščemo njen naslov glede na ime razreda. Lahko smo prepričani, da se bo registracijska programska koda za razred, za katerega želimo pridobiti refleksijske informacije, izvedla, ko se bo poklicala njegova globalna funkcija. Prepričani smo lahko zato, ker globalna funkcija pri prvem klicu ustvari statični objekt razreda *CClass*, ki v svojem konstruktorju pokliče metodo *registerMetadata*. Na podoben način nam refleksijske informacije vrača tudi metoda *Reflectable::getClass()*, kjer smo lahko prepričani, da se bo registracijska koda izvedla ob prvem klicu metode *Reflectable::getClass()*. Na ta način nimamo enega globalnega stanja, kjer hranimo vse refleksijske informacije, kot je to v primeru knjižnice RTTR, ampak to stanje lokaliziramo za vsak posamezen razred v globalnih funkcijah, kjer je vsaka funkcija namenjena razredu, ki ga refleksira, in v metodi *Reflectable::getClass()* za vsak posamezen razred.

5.4 Serializacija in deserializacija

Knjižnica Jobjects++ zna serializirati vse osnovne tipe in vsebovalnike *std::vector*, *std::list* in *std::map*, ki so del standardne knjižnice. Prav tako je omogočeno poljubno sestavljanje vsebovalnikov, tako kot to vidimo v primeru strukture *ExampleStruct*. Če bi hoteli serializirati določen tip ali vsebovalnik, ki ga knjižnica ne pozna in ni refleksibilen, lahko specializiramo šablonski razred, ki je namenjen serializaciji za posamezen tip ali vsebovalnik. Za dejansko serializacijo in deserializacijo sicer uporabljamo knjižnico jsoncpp.

V programski kodi 5.18 vidimo specializacijo serializacije za strukturo *CustomType* in vsebovalnik *CustomContainer*, ki ju knjižnica ne pozna. Ker

knjižnica teh tipov ne pozna in ker tipi niso refleksibilni, moramo specializirati šablonski razred *Serialization*, če želimo, da bo knjižnica znala serializirati in deserializirati vrednosti teh tipov. Pri vsebovalniku *CustomContainer* uporabimo delno specializacijo, saj je le-ta parametriziran s tipom, ki ga vsebuje, pri strukturi *CustomType* pa uporabimo eksplicitno specializacijo. S knjižnico bomo tako lahko serializirali in deserializirali tudi strukturo *CustomType* in vsebovalnik *CustomContainer*, pri čemer bomo lahko poljubno mešali strukturo *CustomType* z obstoječimi vsebovalniki in obstoječe tipe z vsebovalnikom *CustomContainer*. Prav tako lahko poljubno sestavljamo vsebovalnike na način, kot vidimo v primeru strukture *ExampleStruct*.

```
#include "Serialization.h"
using namespace std;

struct CustomType {
    string strValue;
    float floatValue;
    int intValue;
};

template <typename T>
struct CustomContainer : public vector<T>
{
    ...
};

template <>
class Serialization<CustomType> {
public:
    static Json::Value Serialize(CustomType customType) {
        Json::Value::ArrayIndex index = 0;
        Json::Value ret;
        ret["strValue"] =
            Serialization<string>::Serialize(
                customType.strValue
```

```
25              );
26          ret["floatValue"] =
27              Serialization<float>::Serialize(
28                  customType.floatValue
29              );
30          ret["intValue"] =
31              Serialization<int>::Serialize(
32                  customType.intValue
33              );
34          return ret;
35      }
36
37      static CustomType Deserialize(Json::Value val) {
38          size_t index = 0;
39          CustomType ret;
40          ret.strValue =
41              Serialization<string>::Deserialize(
42                  val["strValue"]
43              );
44          ret.floatValue =
45              Serialization<float>::Deserialize(
46                  val["floatValue"]
47              );
48          ret.intValue =
49              Serialization<int>::Deserialize(
50                  val["intValue"]
51              );
52          return ret;
53      }
54  };
55
56  template <typename Type>
57  class Serialization<CustomContainer<Type>> {
58  public:
59      static Json::Value
60      Serialize(CustomContainer<Type> customContainer) {
61          Json::Value::ArrayIndex index = 0;
```

```
62          Json::Value ret;
63          for (Type type : customContainer) {
64              ret.insert(
65                  index++,
66                  Serialization<Type>::Serialize(type)
67              );
68          }
69          return ret;
70      }
71
72      static CustomContainer<Type>
73      Deserialize(Json::Value val) {
74          size_t index = 0;
75          CustomContainer<Type> ret;
76          for (auto it = val.begin(); it != val.end(); it++) {
77              ret.push_back(
78                  Serialization<Type>::Deserialize(*it)
79              );
80          }
81          return ret;
82      }
83  };
```

Programska koda 5.18: Specializacija serializacije in deserializacije za neznane tipe in vsebovalnike.

Knjižnica pri serializaciji kazalcev in referenc trenutno vrže izjemo. Za to smo se odločili, ker serializacija kazalcev in referenc odpira veliko problemov, ki bi jih morali rešiti. Pri uporabi referenc in kazalcev so lahko objekti, na katere kažejo le-ti, polimorfični, torej so lahko drugačnega tipa od deklariranega. Prav tako se pri deserializaciji odpira vprašanje, na kakšen način je spomin za objekte rezerviran v pomnilniku. Ali referenca oziroma kazalec kaže na vrednost znotraj objekta, pri čemer je spomin za objekt rezerviran na skladu, ali kaže na del pomnilnika v kopici? Vse te informacije bi morali pri serializaciji prav tako shraniti in jih upoštevati pri deserializaciji, kar precej

poveča kompleksnost knjižnice.

Sicer bi serializacijo in deserializacijo v večini primerov uporabili za prenos podatkov iz enega procesa v drugega, ali pa da podatke shranimo za kasnejšo uporabo. V takšnem primeru lahko vedno uporabimo strukture in razrede, ki vsebujejo vrednosti brez kazalcev in referenc.

5.5 Netipizirana refleksija

Pri uporabi refleksijskih knjižnic smo videli, da je pri pridobivanju in spreminjanju vrednosti polj ter klicanju metod v določenih primerih potrebno specificirati tipe. V primerih, ko uporabljamo konstante, nam prevajalnik pri tem lahko pomaga, sicer pa se ne moremo zanašati na to, da bo prevajalnik posredoval pravi tip metodi *invoke*, ki sprejme katerikoli tip. Prevajalnik namreč ne more vedeti, ali nameravamo metodi *invoke* podati vrednost ali referenco, če metoda *invoke* sprejme katerikoli tip. V teh primerih je bolje, če eksplicitno specificiramo tip, ki ga metoda pričakuje.

Ker je serializacijski mehanizem v knjižnici Jobjects++ tesno povezan z implementacijo refleksije, nam knjižnica pri tem ponuja možnost, da vse argumente pri nastavljanju polj in klicanju metod specificiramo v obliki nizov. Tako knjižnica ne bo pretvarjala iz vrednosti podanih tipov v vrednosti pričakovanih tipov, ampak bo deserializirala iz podanih nizov v vrednosti pričakovanih tipov z uporabo deserializacije za tipe, ki jih metoda pričakuje. S tem lahko dosežemo, da nam pri pridobivanju ali spreminjanju vrednosti polj in klicanju metod ni potrebno poznati tipov, ampak mora samo obstajati dogovor, kako se vrednost nekega tipa predstavi v obliki niza. V času prevajanja zato ni treba specificirati tipov vrednosti, ki jih posredujemo metodi *invoke*. S tem gremo pri refleksiji v času izvajanja še korak dlje, saj se prekrivna metoda izbira glede na število in vsebino nizov v času izvajanja in ne v trenutku prevajanja, kot je to pri knjižnici RTTR.

V programski kodi 5.19 vidimo implementacijo metode *invoke*, ki namesto pretvarjanja iz podanih tipov uporabi deserializacijo za tipe, ki jih

metoda pričakuje. Konstruktor šablonskega razreda *CAdaptor* vsebuje konstruktor, ki deserializira iz vrednosti tipa *Json::Value* v vrednost tipa, ki ga ovija šablonski razred *CAdaptor*. V ovojnem razredu *Method* vidimo metodo *invokeSerialized*, ki bo iz podanih nizov ustvarila objekte JSON in z njimi poskušala poklicati eno izmed prekrivnih metod z uporabo metode *CMethodInvoker::invokeSerialized*.

```
1  template <typename Type>
2  class CAdaptor : public IAdaptor {
3  private:
4      Type m_value;
5
6  public:
7  ...
8      CAdaptor(Json::Value value) :
9          m_value(Serialization<Type>::Deserialize(value)) {
10     }
11 ...
12 }
13
14 template <typename Class, typename Return, typename... Args>
15 class CMethodInvoker : public IMethodInvoker {
16 private:
17     Return(Class::* m_method)(Args...);
18
19 public:
20     template<typename Method, std::size_t... Index>
21     static Json::Value Invoke(
22         Method method,
23         Object& object,
24         Json::Value args,
25         std::index_sequence<Index...>
26     ) {
27         if constexpr (std::is_same<Return, void>()) {
28             (static_cast<Class&>(
29                 static_cast<Reflectable<Class>&>(object)
```

```
            ).*method)(
                CAdaptor<Args>(
                    args[(int)Index]
                ).getValue()...
            );
            return CAdaptor<void>().marshall();
        }
        else {
            return CAdaptor<Return>(
                (static_cast<Class&>(
                    static_cast<Reflectable<Class>&>(object)
                ).*method)(
                    CAdaptor<Args>(
                        args[(int)Index]
                    ).getValue()...
                )
            ).marshall();
        }
    }
    Json::Value invokeSerialized(
        Object& object,
        Json::Value args
    ) const {
        return CMethodInvoker<
            Class,
            Return,
            Args...
        >::Invoke(
            m_method,
            object,
            args, std::index_sequence_for<Args...>{}
        );
    }
...
}

class Method {
```

```
67  private:
68      const IMethod& m_method;
69  public:
70      ...
71      std::string invokeSerialized(
72          Object& obj,
73          std::vector<std::string> args
74      ) const {
75          Json::Value jsonArgs = PackArgsToJson(args);
76          Json::Value jsonRetVal;
77          size_t nMethods = 0;
78          size_t iMethod = 0;
79          IMethodInvoker* const* methods =
80              m_method.getMethodsByNArgs(args.size(), nMethods);
81          for (iMethod = 0; iMethod < nMethods; iMethod++) {
82              try {
83                  jsonRetVal =
84                      methods[iMethod]->invokeSerialized(
85                          obj,
86                          jsonArgs
87                      );
88                  break;
89              }
90              catch (const ISerializationException&) {
91                  continue;
92              }
93          }
94          if (iMethod == nMethods) {
95              throw  MethodsWithNArgumentsNotMatchingInputArguments(
96                  m_method.getName(),
97                  args.size()
98              );
99          }
100         return UnpackRetvalFromJson(jsonRetVal);
101     }
102     ...
```

```
103 }
```

Programska koda 5.19: Implementacija netipizirane refleksije.

Ker podani nizi ne vsebujejo informacij o tipih, ne moremo točno vedeti, katero metodo, ki ima isto ime in enako število argumentov, lahko pokličemo. Zaradi tega smo izbrali pristop, kjer poskušamo poklicati vse metode z enakim imenom in številom argumentov ter odnehamo takrat, ko uspešno pokličemo katerokoli od njih oziroma uspešno deserializiramo vse argumente v vrednosti tipov, ki jih metoda pričakuje. Če bi imeli metodo z istim imenom in številom argumentov, pri čemer bi ena metoda sprejela nize, druga pa števila, bi to lahko povzročilo, da bi kljub podajanju števila v obliki niza poklicali tisto metodo, ki pričakuje nize. Pri uporabi netipizirane refleksije v kombinaciji s prekrivnimi metodami, ki imajo enako ime in enako število argumentov, moramo zato biti previdni, da ne povzročimo dvoumnosti oziroma se metodam z enakim imenom in številom argumentov izogibamo v primeru, če bi z vrednostmi nizov lahko povzročili semantične dvoumnosti, ki bi povzročile nepravilno obnašanje.

V programski kodi 5.20 vidimo uporabo netipizirane refleksije. Ustvarili smo objekt razreda *ExampleClass* in na objektu poklicali metodo *void ExampleClass::setFactor(float)* in metodo *void ExampleClass::add(std::string, std::vector<int>)*, pri čemer so argumenti podani v obliki nizov.

```
1  void example() {
2      try {
3          ClassRegistry registry;
4          Class exampleClass =
5              registry.getClass("ExampleClass");
6          unique_ptr<Object> exampleClassObj =
7              exampleClass.newInstance();
8          string retVal;
9
10         Method setFactorMethod =
```

```
11              exampleClass.getMethod("setFactor");
12          retVal = setFactorMethod.invokeSerialized(
13              *exampleClassObj,
14              { "3.14159" }
15          );
16
17          Method addMethod = exampleClass.getMethod("add");
18          retVal = addMethod.invokeSerialized(
19              *exampleClassObj,
20              { "D", "[10, 11, 12]" }
21          );
22
23          cout <<
24              exampleClassObj->toString() <<
25              endl;
26      }
27      catch (const Exception & e) {
28          cout << e.Message() << endl;
29      }
30  }
```

Programska koda 5.20: Primer uporabe netipizirane refleksije.

Vrnjena vrednost, ki jo dobimo v obeh primerih, je prazen niz, saj klicani metodi nič ne vračata. Nizi morajo sicer biti podani v formatu JSON, zato pri speficiranju vrednosti tipa *std::vector<int>* uporabimo oglate oklepaje. Metoda *toString* bi v tem primeru vrnila naslednje:

```
{
        "m_factor" : 3.1415901184082031,
        "m_struct" :
        {
                "mainValue" : 0,
                "values" :
                [
```

```
                        {
                                "key" : "D",
                                "value" : [10, 11, 12]
                        }
                ]
        }
}
```

Z uporabo netipizirane refleksije lahko pokličemo samo tiste metode, ki v prototipu ne vsebujejo kazalcev in referenc. Razlog za to omejitev je isti kot pri serializaciji in deserializaciji razredov, ki vsebujejo kazalce in reference.

5.6 Generični razčlenjevalnik ukazne vrstice

Za diplomsko delo smo z uporabo knjižnice Jobjects++ razvili enostaven generični razčlenjevalnik ukazne vrstice. Programska koda 5.21 prikazuje strukturo, ki bi jo radi razčlenili, in uporabo razčlenjevalnika. Struktura *ParseStruct* vsebuje polja, ki ponazarjajo opcije, ki bi jih radi uporabili v ukazni vrstici. Opcijo *new*, ki jo želimo uporabiti kot stikalo, definiramo s tipom *bool*. Opcijo *name* definiramo s tipom *std::string*, saj želimo, da vsebuje ime v obliki niza. Opcijo *address* definiramo s tipom *std::optional<std::string>*, saj želimo, da vsebuje naslov v obliki niza, pri čemer pa definiramo tudi to, da opcija ni obvezna. Opcijo *age* definiramo s tipom *int*, saj želimo, da se starost zapiše s celim številom. Poleg polj imamo tudi metode, ki se začnejo z besedo *help* in nadaljujejo z imenom opcije. Namen teh metod je, da se pri nepravilni uporabi izpiše pomoč za posamezno polje. V primeru, da jih izpustimo, se pri posamezni opciji pomoč ne bo izpisala.

```
1 struct ParseStruct : public Reflectable<ParseStruct> {
2     bool new = false;
3     string name;
4     optional<string> address;
```

```
5       int age;
6
7       string help_new() {
8           return "Is entry new?";
9       }
10
11      string help_name() {
12          return "Name of the entry";
13      }
14
15      string help_address() {
16          return "Address of the entry";
17      }
18
19      string help_age() {
20          return "Age of the entry";
21      }
22  };
23
24  int main(int argc, char** argv) {
25      try {
26          ClassRegistry parserRegistry("CLIParser");
27          Class parserClass = parserRegistry.getClass("Parser");
28          unique_ptr<Object> obj =
29              parserClass.newInstance();
30          IParser& parser = parserClass.upcast<IParser>(*obj);
31          ParseStruct parseStruct;
32          try {
33              parser.parse(argc, argv, parseStruct);
34          }
35          catch (const IParseException & e) {
36              cout << "Exception occurred: " << e.Message() << endl;
37              parser.printUsage(parseStruct);
38              return 1;
39          }
40          cout << parseStruct.toString() << endl;
41          return 0;
```

```
42      }
43      catch (const Exception & e) {
44          cout << "Exception occurred: " << e.Message() << endl";
45          return 1;
46      }
47  }
```

Programska koda 5.21: Primer uporabe generičnega razčlenjevalnika.

V funkciji main vidimo uporabo generičnega razčlenjevalnika ukazne vrstice. Najprej v procesni prostor naložimo knjižnico *CLIParser.dll* (*CLIParser.so* na platformi Linux) in iz nje ustvarimo objekt razreda *Parser*. Razred *Parser* implementira vmesnik *IParser*, ki ga dobimo z uporabo metode *upcast*, ki z uporabo refleksije ugotovi, ali razred *Parser* implementira vmesnik *IParser* in ga vrne klicatelju. Z uporabo vmesnika *IParser* pokličemo metodo *IParser::parse*, ki ji posredujemo ukazno vrstico, ki smo jo specificirali pri zagonu programa (*argc*, *argv*), in strukturo ukazne vrstice *ParseStruct*. V primeru napake uporabimo metodo *printUsage*, ki ji posredujemo strukturo ukazne vrstice *ParseStruct*.

Sledi izpis programa glede na uporabljeno ukazno vrstico:

- `People.exe -new -name Janez -address "Novakova 1"`

  ```
  Exception occurred: Option age not specified
  Usage:
      [-new]                          Is entry new?
      -name <name>                    Name of the entry
      [-address <address>]            Address of the entry
      -age <age>                      Age of the entry
  ```

- `People.exe -new -name Janez -address "Novakova 1" -age 33`

  ```
  {
  ```

```
    "address" : "Novakova 1",
    "age" : 33,
    "new" : true,
    "name" : "Janez"
}
```

Če želimo dodati ali spremeniti določeno opcijo v ukazni vrstici programa, moramo spremeniti strukturo *ParseStruct*. Knjižnice *CLIParser* nam pri takšni spremembi ni potrebno ponovno prevajati, niti ne potrebujemo izvorne programske kode. Največja prednost tovrstne uporabe je ta, da se uporabniku ni treba ukvarjati s transformacijami iz ukazne vrstice v končno vrednost tipa, ki je naveden v strukturi, saj to transformacijo namesto njega opravlja generični razčlenjevalnik z uporabo netipizirane refleksije. Uporabnik lahko z manj programske kode bolje izrazi svoj namen.

5.7 Generični strežnik REST

Za diplomsko delo smo z uporabo knjižnice Jobjects++ razvili tudi enostaven generični strežnik REST. Programska koda 5.22 prikazuje razred *MyRestServer*, ki vsebuje enostavno funkcionalnost, ki jo ponujamo z uporabo generičnega strežnika REST. Imamo strukturo *FilesystemItem*, ki ponazarja element na datotečnem sistemu, in razred *MyRestServer*, ki ima dve metodi. Metoda *GET_listDir* vrne vse elemente v trenutnem imeniku datotečnega sistema, metoda *POST_changeDir* pa spremeni trenutni imenik v datotečnem sistemu. Strežniku REST moramo specificirati, pri katerem glagolu HTTP se posamezna metoda lahko izvede. Pri podobnem orodju JAX-RS v programskem jeziku Java bi za ta namen uporabili anotacijo. Ker v programskem jeziku C++ nimamo anotacij, smo se odločili, da glagol HTTP enkodiramo v ime metode.

```
1 struct FilesystemItem : public Reflectable<FilesystemItem> {
2     string fileName;
```

```
    string type;
    long long size;
};

class MyRestServer : public Reflectable<MyRestServer> {
private:
    string currentDirectory;

public:
    MyRestServer();
    void POST_changeDir(string newDirectory);
    vector<FilesystemItem> GET_listDir() const;
};

int main() {
    try {
        MyRestServer myRestServer;
        ClassRegistry restServerRegistry("RESTServer");
        Class restControllerClass =
            restServerRegistry.getClass("RESTController");
        unique_ptr<Object> obj =
            restControllerClass.newInstance<wstring, Object&>(
                L"http://localhost:6502/v1/reflection/api",
                myRestServer
            );

        IRESTController& restController =
            restControllerClass.upcast<IRESTController>(*obj);
        signal(SIGINT, handleUserInterrupt);

        restController.start();
        waitUntilUserInterrupt();
        restController.shutdown();
        return 0;
    }
    catch (const Exception & e) {
        printf("Exception occurred: %s\n", e.Message());
```

```
40          return 1;
41      }
42  }
```

Programska koda 5.22: Primer uporabe generičnega strežnika REST.

V funkciji main v procesni prostor naložimo knjižnico *RESTServer.dll* (*RESTServer.so* na platformi Linux), ustvarimo objekt razreda *RESTController*, ki vsebuje implementacijo generičnega strežnika REST, in z uporabo definiranega vmesnika *IRESTController* poženemo strežnik REST, ki mu posredujemo spletni naslov in objekt razreda *MyRestServer*. Če program poženemo, lahko z odjemalcem REST curl.exe v konzoli naredimo slednje:

```
C:\Users\user>curl.exe -d "E:\git\reflection" -k -i -X POST -H
"Content-Type:application/json"
http://localhost:6502/v1/reflection/api/changeDir
HTTP/1.1 200 OK
Server: Microsoft-HTTPAPI/2.0
Content-Length: 2
Content-Type: application/json
Date: Wed, 26 Aug 2020 13:50:54 GMT

""
C:\Users\user>curl.exe -k -i -X GET -H
"Content-Type:application/json"
http://localhost:6502/v1/reflection/api/listDir
HTTP/1.1 200 OK
Server: Microsoft-HTTPAPI/2.0
Content-Length: 760
Content-Type: application/json
Date: Wed, 26 Aug 2020 13:50:57 GMT
```

```
[{"fileName":".","size":0,"type":"DIRECTORY"},
{"fileName":"..","size":0,"type":"DIRECTORY"},
{"fileName":".git","size":0,"type":"DIRECTORY"},
{"fileName":".vs","size":0,"type":"DIRECTORY"},
{"fileName":"Client","size":0,"type":"DIRECTORY"},
{"fileName":"codesamples","size":0,"type":"DIRECTORY"},
{"fileName":"components","size":0,"type":"DIRECTORY"},
{"fileName":"libcpprest","size":0,"type":"DIRECTORY"},
{"fileName":"libjobject","size":0,"type":"DIRECTORY"},
{"fileName":"libjson","size":0,"type":"DIRECTORY"},
{"fileName":"LICENSE","size":11558,"type":"FILE"},
{"fileName":"other","size":0,"type":"DIRECTORY"},
{"fileName":"prototype","size":0,"type":"DIRECTORY"},
{"fileName":"reflection.sln","size":5259,"type":"FILE"},
{"fileName":"x64","size":0,"type":"DIRECTORY"}]
C:\Users\user>
```

V izpisu v konzoli lahko vidimo, da smo z odjemalcem curl.exe uspešno poklicali metodi *POST_changeDir* in *GET_listDir*. Če bi želeli razširiti funkcionalnost, nam ni potrebno spreminjati implementacije generičnega strežnika REST. Za ta namen spremenimo razred *MyRestServer*, pri čemer nam ni potrebno skrbeti za transformacije iz vhodnih sporočil, ki so poslani strežniku REST, in izhodnih sporočil, ki jih strežnik REST pošlje nazaj. Knjižnica Jobjects++ z uporabo netipizirane refleksije pretvori v in iz vrednosti tipov, ki jih metode v razredu *MyRestServer* sprejmejo in vrnejo. Na takšen način bi lahko klicali programsko kodo C++ iz kateregakoli skriptnega ali programskega jezika, ki ima možnost uporabe odjemalca REST.

5.8 Primerjava hitrosti klicanja metod

Primerjali smo hitrost klicanja metode *ExampleClass::testPerformance* brez refleksije in z uporabo refleksije s knjižnico RTTR, refl-cpp ter Jobject++.

Metoda *ExampleClass::testPerformance* ima prazno telo oziroma vrne vrednost *0.0*. Vse teste smo pognali v istem okolju. Za obe knjižnici smo pri prevajanju uporabili enake optimizacijske opcije, in sicer smo vklopili vse optimizacijske opcije, ki pohitrijo hitrost izvajanja. Pri testiranju smo uporabili programsko kodo 5.23, 5.24 in 5.25.

```
rttr::method m = exampleClass.get_method("testPerformance");
chrono::steady_clock::time_point begin =
    chrono::steady_clock::now();

for (int i = 0; i < 10000000; i++) {
    auto ret = m.invoke_variadic(
        exampleClassObj,
        { string("test"), vector<int>{1, 2, 3} }
    );
    if (!ret.is_valid()) {
        cerr << "Failed to call testPerformance";
        return -1;
    }
    double retVal = ret.get_wrapped_value<double>();
}

chrono::steady_clock::time_point end =
    chrono::steady_clock::now();

cout <<
    "Time difference = " <<
    chrono::duration_cast<chrono::nanoseconds>(
        end - begin
    ).count() <<
    "[ns]" <<
    endl;
```

Programska koda 5.23: Test klicanja metode ExampleClass::testPerformance s knjižnico RTTR.

```
constexpr auto funcTestPerformance = find_one(
    reflect<ExampleClass>().members,
    [](auto m) {
        return m.name == "testPerformance";
    }
);

chrono::steady_clock::time_point begin =
    chrono::steady_clock::now();

for (int i = 0; i < 10000000; i++) {
    double ret = funcTestPerformance(
        exampleClassObj,
        string("test"), vector<int>{1, 2, 3}
    );
}

chrono::steady_clock::time_point end =
    chrono::steady_clock::now();

cout <<
    "Time difference = " <<
    chrono::duration_cast<chrono::nanoseconds> (
        end - begin
    ).count() <<
    "[ns]" <<
    endl;
```

Programska koda 5.24: Test klicanja metode ExampleClass::testPerformance s knjižnico refl-cpp.

```
Method m = clasz.getMethod("testPerformance");

chrono::steady_clock::time_point begin =
```

```
4       chrono::steady_clock::now();
5
6   for (int i = 0; i < 10000000; i++) {
7       double ret = m.invoke<
8           double,
9           string,
10          vector<int>
11      >(*exampleClassObj, "test", {1, 2, 3});
12  }
13
14  chrono::steady_clock::time_point end =
15      chrono::steady_clock::now();
16
17  cout <<
18      "Time difference = " << chrono::duration_cast<chrono::nanoseconds>
        ↪  (
19          end - begin
20      ).count() <<
21      "[ns]" <<
22      endl;
```

Programska koda 5.25: Test klicanja metode ExampleClass::testPerformance s knjižnico Jobjects++.

Knjižnica refl-cpp je po pričakovanjih pri klicanju metod hitrejša od obeh knjižnic za refleksijo v času izvajanja in le za malenkost počasnejša od klicanja metod brez refleksijske knjižnice. Sklepamo, da lahko razliko v hitrosti klicanja metode s knjižnico refl-cpp in brez knjižnice pripišemo dejstvu, da lahko prevajalnik bolje optimizira programsko kodo, ki pokliče metodo brez refleksijske knjižnice. Knjižnica Jobjects++ pri klicanju metod ponuja več funkcionalnosti od knjižnice RTTR, saj pri iskanju prekrivnih metod zna poiskati pravo metodo tudi glede na kvalifikator metode in kvalifikator objekta, na kateri je metoda klicana. Pri klicanju metod si zapomni imena tipov, s katerimi je bila metoda klicana, da jih v primeru napake izpiše. Kljub temu

knjižnica	min. čas (s)	maks. čas (s)	povprečni čas (s)
brez	0,64	0,65	0,64
RTTR	2,72	2,77	2,74
refl-cpp	0,67	0,70	0,68
Jobjects++	2,63	2,68	2,66

Tabela 5.1: Rezultati testiranja hitrosti klicanja metode ExampleClass::testPerformance.

je knjižnica Jobjects++ pri klicanju metod hitrejša od knjižnice RTTR, kakor vidimo v tabeli 7.1. Hitrost klicanja metod je dober pokazatelj hitrosti knjižnice, saj se mora v tem procesu veliko stvari zgoditi. Ne moremo pa z gotovostjo trditi, da je knjižnica Jobjects++ vsesplošno hitrejša od knjižnice RTTR, saj bi za takšno trditev potrebovali bolj poglobljeno analizo.

Poglavje 6

Sklepne ugotovitve

V diplomskem delu smo pregledali področje refleksije v programskem jeziku C++. Pregledali smo uradni predlog, ki programskemu jeziku C++ prinaša zmožnost refleksije in nekatere obstoječe refleksijske knjižnice, ki se že uporabljajo. Preučili smo razlike med refleksijo v času prevajanja in refleksijo v času izvajanja. Uradni predlog, ki programskemu jeziku C++ prinaša refleksijo v času prevajanja in bo morda postal na voljo s standardom C++23, je vzpodbuden, saj prinaša osnovne gradnike, s katerimi bo mogoče razviti knjižnico za refleksijo v času prevajanja, knjižnico za refleksijo v času izvajanja ali pa specifično domensko knjižnico, kot je knjižnica za serializacijo in deserializacijo.

Pokazali smo, da je programski jezik C++ že danes zmožen refleksije določenih aspektov, saj nam pri uporabi refleksijskih knjižnic, ki se zanašajo na to, da uporabnik sam registrira razrede in vse elemente, ki jih želi refleksirati, ni potrebno specificirati vseh detajlov. Tako nam pri registraciji razrednih polj ni potrebno specificirati tipov polj. Prav tako nam pri registraciji razrednih metod, ki niso prekrivne, ni potrebno specificirati tipov vrnjene vrednosti in tipov argumentov.

Ker nismo bili povsem zadovoljni z določenimi aspekti trenutnih refleksijskih knjižnic, smo se odločili, da sami razvijemo novo refleksijsko knjižnico. Knjižnica Jobjects++, ki smo jo razvili, se od podobne knjižnice RTTR raz-

likuje v tem, da:

- ima bolj enostaven način registracije elementov;
- ima boljše upravljanje z napakami;
- definira osnovni razred, ki nam omogoči pisanje generalizirane programske kode za vse refleksibilne razrede z uporabo polimorfizma;
- omogoča pravilno klicanje prekrivnih metod, ki imajo enake argumente in različne kvalifikatorje;
- ima boljšo učinkovitost pri klicanju razrednih metod.

S primerom generičnega razčlenjevalnika ukazne vrstice in generičnega strežnika REST smo pokazali možnost razvoja orodij, ki poenostavijo razvoj aplikacij. Pokazali smo, da je že danes mogoče uporabiti refleksijo na način, kot je to v programskem jeziku Java in C#, če uporabimo refleksijsko knjižnico Jobjects++ in registriramo razrede, ki bi jih radi refleksirali. Knjižnica Jobjects++ gre pri refleksiji v času izvajanja še korak dlje, saj lahko pridobivamo in spreminjamo vrednosti razrednih polj z uporabo nizov, torej nam v času prevajanja tip vrednosti razrednega polja, ki jo pridobivamo oziroma spreminjamo, ni potrebno specificirati. Enako velja za klicanje metod, saj lahko metodi *invoke* v času prevajanja namesto vrednosti tipov, ki jih klicana metoda pričakuje, posredujemo nize, ki po dogovoru predstavljajo vrednosti tipov, ki jih klicana metoda pričakuje, vrnila pa nam bo vrnjeno vrednost klicane metode v obliki niza. Pri klicanju prekrivnih metod se bo v času izvajanja izbrala tista metoda, ki je najbolj ustrezna glede na število argumentov in vsebino nizov, pri čemer pa moramo biti zaradi dvoumnosti pazljivi, saj nizi, ki vsebujejo vrednosti določenih tipov, ne vsebujejo informacije o tipih.

Sklepamo lahko, da bomo ob prihodu novega standarda C++, po katerem bo programski jezik C++ postal refleksijski programski jezik, lahko preoblikovali knjižnico Jobjects++ na način, da registracija razredov z makroji ne bo več potrebna. Tako nam ne bo več potrebno specializirati konstruktor oziroma registracijsko metodo šablonskega razreda *CClass* za vsak refleksibilen

razred posebej, ampak bomo lahko napisali generično registracijsko metodo šablonskega razreda *CClass*, ki se bo s pomočjo refleksije v času prevajanja prevedla v ekvivalentno programsko kodo kot danes z eksplicitnimi specializacijami konstruktorja razreda *CClass*, zaradi katerih uporabljamo makroje.

Literatura

[1] A modern compile-time reflection library for C++. Dosegljivo: `https://github.com/veselink1/refl-cpp`, 2020. [Dostopano: 9. 10. 2020].

[2] Matúš Chochlík. Portable reflection for C++ with mirror. *Journal of information and organizational sciences*, 36(1):13–26, 2012.

[3] Matúš Chochlík, Axel Naumann, and David Sankel. [p0194r2] static reflection. *Proposal for ISO/IEC JTC1 SC22 WG21 N*, 3996:2014, 2016.

[4] Matúš Chochlík, Axel Naumann, and David Sankel. [p0385r2] static reflection–rationale, design and evolution. *International Organization for Standardization JTC1/SC22/WG21/SG7, Proposal*, 2017.

[5] CppAst. Dosegljivo: `https://github.com/foonathan/cppast`, 2019. [Dostopano: 9. 10. 2020].

[6] Creating a restful root resource class. Dosegljivo: `https://javaee.github.io/tutorial/jaxrs002.html`, 2017. [Dostopano: 9. 10. 2020].

[7] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *Proceedings of the IJCAI*, volume 95, pages 29–38, 1995.

[8] Experimental C++ features. Dosegljivo: `https://en.cppreference.com/w/cpp/experimental`, 2020. [Dostopano: 9. 10. 2020].

[9] Hibernate faq. Dosegljivo: https://developer.jboss.org/docs/DOC-15785#jive_content_id_But_Hibernate_uses_so_much_runtime_reflection, 2015. [Dostopano: 9. 10. 2020].

[10] Klemen Marković. Jobjects++. Dosegljivo: https://github.com/klemenm81/reflection.git, 2020. [Dostopano: 9. 10. 2020].

[11] Jacques Malenfant, Marco Jacques, and Franois Nicolas Demers. A tutorial on behavioral reflection and its implementation. In *Proceedings of the Reflection*, volume 96, pages 1–20, 1996.

[12] Axel Menzel. Run time type reflection. Dosegljivo: https://github.com/rttrorg/rttr, 2020. [Dostopano: 9. 10. 2020].

[13] Odb: C++ object-relational mapping (orm). Dosegljivo: https://www.codesynthesis.com/products/odb/, 2020. [Dostopano: 9. 10. 2020].

[14] Reflection in C++ part 1: The present. Dosegljivo: https://gracicot.github.io/reflection/2018/04/03/reflection-present.html, 2018. [Dostopano: 9. 10. 2020].

[15] David Sankel. The C++ reflection ts. Dosegljivo: https://www.youtube.com/watch?v=VMuML6vLSus, 2019. [Dostopano: 9. 10. 2020].

[16] David Sankel and Daveed Vandevoorde. [p1733] user-friendly and evolution-friendly reflection: A compromise. *International Organization for Standardization JTC1/SC22/WG21/SG7, Proposal*, 2019.

[17] Spring dependency injection. Dosegljivo: https://www.journaldev.com/2410/spring-dependency-injection, 2019. [Dostopano: 9. 10. 2020].

[18] Andrew Sutton. Compile time introspection of source code. Dosegljivo: https://www.youtube.com/watch?v=ARxj3dfF_h0, 2019. [Dostopano: 9. 10. 2020].

[19] Andrew Sutton and Herb Sutter. [p0590r0] a design for static reflection. *International Organization for Standardization JTC1/SC22/WG21/SG7, Proposal*, 2017.

[20] Andrew Sutton, Faisal Vali, and Daveed Vandevoorde. [p1240r0] scalable reflection in C++. *International Organization for Standardization JTC1/SC22/WG21/SG7, Proposal*, 2018.

[21] The draft C++ reflection technical specification. Dosegljivo: `https://github.com/cplusplus/reflection-ts`, 2020. [Dostopano: 9. 10. 2020].

[22] Daveed Vandevoorde and Louis Dionne. Exploring the design space of metaprogramming and reflection. C++ standards committee papers. *International Organization for Standardization JTC1/SC22/WG21/SG7, Proposal*, 2017.

[23] Reflection (computer programming). Dosegljivo: `https://en.wikipedia.org/wiki/Reflection_(computer_programming)`, 2019. [Dostopano: 9. 10. 2020].